CS 523 – BDT
Big Data Technology

Lesson 11

Apache Spark

The fastest ride has the greatest comfort

Maharishi International
University

Spark Motivation

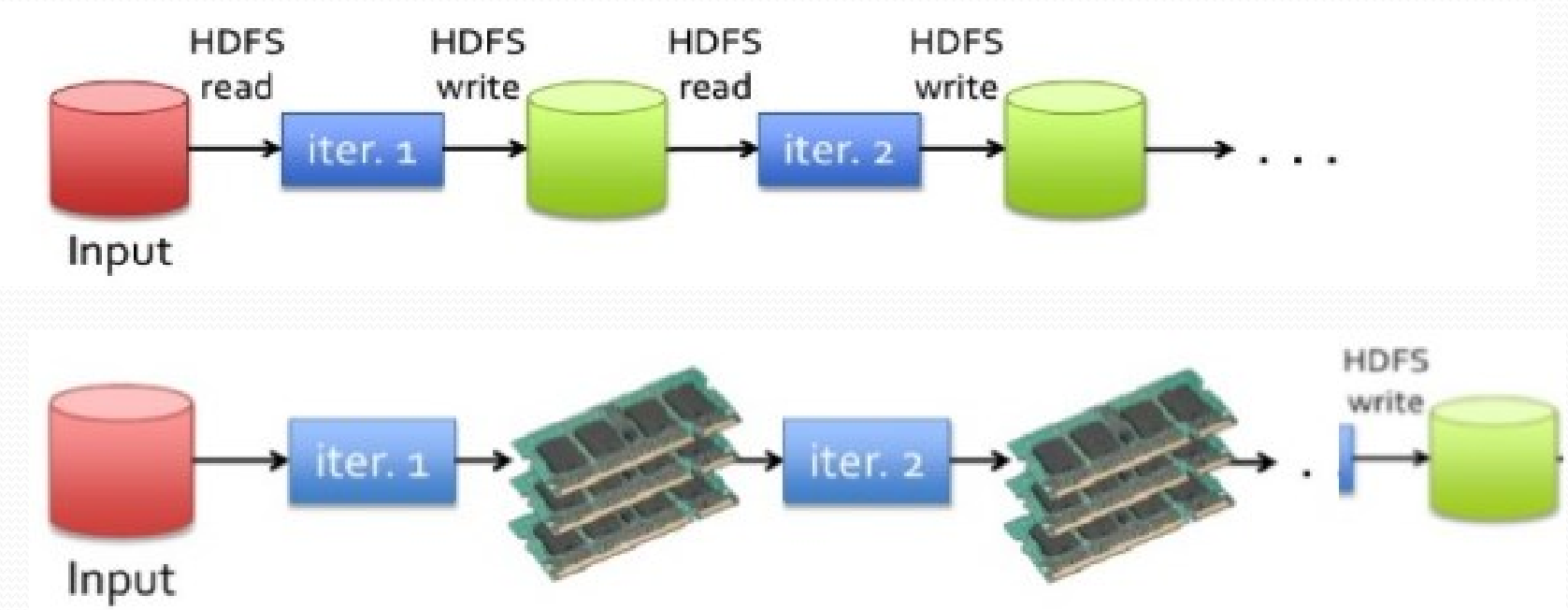
- **Shortcomings of MR**
  - MR Suitable only for simple batch jobs
- As big data analytics evolved, there was a need for both more complex multi-stage applications (e.g. machine learning algorithms) and more interactive ad-hoc queries.
- Matei Zaharia created Spark for his PhD at UC Berkeley, in response to the limitations he had seen in MR while working in summer internships at early Hadoop users, including Facebook.
- Spark makes efficient use of memory and can execute equivalent jobs 10 to 100 times faster than Hadoop's MapReduce.

Apache Spark

- Fast and general-purpose engine for large scale distributed 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.
- Implemented in Scala, which is a JVM-based language having excellent integration with Java.
- Has rich set of APIs in Java, Scala, Python and R for performing many common data processing tasks, such as joins.
- It 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's idea but not implementation. It has its own distributed runtime for executing work on a cluster.

MapReduce vs. Spark

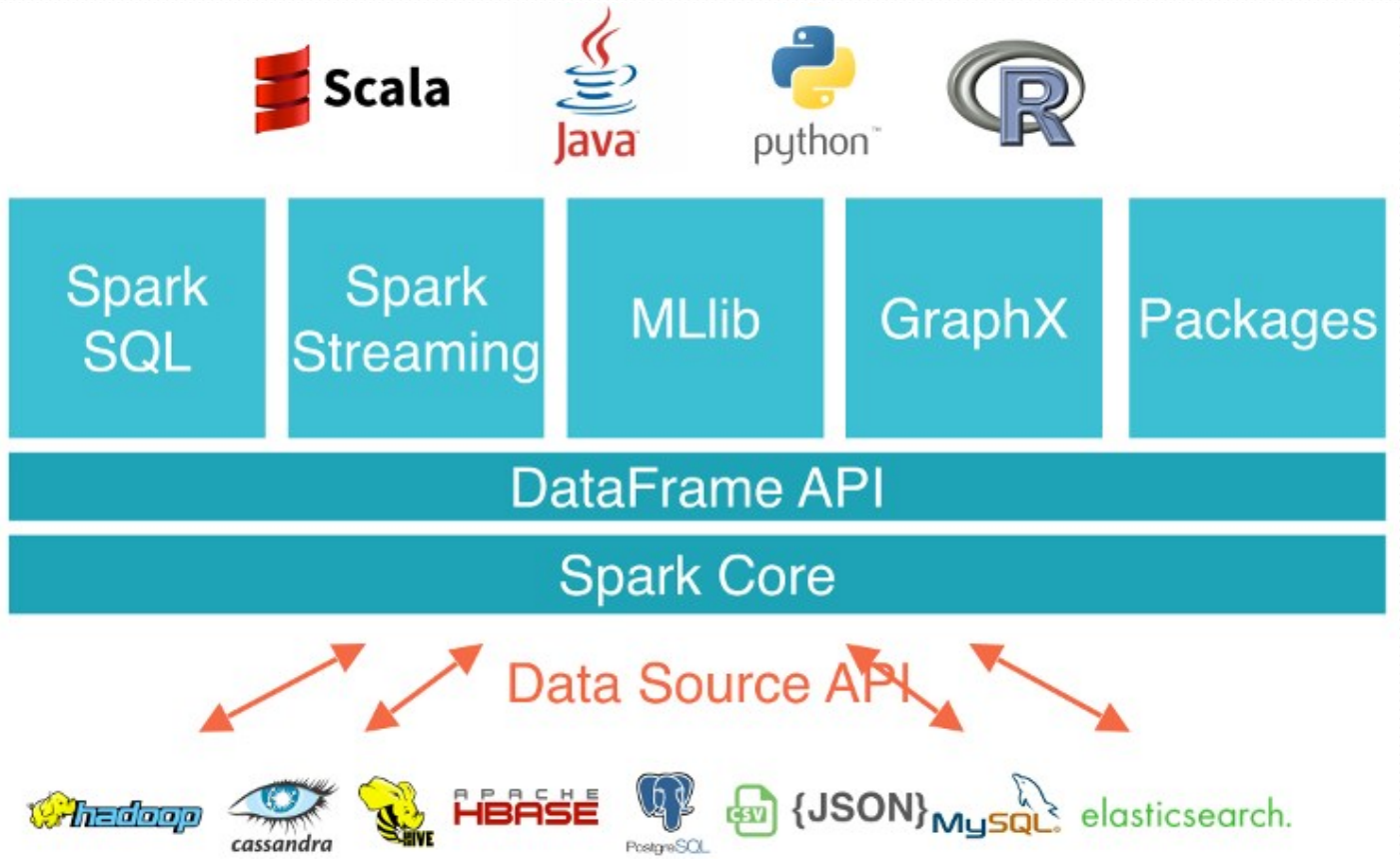
- Spark offers a far faster way to process data than passing it through multiple unnecessary Hadoop MapReduce processes.



- 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.

Spark Modules

- Spark consists of several purpose-built components - Spark Core, Spark SQL, Spark Streaming, Spark GraphX, and Spark MLlib.
- These components make Spark a feature-packed *unifying platform*: it can be used for many tasks that previously had to be accomplished with several different frameworks.

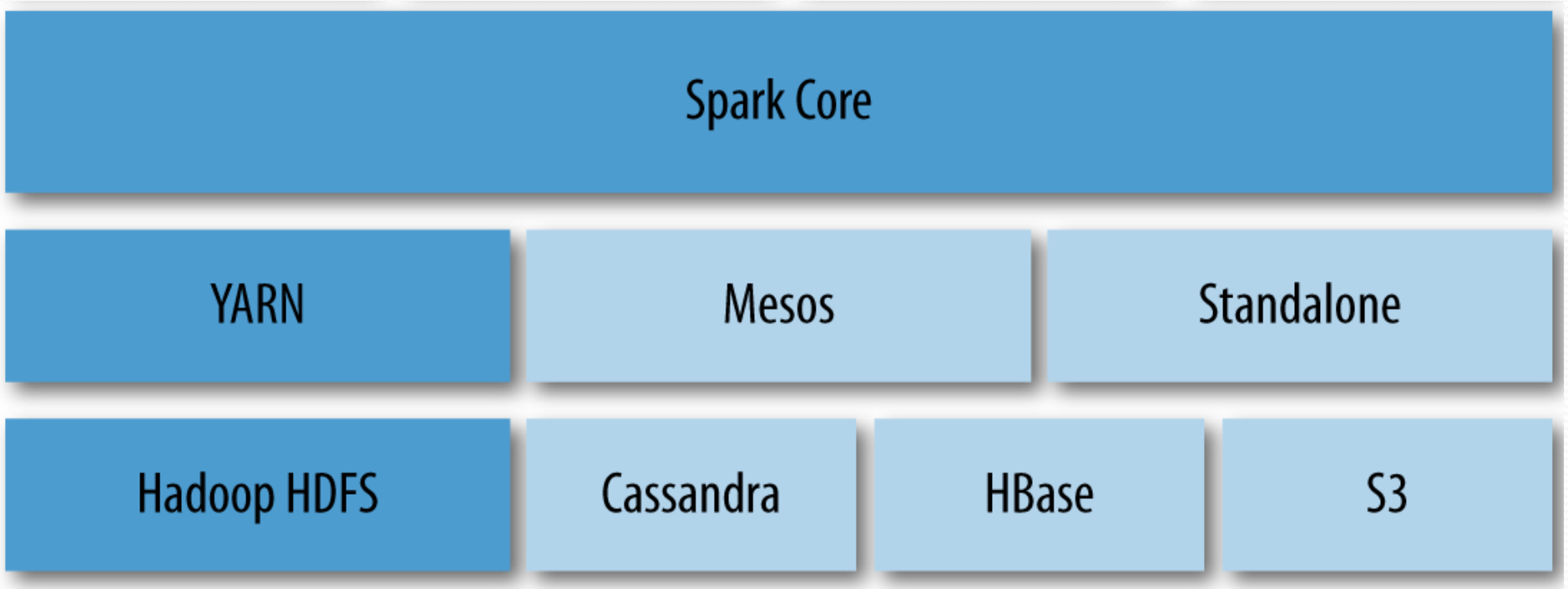


Spark Modules

- **Spark SQL** - It allows you to use SQL queries for structured data processing.
- **Spark Streaming** - It helps you to consume and process continuous data streams.
- **MLlib** - It is a machine learning library that delivers high-quality algorithms.
- **GraphX** - It comes with a library of typical graph algorithms.

- These are nothing but a set of packages and libraries. They offer APIs, DSLs, and algorithms in multiple languages. They directly depend on Spark Core APIs to achieve distributed processing.

Spark Deployment



Executing Spark Applications

- There are two different ways you can interact with Spark.

1. **Static compiled programs**
   - Write a program in Scala, Java or Python that uses Spark's library (API).
   - Submit the job using "spark submit" utility

2. **REPL Shell -** Scala shell, Python shell (pyspark), Notebooks
   - 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

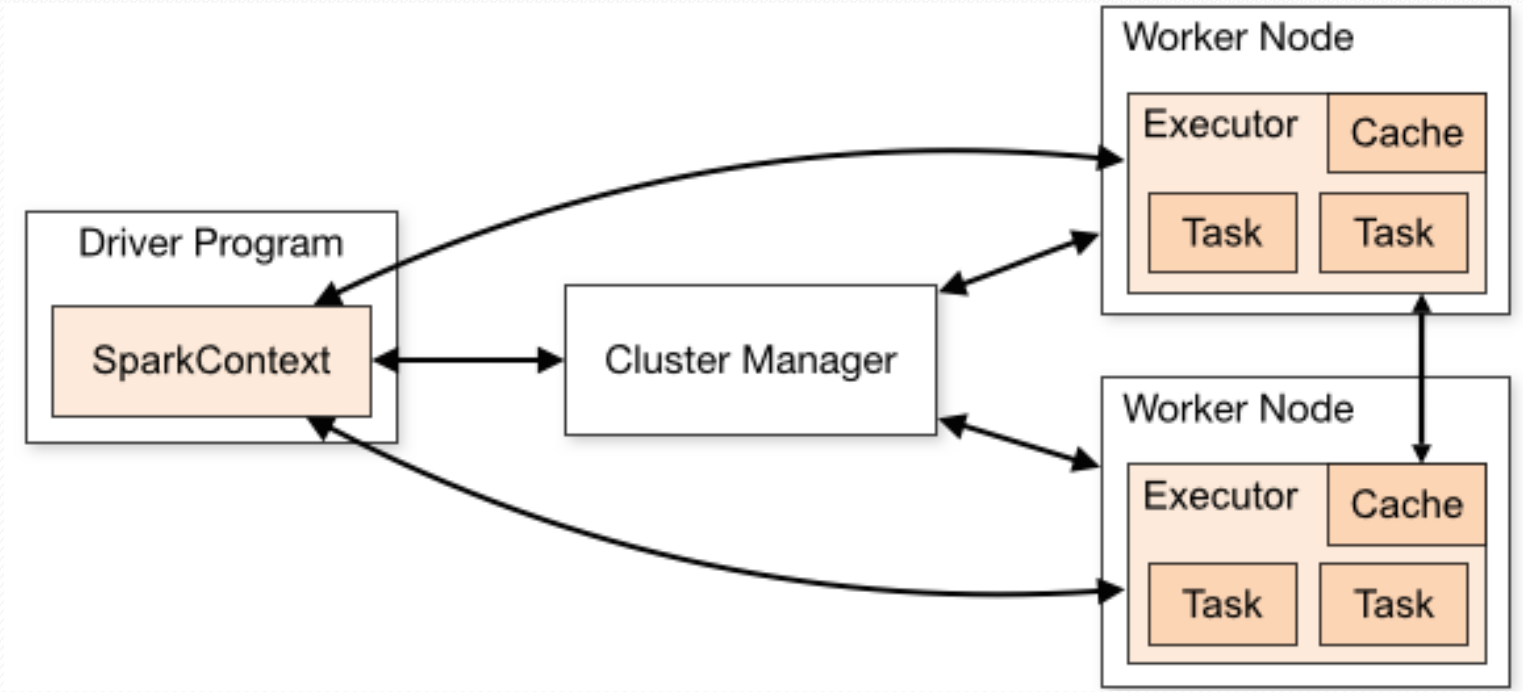
- *Spark REPL* - It reads your input, evaluates it, prints the result, and then does it all over again—that is, after a command returns a result, it doesn't exit the scala> prompt; it stays ready for your next command (thus *loop*).
- When you start the shell, the **SparkContext (sc)** and the **SQLContext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

```
$ 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.
```

Spark's Runtime Components

- Client
- Driver
- Executors
- Spark Context
- Cluster Manager

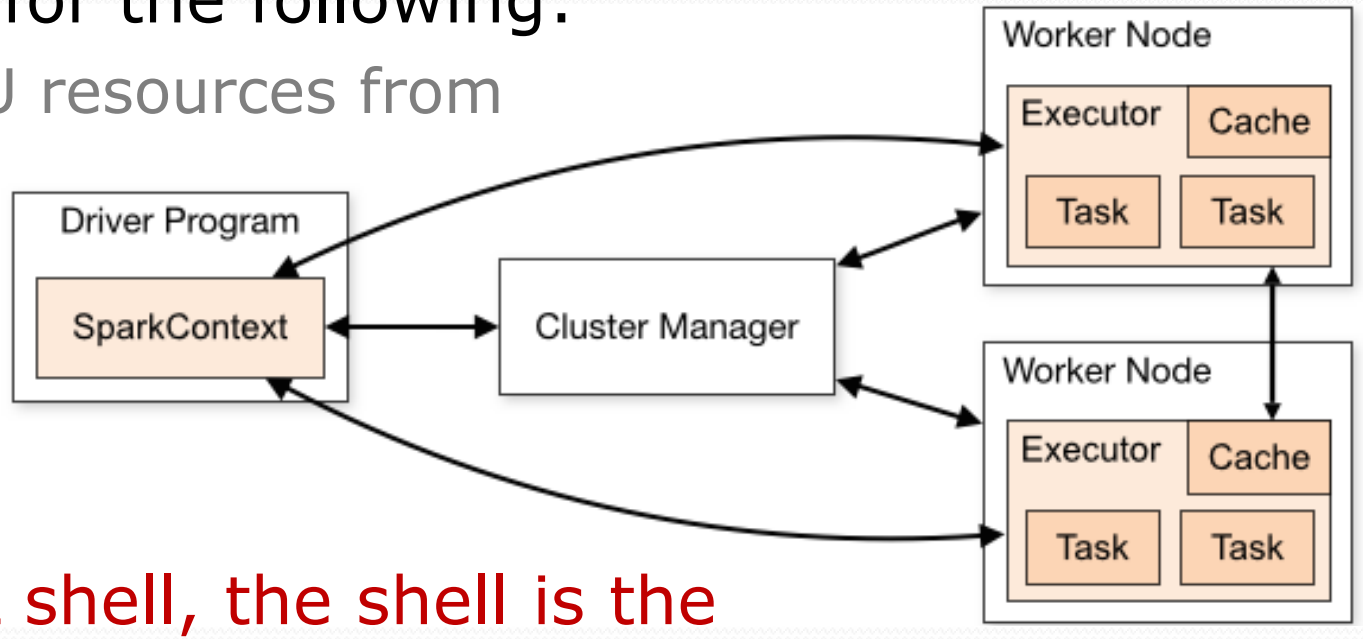


Spark's Runtime Components contd...

- **Client**: 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* starts the driver program and 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 creates one driver and a bunch of executors for each application.

Spark's Runtime Components contd...

- **Driver**: The driver orchestrates and monitors execution of a Spark application. It's like a wrapper around the application.
  - There is always one driver per Spark 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
  - When running a Spark REPL shell, the shell is the driver program.



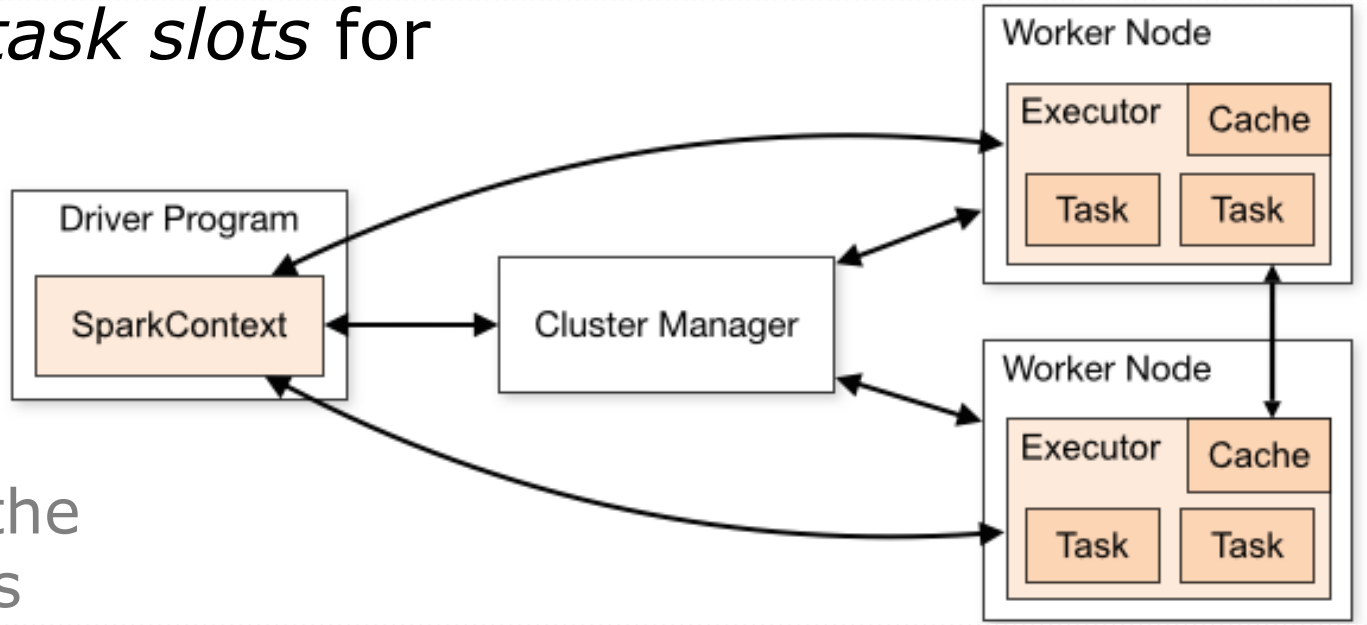
Spark Context

- Once the driver is started, it starts and configures an instance of SparkContext.
- SparkContext is the main interface for accessing Spark runtime which is used for connecting to Spark from an application, configuring a session, managing job execution, loading or saving a file, and so on.
  - Spark context is already preconfigured and available as an **sc** variable in Spark shell. (SQL Context is also available)
  - When using the spark-submit tool, the Spark application starts and configures the Spark Context.
- There can be only one Spark Context per JVM and it is thread-safe.

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

Spark's Runtime Components contd...

- **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.



Cluster Manager

- Cluster Manager is a pluggable component in Spark, to launch Executors and Drivers.
- Spark gets the resources for the Driver and the Executors from Cluster Managers.
- Spark can run on several types of clusters including Standalone. (Standalone is a very small cluster only running spark; good for development and testing)

1. Apache YARN
2. Apache Mesos
3. Kubernetes
4. Standalone

Spark Master

- The Master defines the **master** service of a **cluster manager** where spark will connect.
- The resources are allocated by **Master**. It uses the "Workers" running throughout the cluster for the creation of “Executors” for the "Driver". After that, the Driver runs tasks on Executors.

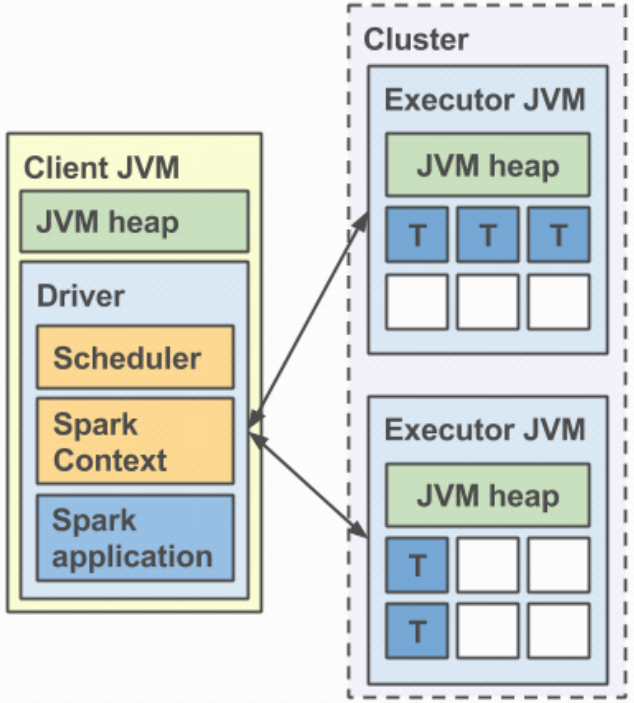
Value

- local. Start the standalone spark locally. (The SPARK_HOME environment variable gives the installation directory)
- spark://hostnameMaster:port. Connect to the Standalone Spark Cluster manager: Example: spark://myMachine:7077
- yarn. Connect to Yarn.
- k8s://HOST:PORT. Connect to the Kubernetes cluster manager

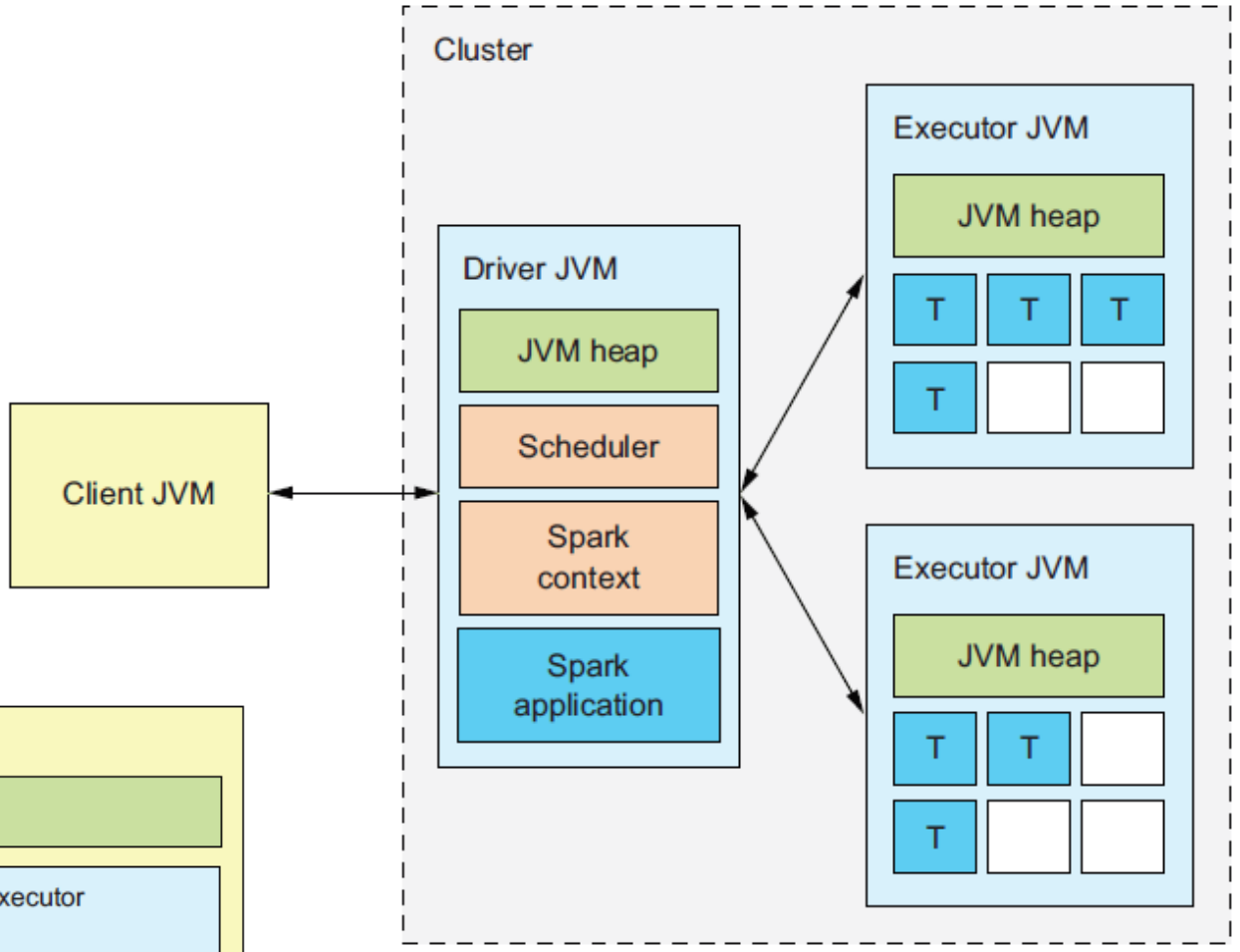
Spark Deploy Modes

- Deploy mode controls where each of the spark component will run, either on the client which runs **`spark-submit`**, or on one of the worker machines.
  1. Client Mode (Default)
  2. Cluster Mode
  3. Local Mode

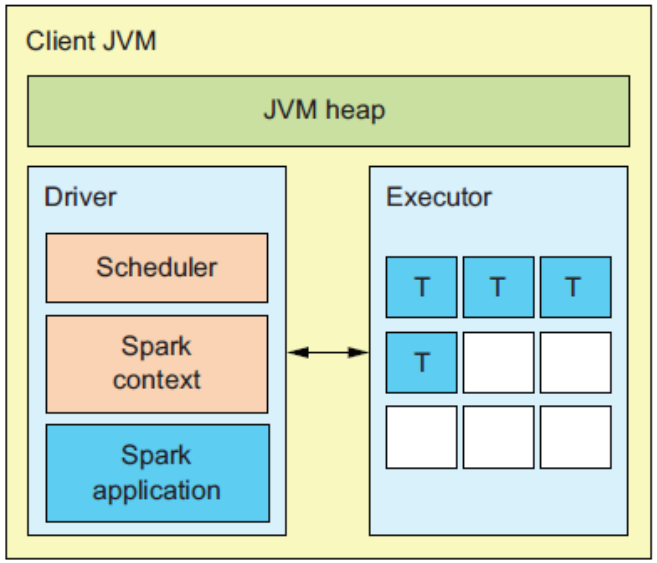
Client Mode



Cluster Mode



Local Mode



Spark Deploy Modes contd..

- **Client Mode** (Default) **–** Driver on client machine, Executors on cluster
  - Driver is launched directly within the spark-submit process which acts as a client to the cluster.
  - The input and output of the application is attached to the console. So this mode is good for debugging.
- **Cluster Mode –** Both Driver & Executors on cluster
  - Should be used if the job submission node (e.g. your laptop) is away from the worker nodes
  - Common to use cluster mode to minimize network latency between the drivers and the executors.
- **Local Mode –** Start everything in a single local JVM
  - Used mainly for testing & debugging purposes when you don't have access to a full cluster, or you want to try out something quickly.
  - There is only one executor in the same client JVM as the driver, but this executor can spawn several threads to run tasks.

Launching Applications with spark-submit

```
./bin/spark-submit \
  --class <main-class> \
  --master <master-url> \
  --deploy-mode <deploy-mode> \
  --conf <key>=<value> \
  ... # other options
  <application-jar> \
  [application-arguments]
```

Launching Applications with `spark-submit`

- **--class**: The entry point for your application
  (e.g. org.apache.spark.examples.SparkWC)
- **--master**: The master URL for the cluster
  (e.g. spark://23.195.26.187:7077)
- **--deploy-mode**: Whether to deploy your driver on the worker nodes (*cluster*) or locally as an external client (*client*) *(default: client)*
- **--conf**: Arbitrary Spark configuration property in key=value format. For values that contain spaces wrap "key=value" in quotes.
- **application-jar**: Path to a bundled jar including your application and all dependencies. The URL must be globally visible inside of your cluster, for instance, an *hdfs://* path or a *file://* path that is present on all nodes.
- **application-arguments**: Arguments passed to the main method of your main class, if any.

Run on a Spark Standalone Cluster in Client Deploy Mode

```
./bin/spark-submit \
  --class org.apache.spark.examples.SparkWC \
  --master spark://207.184.161.138:7077 \
  --executor-memory 20G \
  --total-executor-cores 100 \
  /path/to/examples.jar \
  1000
```

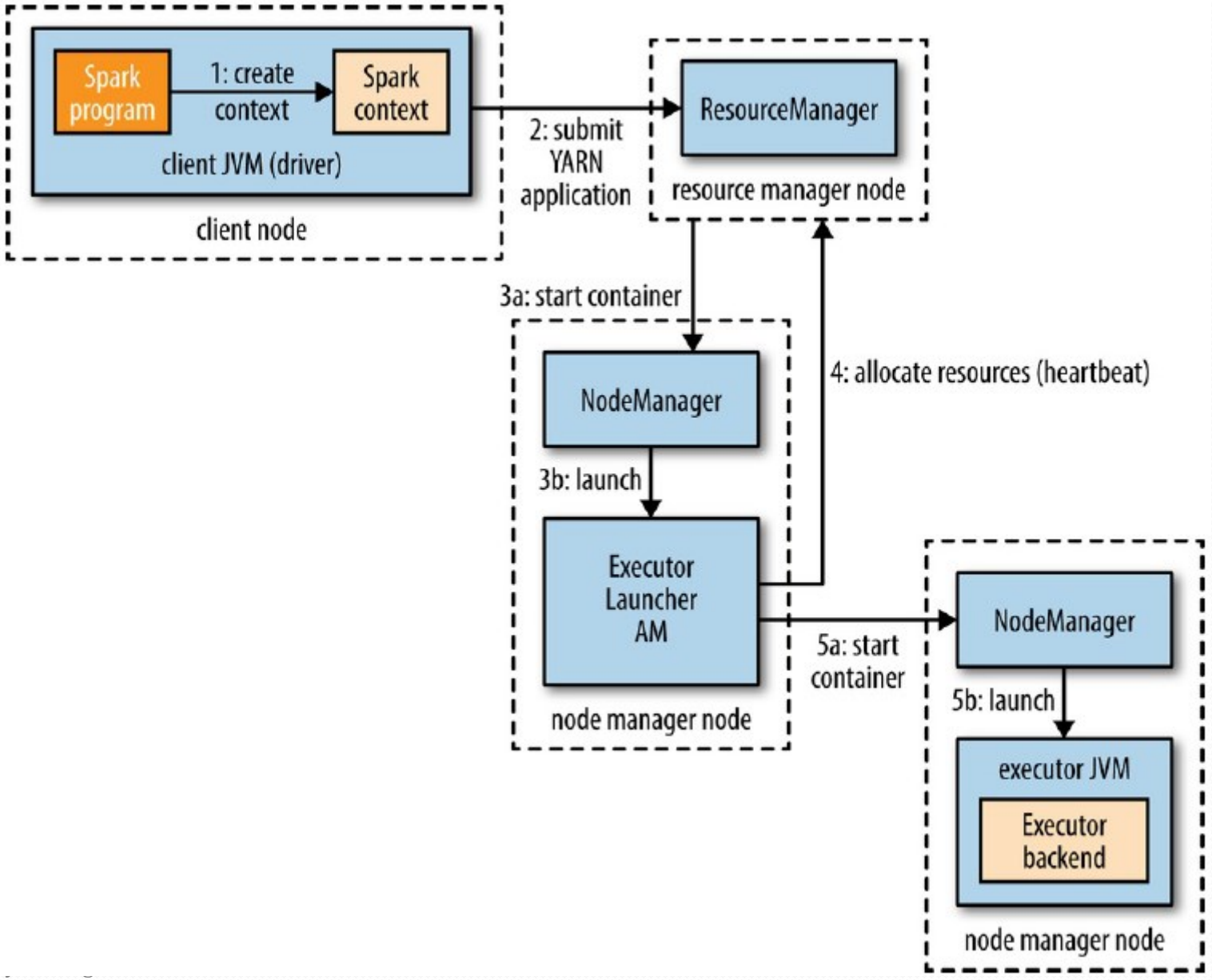
Run on a YARN cluster

```
./bin/spark-submit \
  --class org.apache.spark.examples.SparkWC \
  --master yarn \
  --deploy-mode cluster \  # can be client for client mode
  --executor-memory 20G \
  --num-executors 50 \
  /path/to/examples.jar \
  1000
```

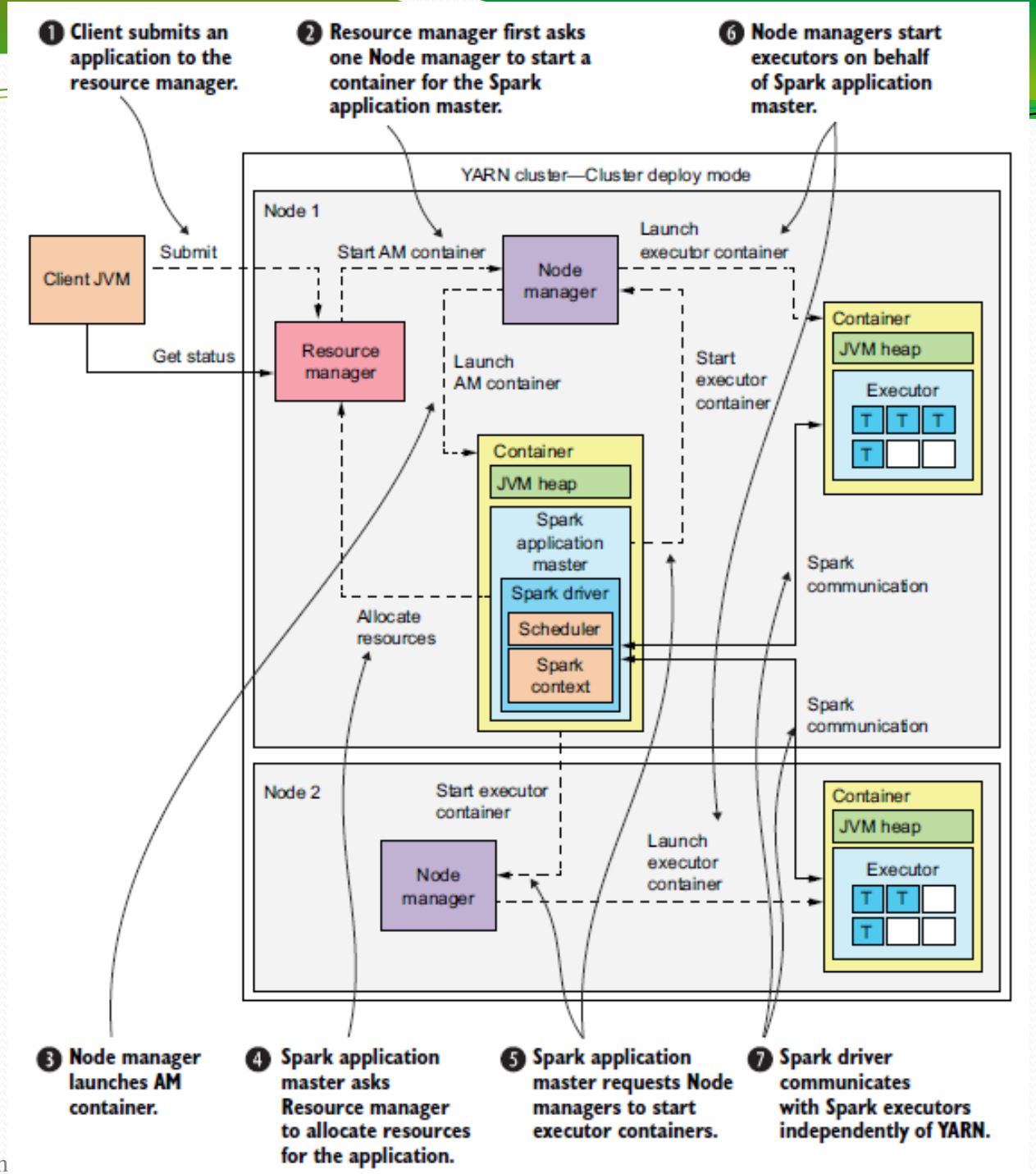
Spark on Yarn

- Running Spark on YARN provides the tightest integration with other Hadoop components and is the most convenient way to use Spark when you have an existing Hadoop cluster.
- Spark offers two deploy modes for running on YARN: ***YARN client* mode**, where the driver runs in the client, and ***YARN cluster* mode**, where the driver runs on the cluster in the YARN application master.
- **YARN client mode** is required for programs that have any interactive component, such as *spark-shell* or *pyspark*.
  - Client mode is also useful when building Spark programs, since any debugging output is immediately visible.
- **YARN cluster mode**, on the other hand, is appropriate for production jobs, since the entire application runs on the cluster, which makes it much easier to retain logfiles (including those from the driver program) for later inspection.
- The master URL is **yarn**.

YARN Client Mode



YARN Cluster Mode

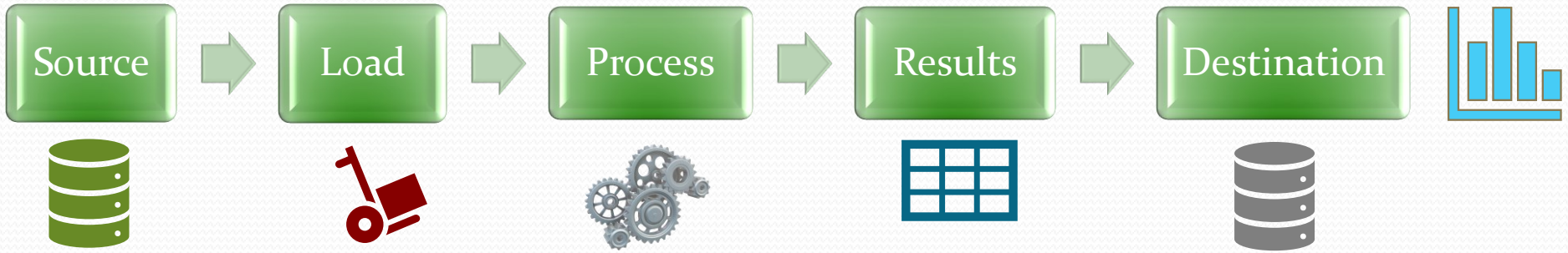


Local Mode

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 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 then (local[*]), local mode taking all CPU cores is assumed.

Typical Spark Application Process Flow



Data Structures used in the Process Flow

- **RDD**
- **Data Frames**
- **Data Set**

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)*
- 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.
- RDDs are built through parallel transformations (map, filter, reduce, groupBy etc.) - Generate RDD from other RDD
- Once created, RDDs never change
- Lazy operations that builds a DAG (Directed Acyclic Graph)
- Automatically rebuilt on failure using lineage
- Controllable persistence (RAM, HDFS, etc.)

Resilient Distributed Dataset contd...

- 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

RDD Creation

There are 2 ways to create RDD:

- Parallelizing an existing collection: E.g. array.
  - **Parallelized Collection**

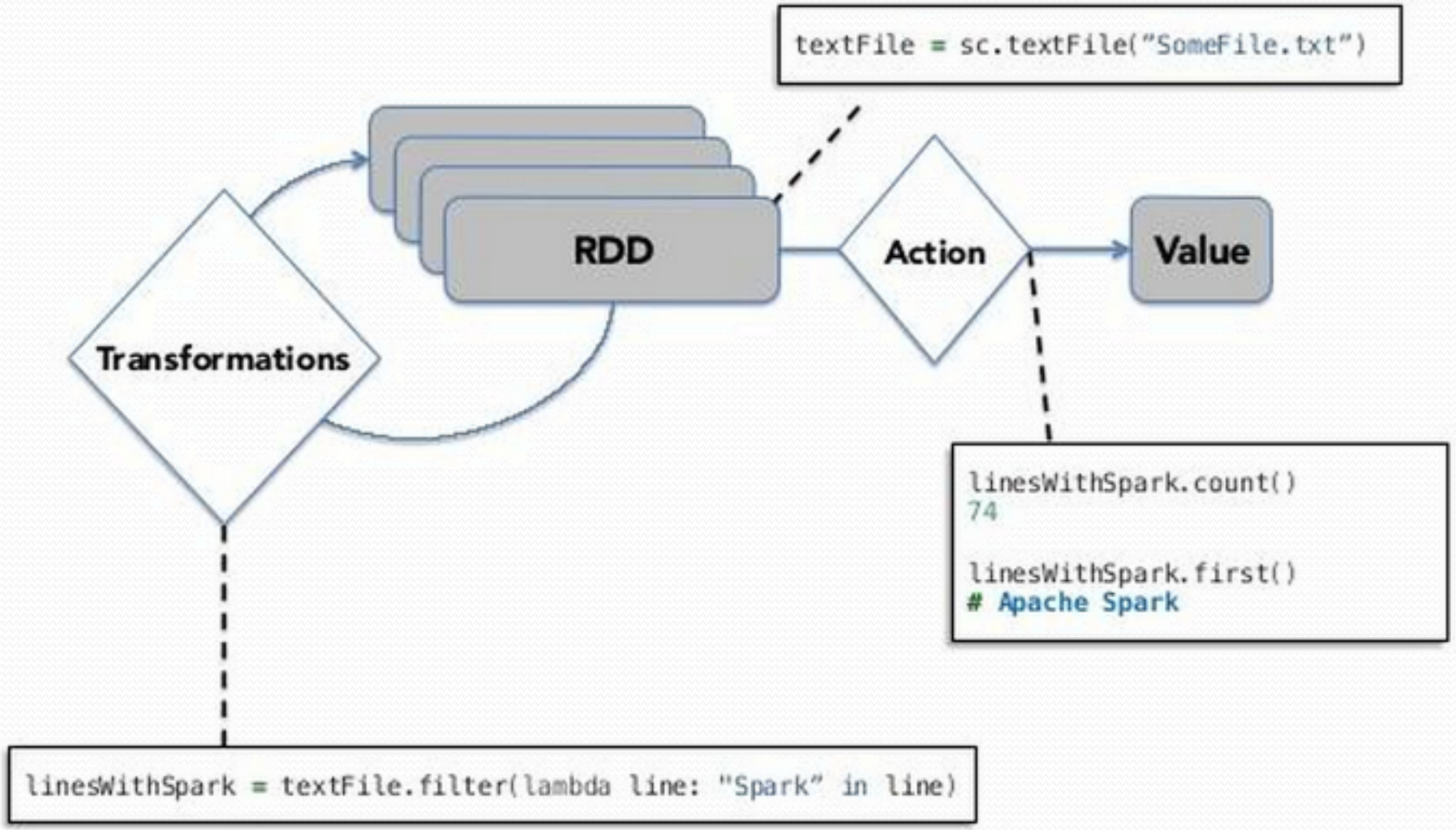
```
val data = Array (1,2,3,4)
val distData = sc.parallelize(data)
```

- Create from an existing storage: E.g. a file in local or distributed file system
  - **External Data Set**

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

Transformations and Actions

- **2 types of RDD operations**: **Transformations and Actions**
- 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 which triggers computation.
- **Transformation** - generates a new RDD from an existing one
  - Loading an RDD or performing a transformation on one does not trigger any data processing; it merely creates a plan for performing a computation.
  - Transformations are lazy
- **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.



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.

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(*otherDataset***)**	return a new dataset that contains the union of the elements in the source dataset and the argument
distinct([*numTasks***]))**	return a new dataset that contains the distinct elements of the source dataset

Transformations

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

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

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

```java
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 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 contd...

- 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.

Levels of Persistence

- Spark automatically persists intermediate data from various shuffle operations, however it is often suggested that users call *persist ( )* method on the RDD in case they plan to reuse it.
- *persist()* allows the user to specify the storage level whereas *cache()* uses the default storage level (MEMORY_ONLY).
- Spark has various persistence levels to store the RDDs on disk or in memory or as a combination of both with different replication levels.

Levels of Persistence

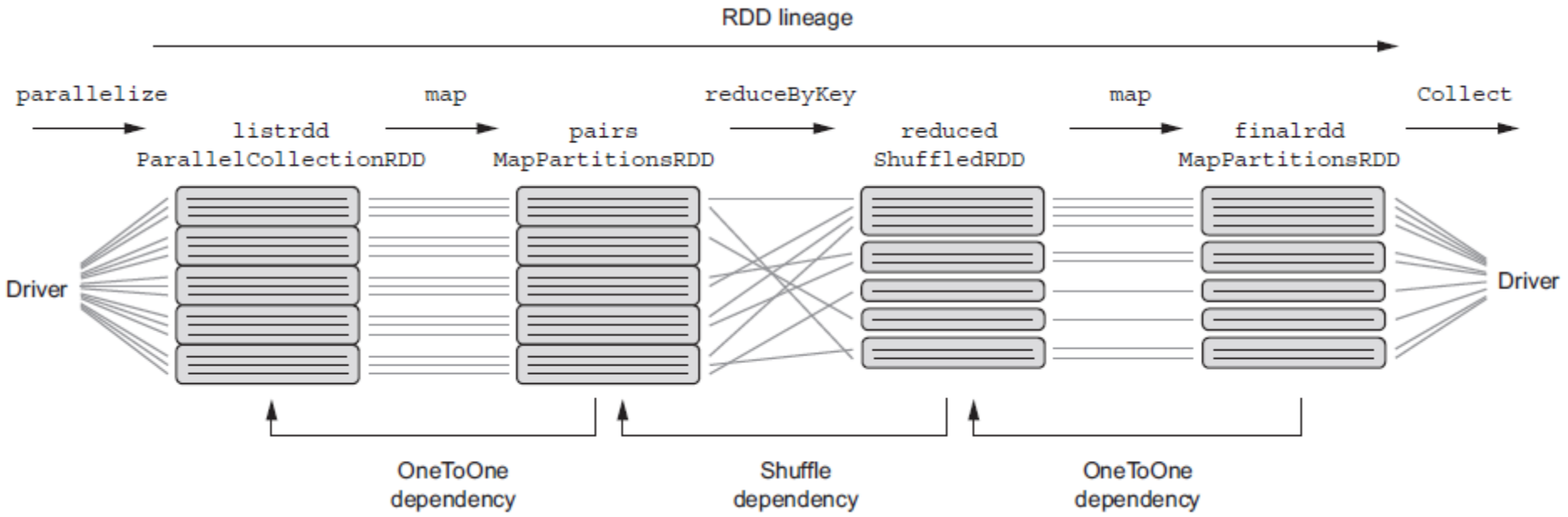
transformation	*description*
MEMORY_ONLY	Store RDD as deserialized Java objects in the JVM. If the RDD does not fit in memory, some partitions will not be cached and will be recomputed on the fly each time they're needed. This is the default level.
MEMORY_AND_DISK	Store RDD as deserialized Java objects in the JVM. If the RDD does not fit in memory, store the partitions that don't fit on disk, and read them from there when they're needed.
MEMORY_ONLY_SER	Store RDD as serialized Java objects (one byte array per partition). This is generally more space-efficient than deserialized objects, especially when using a fast serializer, but more CPU-intensive to read.
MEMORY_AND_DISK_SER	Similar to MEMORY_ONLY_SER, but spill partitions that don't fit in memory to disk instead of recomputing them on the fly each time they're needed.
DISK_ONLY	Store the RDD partitions only on disk.
MEMORY_ONLY_2, MEMORY_AND_DISK_2, etc	Same as the levels above, but replicate each partition on two cluster nodes.

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.

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.



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 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://...")
```

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(" ")))
                .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]);
```

Avoid GroupByKey

```
val words = Array("one", "two", "two", "three", "three", "three")
val wordPairsRDD = sc.parallelize(words).map(word => (word, 1))

val wordCountsWithReduce = wordPairsRDD
  .reduceByKey(_ + _)
  .collect()

val wordCountsWithGroup = wordPairsRDD
  .groupByKey()
  .map(t => (t._1, t._2.sum))
  .collect()
```

- While both of these functions will produce the correct answer, the *reduceByKey* example works much better on a large dataset.
- That's because Spark knows it can combine output with a common key on each partition before shuffling the data.

- On the other hand, when calling *groupByKey* - all the key-value pairs are shuffled around. This is a lot of unnecessary data that is being transferred over the network.
- Here are more functions to prefer over *groupByKey*:
  - *combineByKey* can be used when you are combining elements but your return type differs from your input value type.
  - *foldByKey* merges the values for each key using an associative function and a neutral "zero value".

Don't copy all elements of a large RDD to the driver

- If your RDD is so large that all of it's elements won't fit in memory on the driver machine, don't do this:
  - `val values = myVeryLargeRDD.collect()`
- Collect will attempt to copy every single element in the RDD onto the single driver program, and then run out of memory and crash.
- Instead, you can make sure the number of elements you return is capped by calling *`take`* or *`takeSample`*, or perhaps filtering or sampling your RDD.
- If you really do need every one of these values of the RDD and the data is too big to fit into memory, you can write out the RDD to files.
- Similarly, be cautious of these other actions as well unless you are sure your dataset size is small enough to fit in memory:
  - *`countByKey`*
  - *`countByValue`*
  - *`collectAsMap`*

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
    - pom.xml file
  - Java word count program

- 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