Introduction to Spark

Ángel J. Rey


AGENDA
1. Scope of the course
2. Spark core concepts
3. RDDs and Partitioning
4. Spark Architecture
5. Spark SQL and DataFrames
6. Spark Streaming


INTRODUCTION TO SPARK
1. Scope of the course

Scope of the course

- Introduction to Spark with:
  - Lectures
  - Hands-on exercises
  - Virtual Machine
  - And much more …
- Course Materials available in GitHub

https://github.com/chicochica10/spark-scala-developer

Contact me if you need help !!


INTRODUCTION TO SPARK
2. Spark core concepts

2. Spark Core Concepts

- What is Apache Spark
- Basic Concepts
- The Spark Shell

Exercise 0

- Prepare the Environment

What is Apache Spark

- Apache Spark is a distributed computing platform designed to be **fast** and **general purpose**
- Spark extends the **map-reduce** model to efficiently support more types of **in-memory** computations, including **interactive queries** and **stream processing**

What is Apache Spark

- From the **general purpose** Spark is designed to cover a wide range of workloads that previously required separate distributed systems:
  - batch applications
  - iterative algorithms
  - streaming

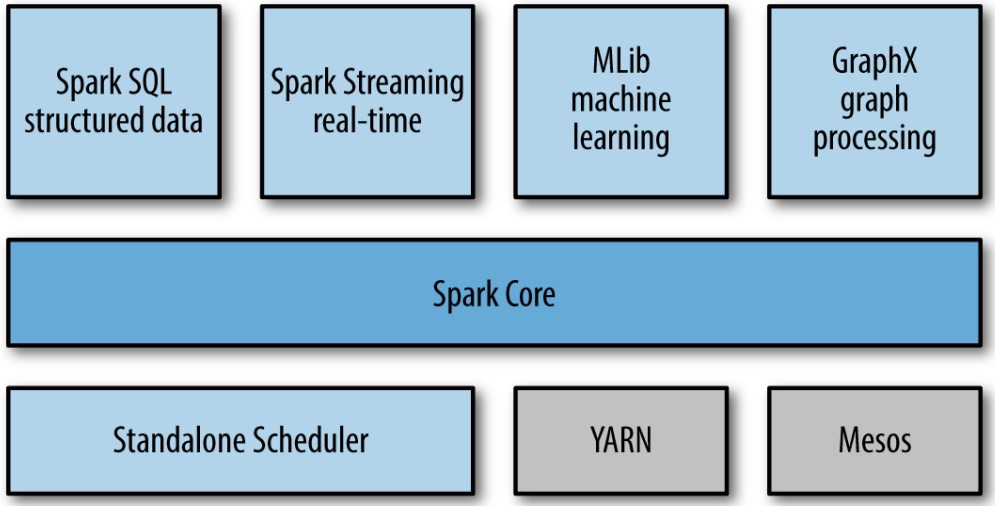
What is Apache Spark

- Spark is designed to be highly accessible, offering simple APIs
  - Python, Java, Scala, R, SQL

- Spark can run in Hadoop clusters and access any Hadoop data source

Unified Stack

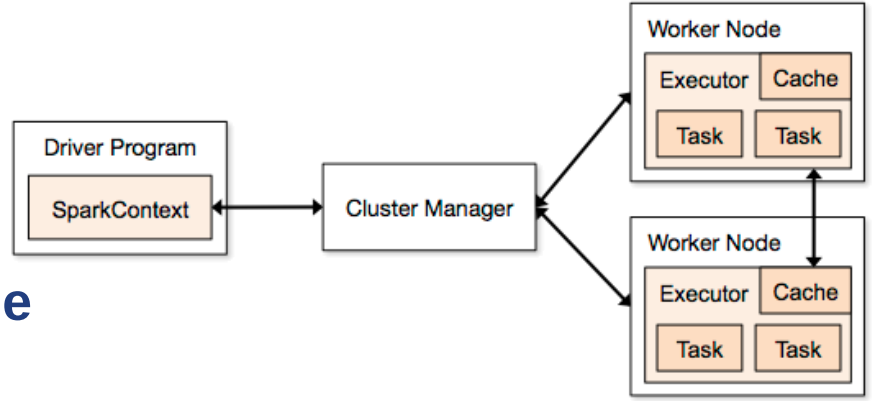
- **Core**:
  - scheduling, distributing and monitoring computational tasks across nodes
- **High-level components:**
  - specialized for various applications



- **SQL:** allows querying data via SQL, Hive and Data Frames
- **Stream processing:** processing of data live streams (events, logs, …)
- **Machine Learning, graphs:** for parallel computation of machine learning algorithms: classification, regression, clustering, etc…

Spark cluster infraestructure

- Spark is an Open-source cluster computing framework
- Requires **cluster manager**
  - Standalone – simple cluster manager included in Spark
  - Apache Mesos – general, large scale cluster manager
  - Hadoop YARN – resource manager for Hadoop
- Distributed **storage system**
  - HDFS (Hadoop Distributed File System
  - Amazon S3
  - Cassandra, OpenStack Swift, Tachyon
- Also supports **pseudo-distributed mode**
  - Development and testing
  - local file system, one worker per CPU core

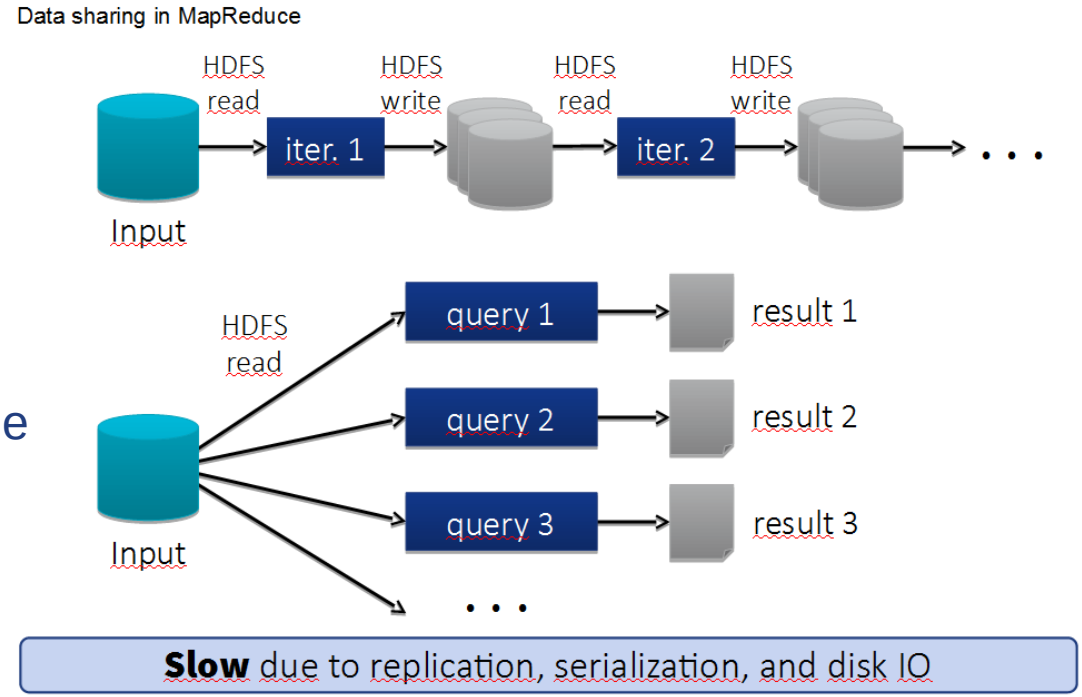


Brief History

- **2009** - Initially started at UC Berkeley AMPLab
  - https://amplab.cs.berkeley.edu/
- **2010** - Open sourced under a BSD license
- **2013** - Donated to the Apache Software Foundation
  - Apache 2.0 license
- **Feb 2014** - Became an Apache Top-Level Project
- **Nov 2014** - Databricks set a new world record in large scale sorting using Spark
  - 100TB (1 trillion records) sorted in 23 minutes, 206 EC2 machines
  - Previously: Hadoop MapReduce, 72 minutes, 2100 machines
  - 3X faster, 10X fewer machines
  - Sorting on HDFS (disk), not using in-memory cache!

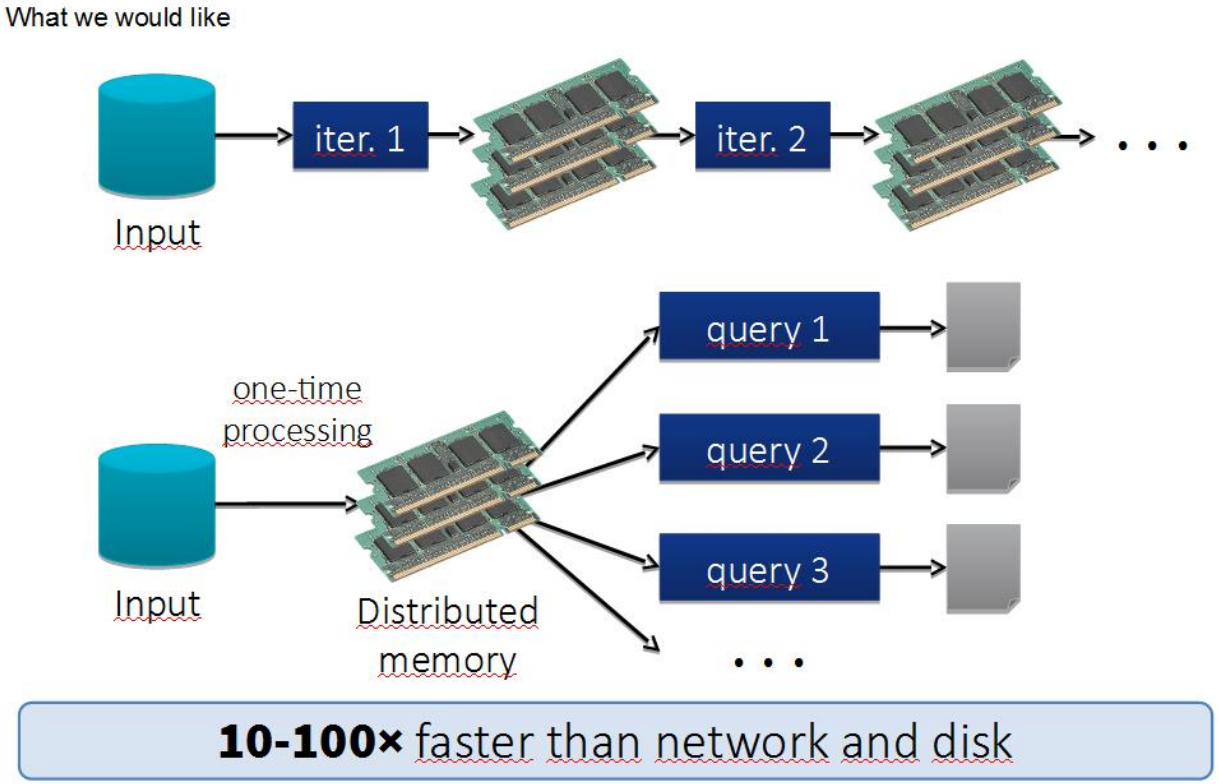
Problems with Hadoop Map-Reduce

- **Batch oriented**
  - So, Hive, Pig and all MR-based systems too
  - Great throughput but high latency
  - Not for BI tools
  - Not for real-time (stream) processing
- **Programming model** not good for iterative programming (*e.g.* machine learning)
  - Forces to concatenate multiple jobs
  - External orchestration
  - Each job flushes to disk
  - Data sharing requires external storage



Slow due to replication, serialization, and disk IO

Spark Alternative



Spark vs Map-Reduce

MapReduce vs Spark

- Only batch oriented
- Maintainability
  - Tedious to add/modify functionality
- Complexity (more LoC)
- Needs explicit Orchestration
- Testing somehow tedious with a lot of Integration Tests

- In memory processing
- Exploration
  - Prototyping, PoC, etc.
- Speed of development
- Orchestration natively done in the code.
- Testing

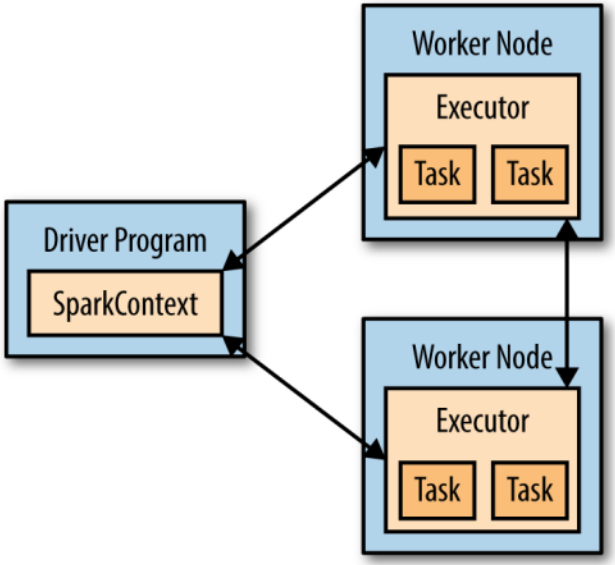
Multiple Map-Reduce and other operations can be managed in the same job

Exercise 1

- Word count Map Reduce

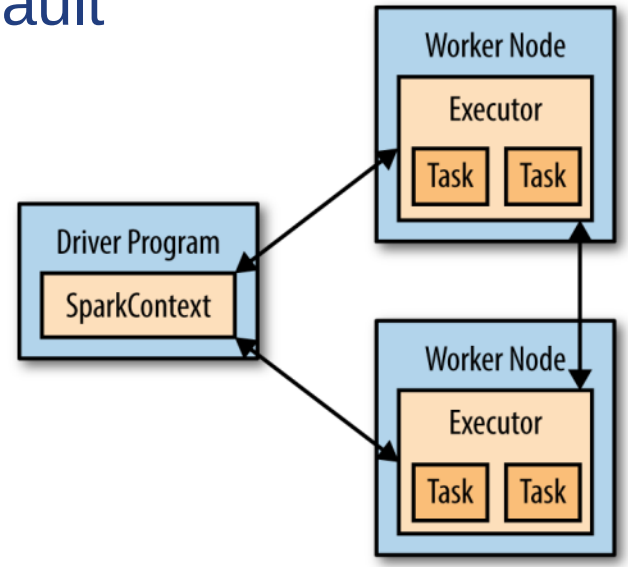
Basic Concepts

- Every Spark application consists of a **driver program** that defines **distributed datasets** on a cluster and then launches various **parallel** operations on them
- The **driver** can be:
  - a **shell** that you can control interactively
  - a **program** in your local machine
  - a **task** in the cluster



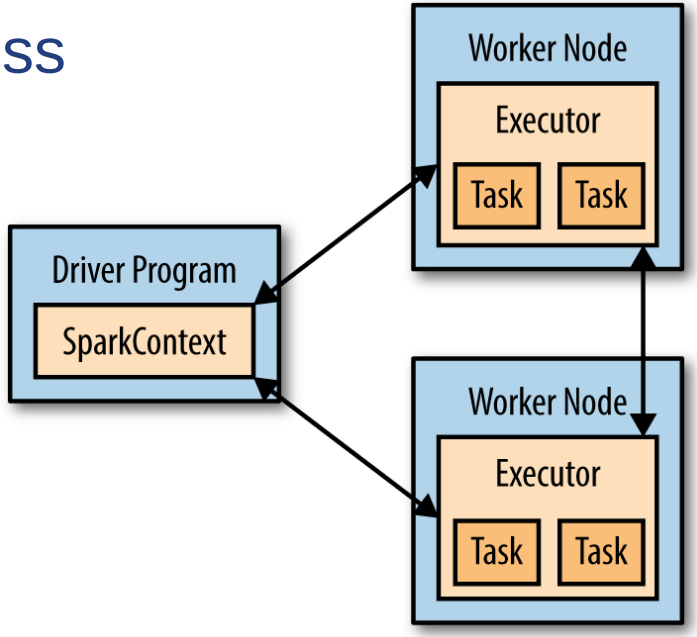
Basic Concepts

- The **driver** accesses Spark through a **SparkContext object** which encapsulates the connection to the cluster
- In the **shell** this object is created by default
- The **sc** object



Basic Concepts

- In Spark computation on data is expressed as **operations** on distributed collections that are automatically parallelized across the cluster, called RDDs (Resilient Distributed Datasets).
- To run these operations, the driver typically manages a number of nodes, called executors
- The basic components are →



Using the Spark-Shell

```
$ spark-shell
```

```
$ spark-shell --master yarn
```

```
$ spark-shell --master local[4] --jars code.jar
```

```
vagrant@master1:~$ spark-shell
Welcome to
      ____              __
     / __/__  ___ _____/ /__
    _\ \/ _ \/ _ `/ __/  '_/
   /___/ .__/\_,_/_/ /_/\_\   version 1.1.0
      /_/

Using Scala version 2.10.4 (Java HotSpot(TM) 64-Bit Server VM, Java 1.7.0_67)
Type in expressions to have them evaluated.
Type :help for more information.
15/01/21 16:05:38 INFO SparkILoop: Created spark context..
Spark context available as sc.

scala> val file = sc.textFile("file.txt")
scala> file.count()
res1: Int = 10
```

Spark Operations Example

```
// Create an RDD called lines
scala> val lines = sc.textFile(“input/core-site.xml")
lines: spark.RDD[String] = MappedRDD[...]
scala> lines.count() // Count the number of items in this RDD
res0: Long = 127
scala> lines.first() // First item in this RDD, i.e. first line of README.md
res1: String = <?xml version="1.0"?>

scala> val hadoopLines = lines.filter(line => line.contains(“hadoop"))
hadoopLines: spark.RDD[String] = FilteredRDD[...]
scala> hadoopLines.first()
res2: String = "    <name>hadoop.proxyuser.oozie.hosts</name>"
```

Building your own application

- When building your own driver program, the spark context must be defined

```
import org.apache.spark.SparkConf;
import org.apache.spark.api.java.JavaSparkContext;
SparkConf conf = new SparkConf().setMaster("local").setAppName("My App");
JavaSparkContext sc = new JavaSparkContext(conf);
// …
sc.stop()
```

A *cluster URL*. in this example **local** tells Spark how to connect to a cluster. **local** is a special value that runs Spark on one thread on the local machine, without connecting to a cluster.

An *application name*, **My App** in this example. This will identify your application on the cluster manager's UI if you connect to a cluster

Building your own application

- For compiling add Spark core dependencies in the build.sbt project file

```
// additional libraries
libraryDependencies ++= Seq(
"org.apache.spark" %% "spark-core" % "1.5.0" % "provided"
)
```

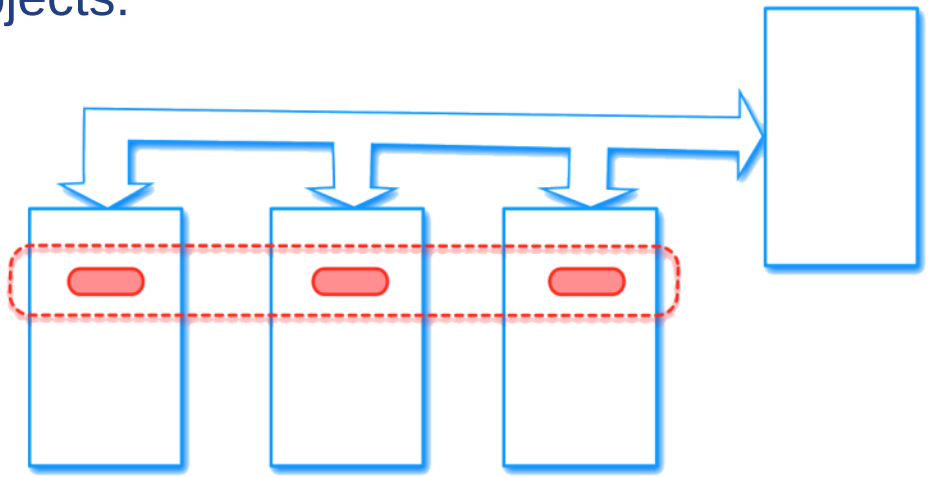
Exercise 2

- Using the Spark shell


INTRODUCTION TO SPARK
3. RDDs and Partitioning

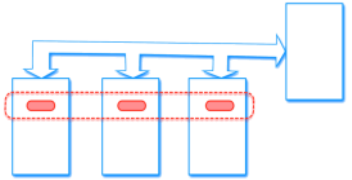
RDD Definition

- An RDD in Spark is simply an immutable distributed collection of objects.
- Each RDD is split into **multiple *partitions***, which may be computed on different **nodes** of the cluster.
- RDDs can contain any type of objects:
  - Python
  - Java
  - R
  - Scala
  - user defined classes



RDD Creation

- RDDs can be created in **three ways**:



1. By loading from an external dataset in HDFS, S3, local file, etc…

```
scala> val lines = sc.textFile("README.md")
```

2. Parallelizing a collection of objects already existing in the memory of the driver / workers

```
scala> val input = sc.parallelize(List(1, 2, 3, 4))
```

3. As a result of a transformation from another RDD

```
scala> val rdd2 = rdd1.filter(line => line.contains("error"))
```

RDD Basics

- Once created, RDDs offer two types of operations:
  - ***Transformations*** construct a new RDD from a previous one. For example, one common transformation is filtering data that matches a predicate.

```
val pythonLines = lines.filter(line => line.contains("Python"))
```

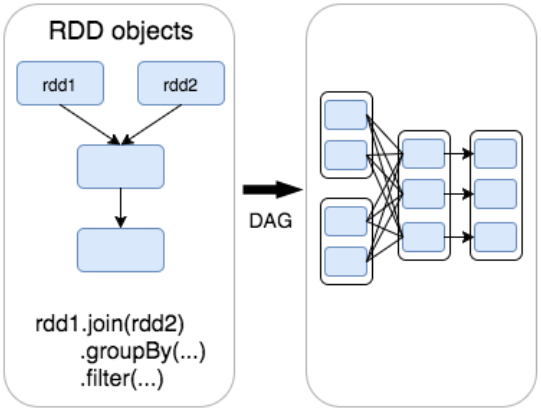
  - ***Actions*** compute a result based on an RDD, and either return it to the driver program or save it to an external storage system (e.g., HDFS).

```
scala> lines.first()
```

RDD Basics

- Sparks computes **RDDs in a Lazy way**. It sees the **whole chain** of transformations and compute just the data needed for its result the first time they are used in an action.
- Spark's **RDDs** are by **default recomputed** each time you run an action on them.
- If you would like to reuse an RDD in **multiple actions**, you can ask Spark to ***persist*** it using `RDD.persist()`.
- Example:

```
scala> pythonLines.persist
scala> pythonLines.count()
2
scala> pythonLines.first()
u'## Interactive Python Shell'
```

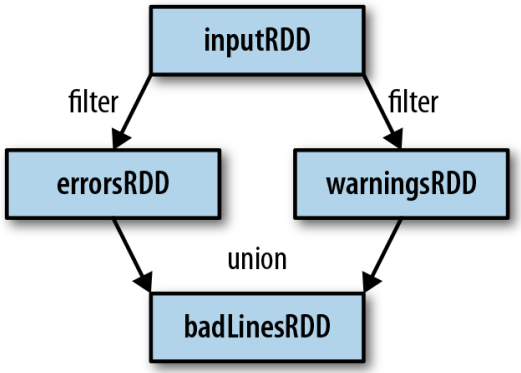


Transformations

- Lazy operations, returning new RDD

This operation does not mutate the existing **inputRDD**.
Returns a pointer to an entirely new RDD.
inputRDD can be reused later

```
val inputRDD = sc.textFile("log.txt")
val errorsRDD = inputRDD.filter(line => line.contains("error"))
val warningsRDD = inputRDD.filter(line => line.contains("warning"))
val badLinesRDD = errorsRDD.union(warningsRDD)
```



- Spark keeps track of the set of dependencies between different RDDs, called the ***lineage graph***.
- It uses this information to compute each RDD on demand and to recover lost data if part of a persistent RDD is lost.

Actions

- Operations that return a final value to the driver program or write data to an external storage system.
- Force the evaluation of the transformations required for the RDD
- Example: counting errors

```
println("Input had " + badLinesRDD.count() + " bad lines")
println("Here are 10 examples:")
badLinesRDD.take(10).foreach(println)
```

Here we use take() to retrieve a small number of elements in the RDD at the driver program.

Iteration and print out is performed locally in the driver

Actions

- RDDs also have a **collect()** function to retrieve the entire RDD.
- This can be useful if your program filters RDDs down to a very small size and you'd like to deal with it locally.
- **Keep in mind** that your entire dataset **must fit in memory** on a single machine to use collect()

Passing Functions to Spark

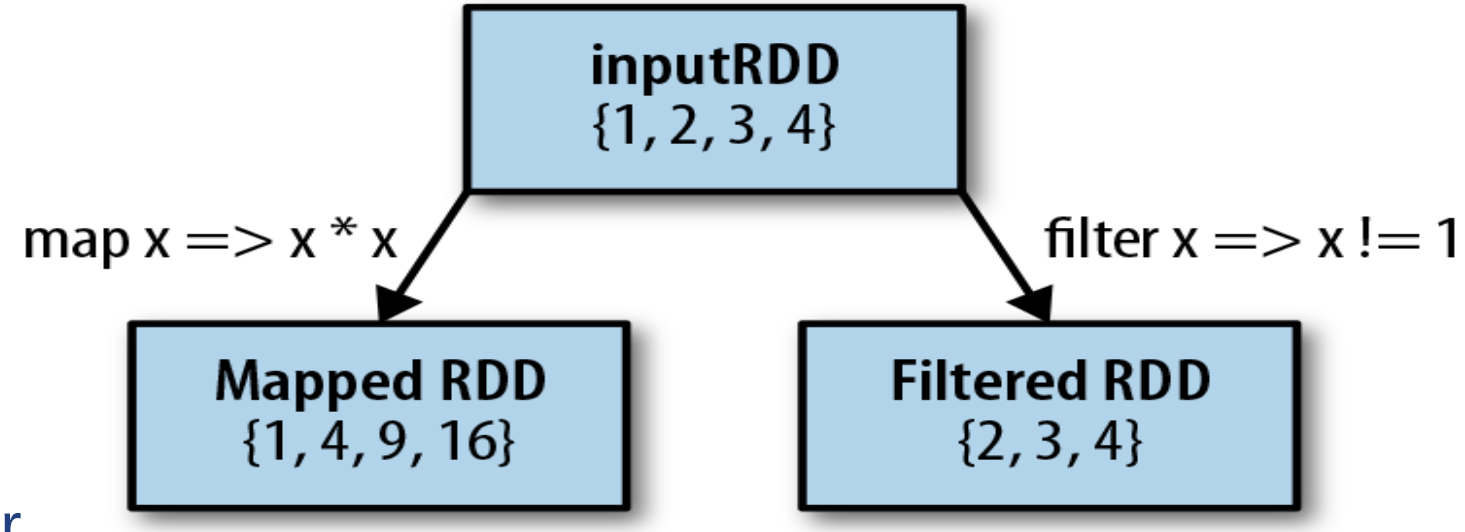
- Most of Spark's transformations, and some of its actions, depend on passing in functions that are used by Spark to compute data
- We can pass in functions defined inline, references to methods, or static functions as we do for Scala's other functional APIs
- The function we pass and the data referenced in it needs to be serializable (@serializable)

Passing Functions to Spark

- Passing a method or field of an object to a *e.g.* map of an RDD includes a reference to the whole object. Spark will then send the entire object to all worker nodes. To avoid this overhead, extract the needed fields as local variables

```
class SearchFunctions(val query: String) {
   def getMatchesFunctionReference(rdd: RDD[String]): RDD[Boolean] = {
      //Problem: "isMatch" is a class method, so "this.isMatch". All of "this" is passed
      rdd.map(isMatch)
   }
   def getMatchesFieldReference(rdd: RDD[String]): RDD[Array[String]] = {
      // Problem: "query" means "this.query", so we pass all of "this"
      rdd.map(x => x.split(query))
   }
   def getMatchesNoReference(rdd: RDD[String]): RDD[Array[String]] = {
      // Safe: extract just the field we need into a local variable
      val query_ = this.query
      rdd.map(x => x.split(query_))
   }
}
```

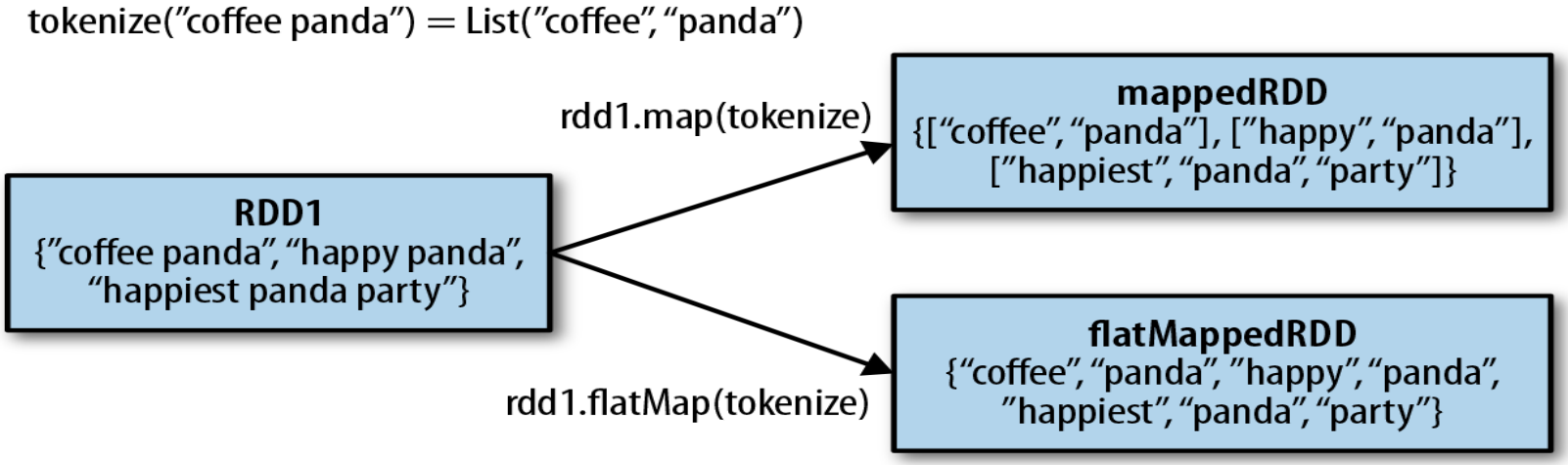
Common Transformations And Actions



- Map & Filter

```
val input = sc.parallelize(List(1, 2, 3, 4))
val result = input.map(x => x * x)
val resultF = input.filter(x => x != 1)
```

Common Transformations And Actions



- Flatmap

```
val lines = sc.parallelize(List("hello world", "hi"))
val words = lines.flatMap(line => line.split(" "))
words.first() // returns "hello"
```

Basic RDD transformations

RDD containing {1, 2, 3, 3}

Function name	Purpose	Example	Result
`map()`	Apply a function to each element in the RDD and return an RDD of the result.	`rdd.map(x => x + 1)`	`{2, 3, 4, 4}`
`flatMap()`	Apply a function to each element in the RDD and return an RDD of the contents of the iterators returned. Often used to extract words.	`rdd.flatMap(x => x.to(3))`	`{1, 2, 3, 2, 3, 3, 3}`
`filter()`	Return an RDD consisting of only elements that pass the condition passed to `filter()`.	`rdd.filter(x => x != 1)`	`{2, 3, 3}`
`distinct()`	Remove duplicates.	`rdd.distinct()`	`{1, 2, 3}`
`sample(withReplacement, fraction, [seed])`	Sample an RDD, with or without replacement.	`rdd.sample(false, 0.5)`	Nondeterministic

Two-RDD transformations

Two RDDs containing {1, 2, 3} and {3, 4, 5}

Function name	Purpose	Example	Result
union()	Produce an RDD containing elements from both RDDs.	rdd.union(other)	{1, 2, 3, 3, 4, 5}
intersection()	RDD containing only elements found in both RDDs.	rdd.intersection(other)	{3}
subtract()	Remove the contents of one RDD (e.g., remove training data).	rdd.subtract(other)	{1, 2}
cartesian()	Cartesian product with the other RDD.	rdd.cartesian(other)	{(1, 3), (1, 4), … (3,5)}

RDD Basic Actions

RDD contains {1, 2, 3, 3}

Function name	Purpose	Example	Result
`collect()`	Return all elements from the RDD.	`rdd.collect()`	`{1, 2, 3, 3}`
`count()`	Number of elements in the RDD.	`rdd.count()`	`4`
`countByValue()`	Number of times each element occurs in the RDD.	`rdd.countByValue()`	`{(1, 1), (2, 1), (3, 2)}`

RDD Other Actions

RDD

{1, 2, 3, 3}

Function name	Purpose	Example	Result
`take(num)`	Return num elements from the RDD.	`rdd.take(2)`	`{1, 2}`
`top(num)`	Return the top num elements the RDD.	`rdd.top(2)`	`{3, 3}`
`takeOrdered(num)(ordering)`	Return num elements based on provided ordering.	`rdd.takeOrdered(2)(myOrdering)`	`{3, 3}`
`takeSample(withReplacement, num, [seed])`	Return num elements at random.	`rdd.takeSample(false, 1)`	Nondeterministic
`reduce(func)`	Combine the elements of the RDD together in parallel (e.g., sum).	`rdd.reduce((x, y) => x + y)`	9

RDD Other Actions

RDD

{1, 2, 3, 3}

`fold(zero)(func)`	Same as `reduce()` but with the provided zero value.	`rdd.fold(0)((x, y) => x + y)`	9
`aggregate(zeroValue)(seqOp, combOp)`	Similar to `reduce()` but used to return a different type.	`rdd.aggregate((0, 0))((x, y) => (x._1 + y, x._2 + 1), (x, y) => (x._1 + y._1, x._2 + y._2))`	(9, 4)
`foreach(func)`	Apply the provided function to each element of the RDD.	`rdd.foreach(func)`	Nothing

Key/Value Pairs RDDs

- Spark provides special operations on RDDs containing **(key, value)** pairs

- They expose operations that allow you to **act on each key** in parallel, **regroup** or **aggregate** data across the network.
- Many formats loading return pair RDDs for key/value data.
- Parallelize (for testing and pocs)

```
val pairs = sc.parallelize
(List((1,”a”),(2,”b”),(3,”c”)))
```

- Running transformations over regular RDDs

```
//pair RDD using the first word as the key
val pairs = lines.map(x => (x.split(" ")(0), x))
```

Pair RDD Operations

Transformations on **one** pair RDD: {(1, 2), (3, 4), (3, 6)}

Function name	Purpose	Example	Result
`reduceByKey(func)`	Combine values with the same key.	`rdd.reduceByKey((x, y) => x + y)`	`{(1, 2), (3, 10)}`
`groupByKey()`	Group values with the same key.	`rdd.groupByKey()`	`{(1, [2]), (3, [4, 6])}`
`values()`	Return an RDD of just the values.	`rdd.values()`	`{2, 4, 6}`
`sortByKey()`	Return an RDD sorted by the key.	`rdd.sortByKey()`	`{(1, 2), (3, 4), (3,`

Pair RDD Operations

Transformations on **one** pair RDD: {(1, 2), (3, 4), (3, 6)}

`mapValues(func)`	Apply a function to each value of a pair RDD without changing the key.	`rdd.mapValues(x => x+1)`	`{(1, 3), (3, 5), (3, 7)}`
`flatMapValues(func)`	Apply a function that returns an iterator to each value of a pair RDD, and for each element returned, produce a key/value entry with the old key. Often used for tokenization.	`rdd.flatMapValues(x => (x to 5)`	`{(1, 2), (1, 3), (1, 4), (1, 5), (3, 4), (3, 5)}`
`keys()`	Return an RDD of just the keys.	`rdd.keys()`	`{1, 3, 3}`

Pair RDD Operations

Two RDDs of Paris **rdd** = {(1, 2), (3, 4), (3, 6)} **other** = {(3, 9)})

Function name	Purpose	Example	Result
`subtractByKey`	Remove elements with a key present in the other RDD.	`rdd.subtractByKey(other)`	`{(1, 2)}`
`join`	Perform an inner join between two RDDs.	`rdd.join(other)`	`{(3, (4, 9)), (3, (6, 9))}`
`rightOuterJoin`	Perform a join between two RDDs where the key must be present in the other RDD.	`rdd.rightOuterJoin(other)`	`{(3,(Some(4),9)), (3,(Some(6),9))}`
`leftOuterJoin`	Perform a join between two RDDs where the key must be present in the first RDD.	`rdd.leftOuterJoin(other)`	`{(1,(2,None)), (3, (4,Some(9))), (3, (6,Some(9)))}`
`cogroup`	Group data from both RDDs sharing the same key.	`rdd.cogroup(other)`	`{(1,([2],[])), (3, ([4, 6],[9]))}`

Pair RDD Operations

Actions on (key, value) RDDs {(1, 2), (3, 4), (3, 6)}

Function	Description	Example	Result
`countByKey()`	Count the number of elements for each key.	`rdd.countByKey()`	`{(1, 1), (3, 2)}`
`collectAsMap()`	Collect the result as a map to provide easy lookup.	`rdd.collectAsMap()`	`Map{(1, 2), (3, 4), (3, 6)}`
`lookup(key)`	Return all values associated with the provided key.	`rdd.lookup(3)`	`[4, 6]`

Exercises

- **Exercise 3** WordCount Spark
- **Exercise 4** Top-10 WordCount
- **Exercise 5** Aggregation by Key
- **Exercise 6** Combine RDDs

Persistence (caching)

- Remember: in Spark the RDD is computed again each time an **action** is called

```
val result = input.map(x => x*x)
// result.persist (StorageLevel.LEVEL)
println(result.count())
println(result.collect().mkString(","))
```

- To avoid this we can persist the data, **especially with iterative algorithms**.
- Each node stores their partitions of the RDD
- the default **persist()** will store the data in the JVM heap (memory) as **unserialized** objects

Persistence (caching)

- If a node that has data persisted on it fails, Spark will recompute the lost partitions of the data as needed.
- If you attempt to cache too much data to fit in memory, Spark will automatically evict old partitions using a Least Recently Used (**LRU**) cache policy.
- RDDs come with a method called **unpersist**() that lets you manually remove them from the cache

Persistence Levels

Level	Space used	CPU time	In memory	On disk	Comments
MEMORY_ONLY	High	Low	Y	N	
MEMORY_ONLY_SER	Low	High	Y	N	
MEMORY_AND_DISK	High	Medium	Some	Some	Spills to disk if there is too much data to fit in memory.
MEMORY_AND_DISK_SER	Low	High	Some	Some	Spills to disk if there is too much data to fit in memory. Stores serialized representation in memory.
DISK_ONLY	Low	High	N	Y	

- ***cache()*** = *persist(StorageLevel.MEMORY_ONLY)*

Persistence Levels

- Which storage level to choose?
  - Use **MEMORY_ONLY** if there is memory enough
  - If not **MEMORY_ONLY_SER**
  - Spill to disk only if
    - the functions that computed your RDDs are expensive
    - or they filter a large amount of the data
    - Otherwise, re computing a partition is faster than reading from disk

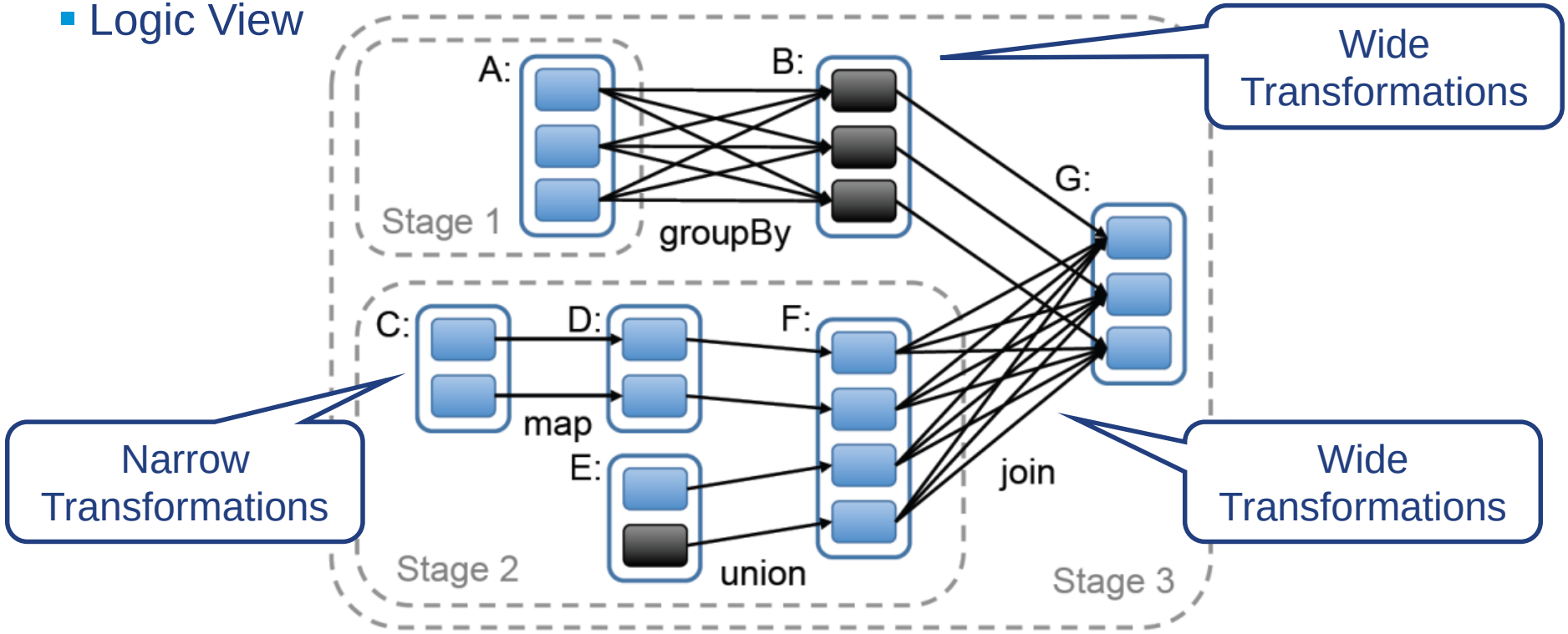
Level	Space used	CPU time
MEMORY_ONLY	High	Low
MEMORY_ONLY_SER	Low	High
MEMORY_AND_DISK	High	Medium
MEMORY_AND_DISK_SER	Low	High
DISK_ONLY	Low	High

Exercises

- **Exercise 7** Data Mining with MovieLens

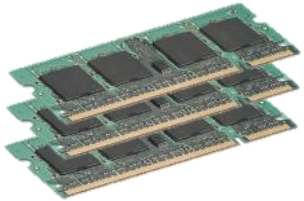
Spark Data Transformations

- Logic View

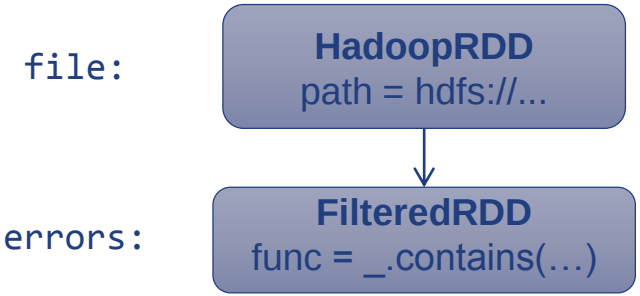


Spark Workflow – Partitions (1/2)

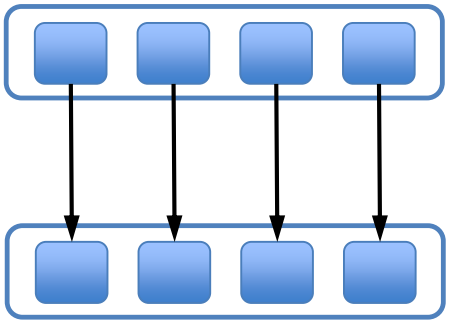
```
val sc = new SparkContext(“spark://...”, “MyJob”, home, jars)
val file = sc.textFile(“hdfs://...”)
val errors = file.filter(_.contains(“ERROR”))
errors.count()
```



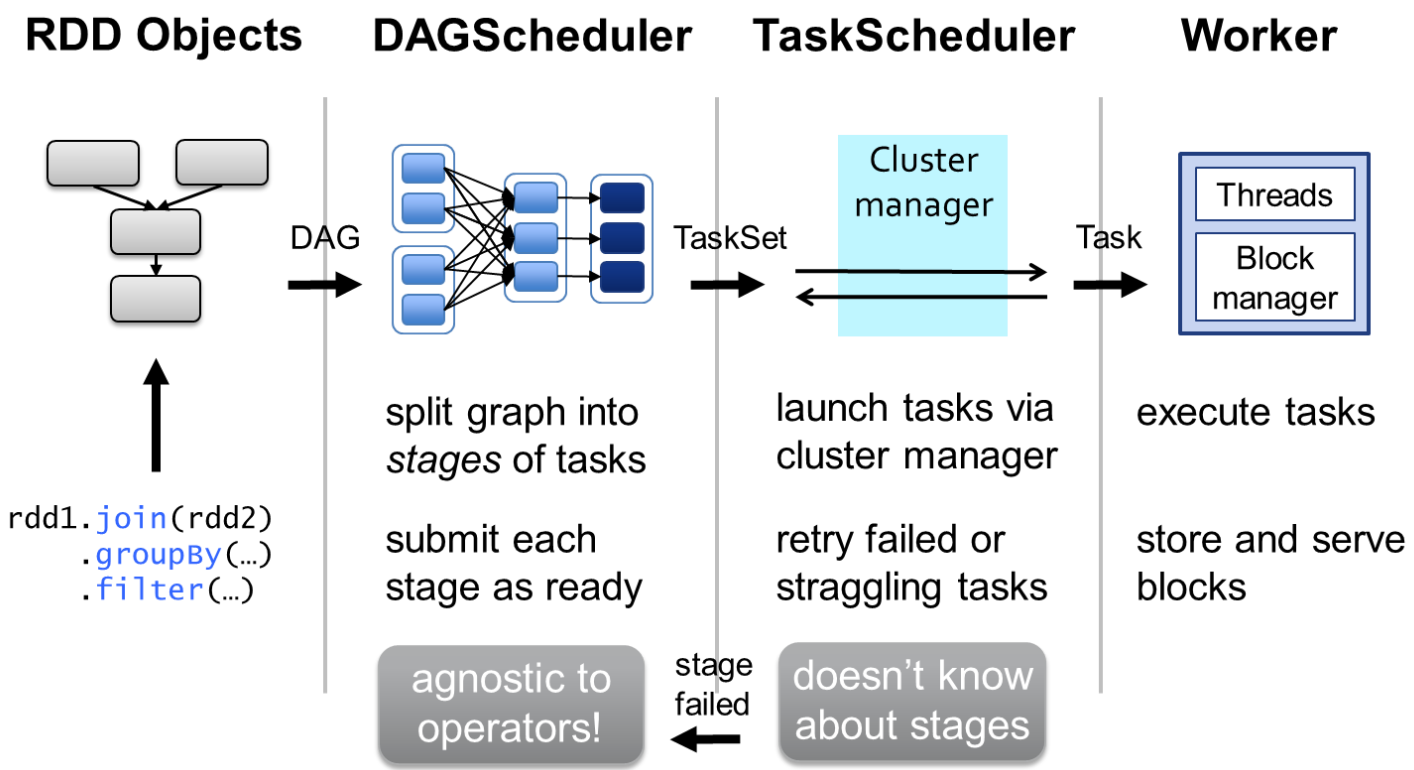
Dataset-level view



Partition-level view

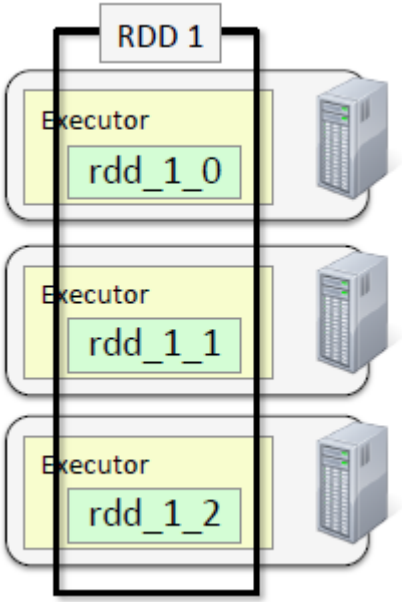


Spark Workflow – the rest (2/2)



Data Partitioning -- default

- Partitioning defines the degree of parallelism by “partitioning” the data between workers. Then in each worker the data will be processed in parallel
- By default is done automatically by **Spark**
  - spark.default.parallelism
  - In **local mode** the default is #cores
  - In **cluster mode** the default is #total number of cores

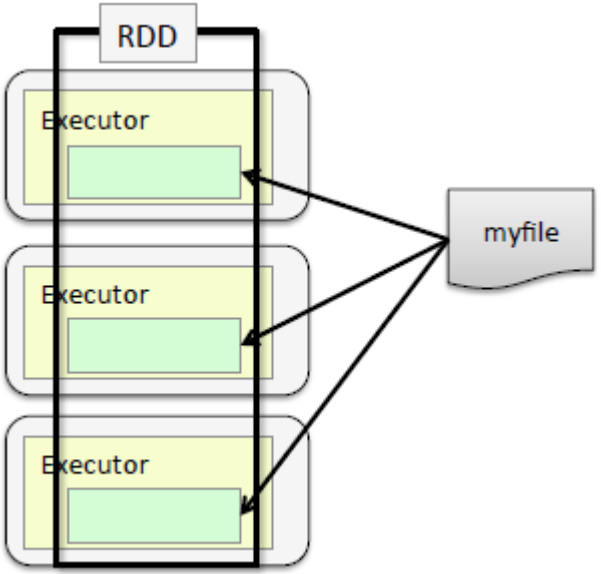


Data Partitioning – single small files

- Small – less than 1 HDFS block
- Partitions based in size
- Optionally minimum can be specified

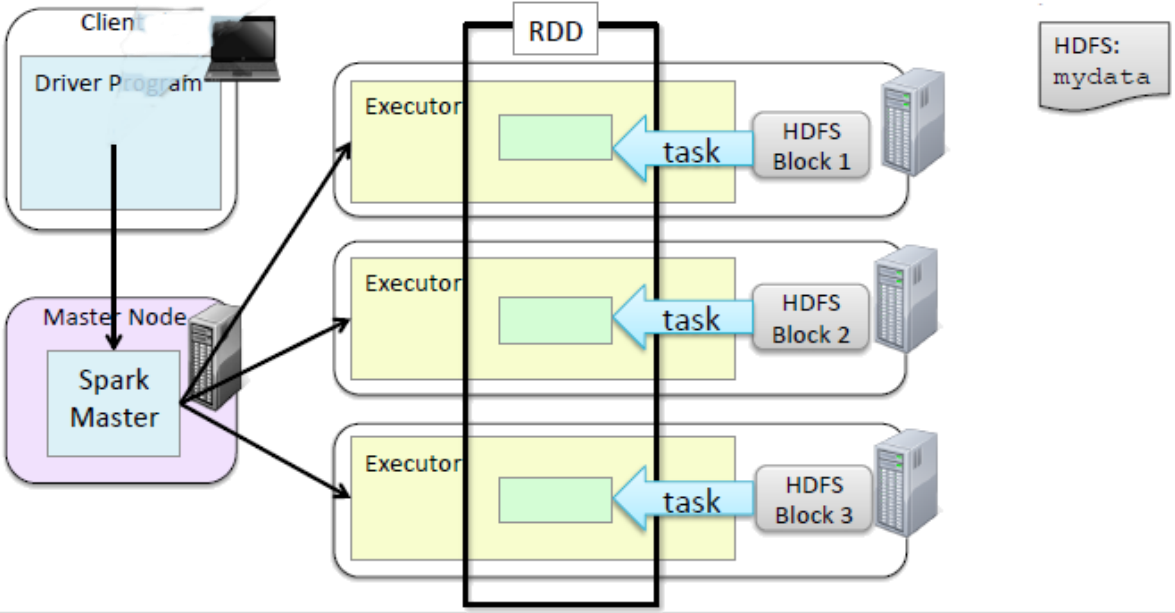
```
textFile(file, minPartitions)
```

- Default is 2
- More partitions → more parallelization



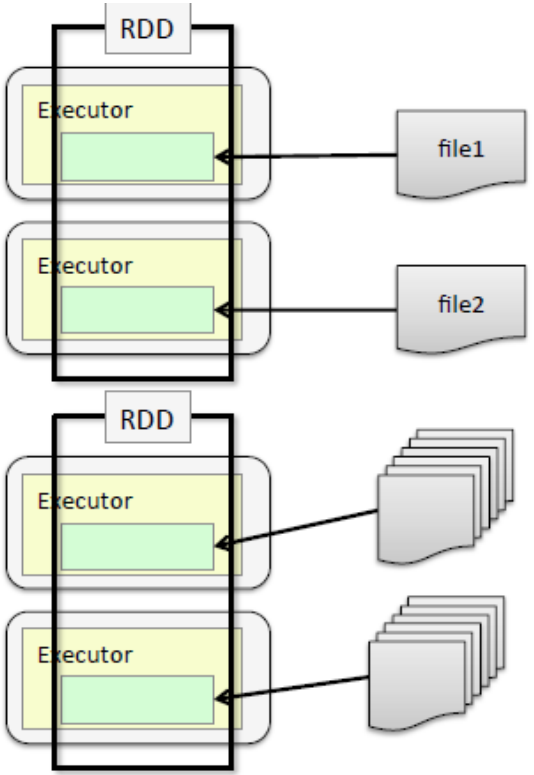
Data Partitioning – single big files

- big – more than 1 HDFS block
- Default – one block / one partition



Data Partitioning – multiple small files

- `textFile(file, minPartitions)`
- Each file becomes (at least) one partition
- File-based operations can be done per partition
- For many small files treat them as a single key-value RDD
  - key = file name
  - value = file contents
  - can be partitioned in a custom way – see later



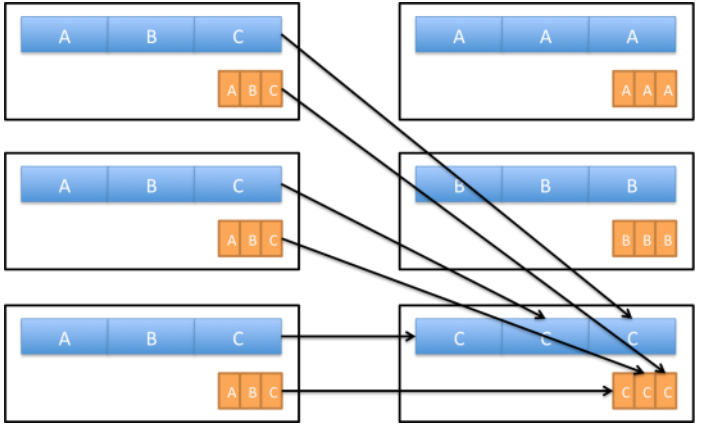
Data Partitioning – operations

- Most operations work on each element of an RDD
- Few work on each partition – avoid element setup overhead
- Functions per partitions take iterators

Function name	We are called with	We return	Function signature on RDD[T]
`mapPartitions()`	Iterator of the elements in that partition	Iterator of our return elements	`f: (Iterator[T]) → Iterator[U]`
`mapPartitionsWithIndex()`	Integer of partition number, and Iterator of the elements in that partition	Iterator of our return elements	`f: (Int, Iterator[T]) → Iterator[U]`
`foreachPartition()`	Iterator of the elements	Nothing	`f: (Iterator[T]) → Unit`

Data Partitioning for Key/Value RDDs

- Partitioning is available for all RDDs of key/value pairs
- It causes the system to **group elements** based on a function of **each key**.
- Spark does not give explicit control of which worker node each key goes to but it lets the program **ensure that a *set* of keys** will appear together on the same node ***somewhere***.
- For example, you might choose to hash-partition an RDD into 100 partitions so that keys that have the same hash value modulo 100 appear on the same node



Data Partitioning

- Example (works but it is not efficient)

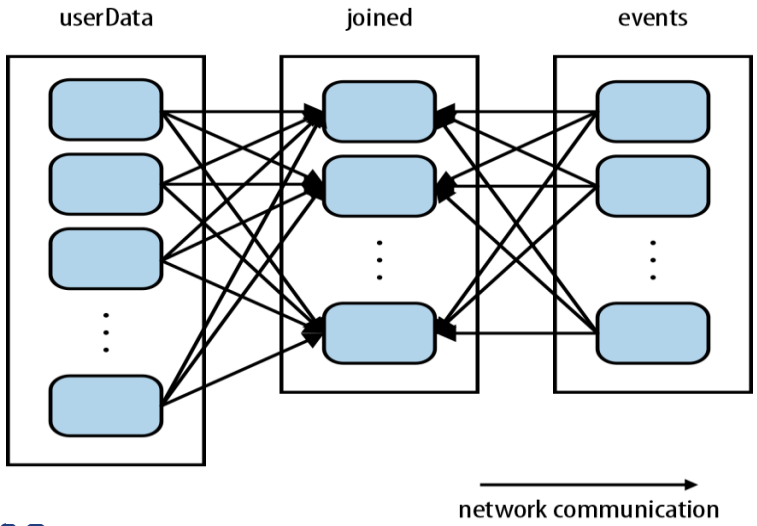
```
// Initialization code; we load the user info from a Hadoop SequenceFile on HDFS.
// This distributes elements of userData by the HDFS block where they are found,
// and doesn't provide Spark with any way of knowing in which partition a
// particular UserID is located.

val sc = new SparkContext(...)
val userData = sc.sequenceFile[UserID, UserInfo]("hdfs://...").persist()

// Function called periodically to process a logfile of events in the past 5 minutes;
// we assume that this is a SequenceFile containing (UserID, LinkInfo) pairs.
def processNewLogs(logFileName: String) {
    val events = sc.sequenceFile[UserID, LinkInfo](logFileName)
    val joined = userData.join(events)// RDD of (UserID, (UserInfo, LinkInfo)) pairs
    val offTopicVisits = joined.filter {
        case (userId, (userInfo, linkInfo)) => // Expand the tuple into its
        components
        …
    }.count()
    println("Number of visits to non-subscribed topics: " + offTopicVisits)
}
```

Data Partitioning

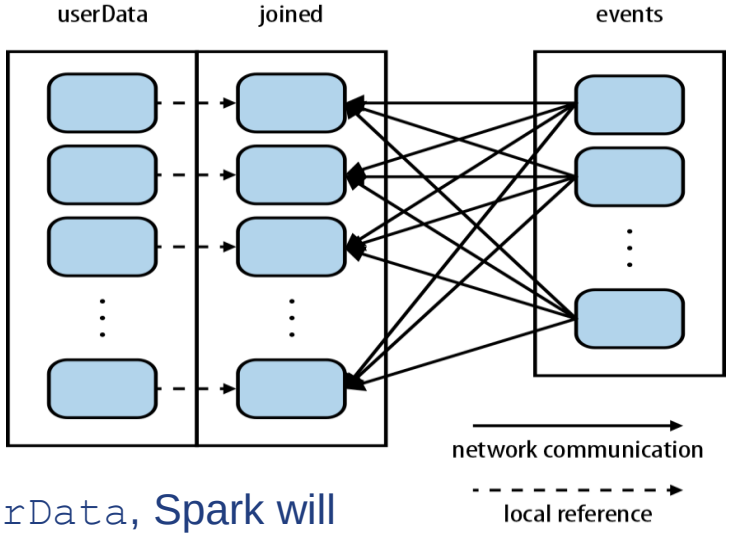
- Why is not efficient?
- The join() operation is called each time **processNewLogs**() is invoked.
- It does not know anything about how the keys are partitioned in the datasets, so the **userData** table is hashed and shuffled across the network on every call, even though it doesn't change.
- Because we expect the **userData** table to be much larger than the small **log of events** seen every five minutes, this wastes a lot of work:



Data Partitioning - Solution

- Use `org.apache.spark.HashPartitioner` with the **transformation** `partitionBy()`:

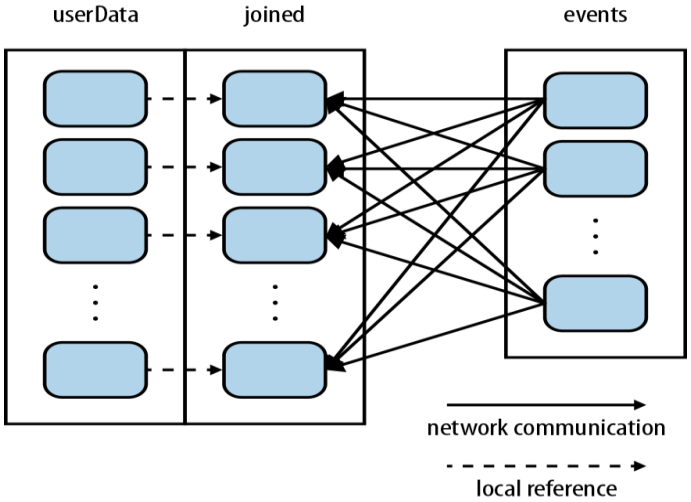
```
val sc = new SparkContext(...)
val userData = sc.sequenceFile[UserID,
    UserInfo]("hdfs://...")
   .partitionBy(new HashPartitioner(100))
   //Create 100 partitions
   .persist()
```



- Because we called `partitionBy()` when building `userData`, Spark will now know that it is hash-partitioned, and calls to join() on it will take advantage of this information. When we call `user Data.join(events)`, Spark will shuffle only the events RDD, sending events with each particular `UserID` to the machine that contains the corresponding hash partition of `userData`

Data Partitioning -- Solution

- It is important to **persist** `userData` otherwise subsequent uses of the RDD will result in **complete lineage revaluation**, repeated partitioning and **shuffling** data across the network
- The **number of partitions** control how many parallel **tasks** will be performed on the RDD (e.g. joins)
  - at least as large of the number of cores in the cluster
  - or even slightly more
- It's possible use **others** (RangePartitioner) or implement our own partitioner from org.apache.spark.Partitioner



Define the RDD Data Partitioner

```
scala> val pairs = sc.parallelize(List((1, 1), (2, 2), (3, 3)))
pairs: spark.RDD[(Int, Int)] = ParallelCollectionRDD[0] at parallelize at <console>:12
scala> pairs.partitioner
res0: Option[spark.Partitioner] = None
scala> import org.apache.spark.HashPartitioner
import org.apache.spark.HashPartitioner
scala> val partitioned = pairs.partitionBy(new HashPartitioner(2)).persist()
partitioned: spark.RDD[(Int, Int)] = ShuffledRDD[1] at partitionBy at <console>:14
scala> partitioned.partitioner
res1: Option[spark.Partitioner] = Some(spark.HashPartitioner@5147788d)
```

When RDD Data Partitioning

- **Operations That Benefit from Partitioning**
  - **One RDD**: All the values for each key to be computed locally on a single machine, requiring only the final, locally reduced value to be sent from each worker node back to the master
    - `groupByKey(), reduceByKey(), lookup()`
  - **Two RDD**: -At least one of the RDDs will not be shuffled. -If both have the same partitioner and cached on the same server (eg. One was created from other using mapValues) then no shuffling will occur.
    - `cogroup(), join(), leftOuterJoin(), rightOuterJoin()`

- **Operations That Affect Partitioning**
  - if you call `map()` on a hash-partitioned RDD of key/value pairs, the function passed to `map()` can in theory change the key of each element, so the result will not have a partitioner
  - Therefore, operations that result in a partitioner being set on the output RDD:
    - `cogroup(), join(), leftOuterJoin(), rightOuterJoin(), groupByKey(), reduceByKey(), partitionBy(), sort()`
  - If the parent RDD has a partitioner: `mapValues(),flatMapValues(),filter()`

Numeric RDD Operations

- Spark provides several descriptive statistics operations on RDDs containing numeric data
- The descriptive statistics are all computed in a single pass over the data and returned as a `StatsCounter` object by calling `stats()`.

StatsCounter

Method	Meaning
`count()`	Number of elements in the RDD
`mean()`	Average of the elements
`sum()`	Total
`max()`	Maximum value
`min()`	Minimum value
`variance()`	Variance of the elements
`sampleVariance()`	Variance of the elements, computed for a sample
`stdev()`	Standard deviation
`sampleStdev()`	Sample standard deviation

```
// remove outliers since those may have misreported
locations
// first we need to take our RDD of strings and turn
it into doubles.

val distanceDouble = distance.map(string =>
                              string.toDouble)
val stats = distanceDoubles.stats()
val stddev = stats.stdev
val mean = stats.mean
val reasonableDistances = distanceDoubles.filter(x
                => math.abs(x-mean) < 3 * stddev)
println(reasonableDistance.collect().toList)
```

Exercises

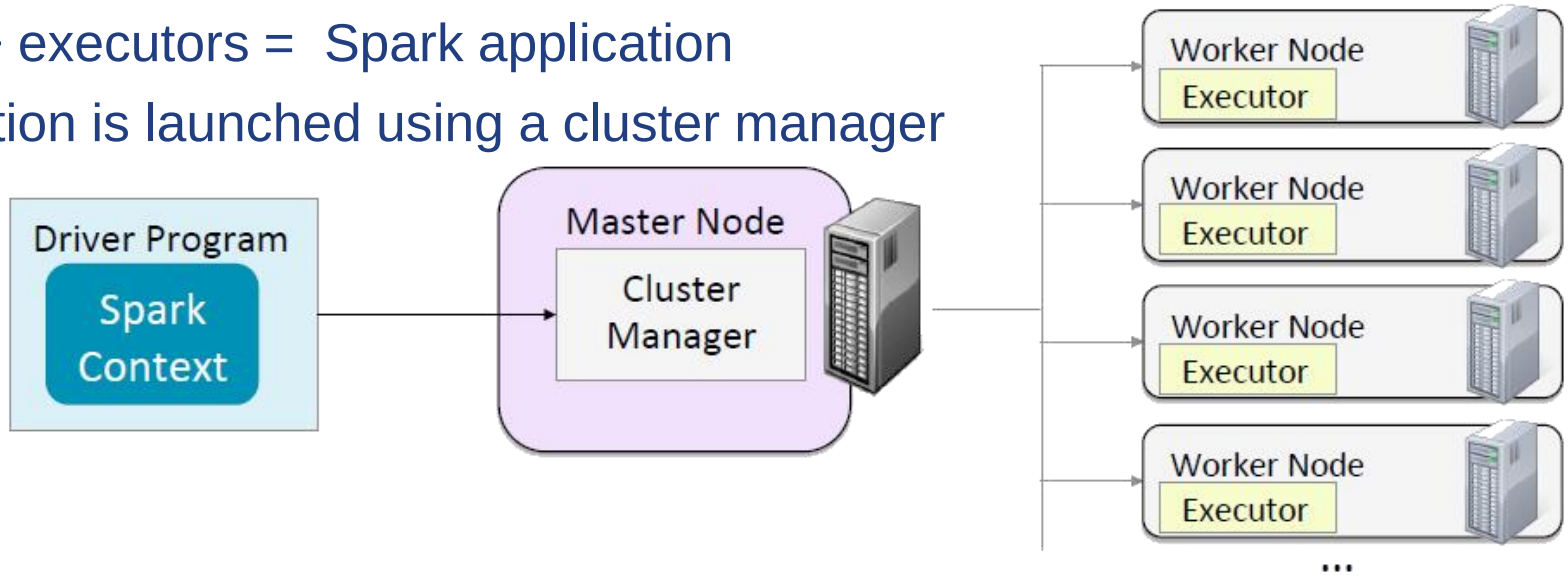
- **Exercise 8** Transformations per partition

INTRODUCTION TO SPARK

4. Spark Architecture

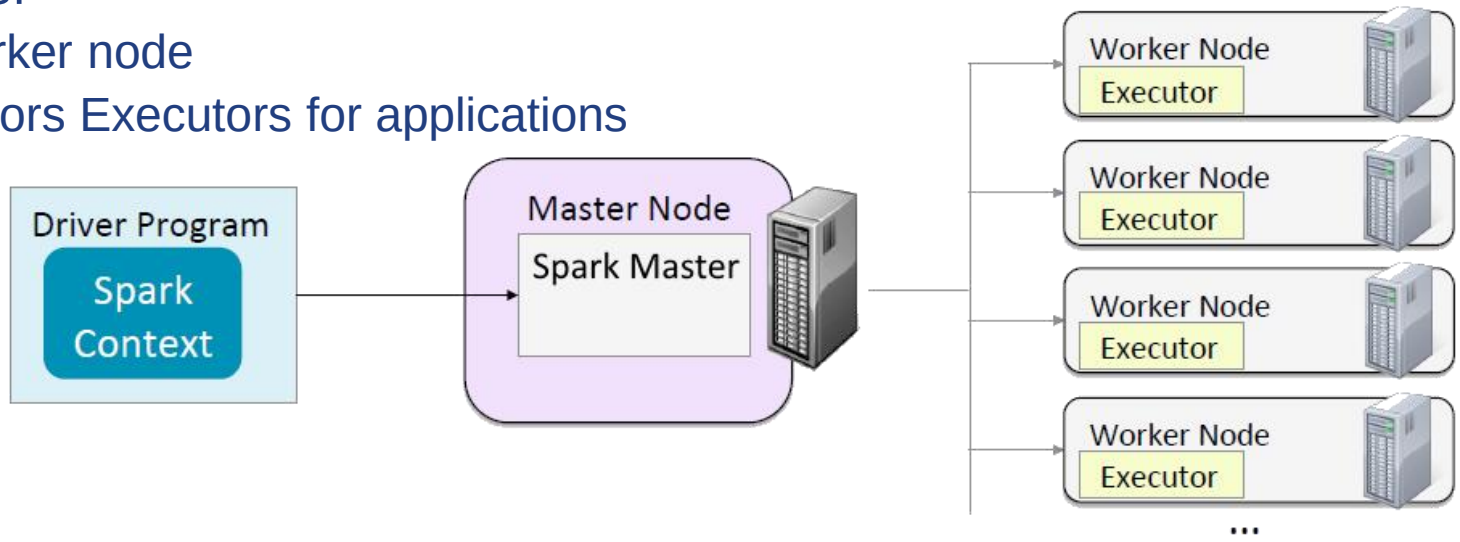
Spark Runtime architecture

- Master/slave architecture
- Central coordinator driver (own java process)
- Daemons on workers called executor (own java process)
- Driver + executors = Spark application
- Application is launched using a cluster manager



Spark Standalone Cluster -- Daemons

- Spark Master
  - One per cluster
  - Manages applications, distributes tasks to Spark Workers
- Spark Worker
  - One per worker node
  - Starts monitors Executors for applications

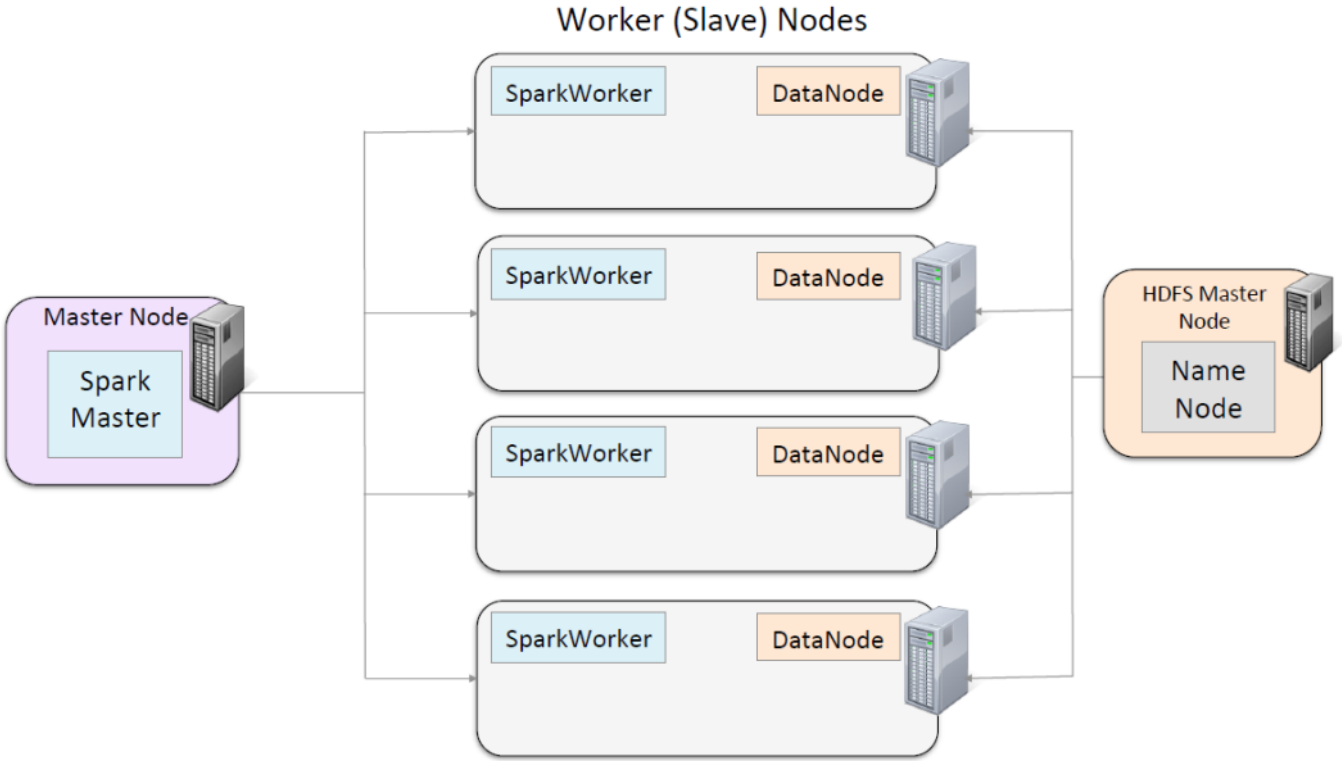


Launching a program

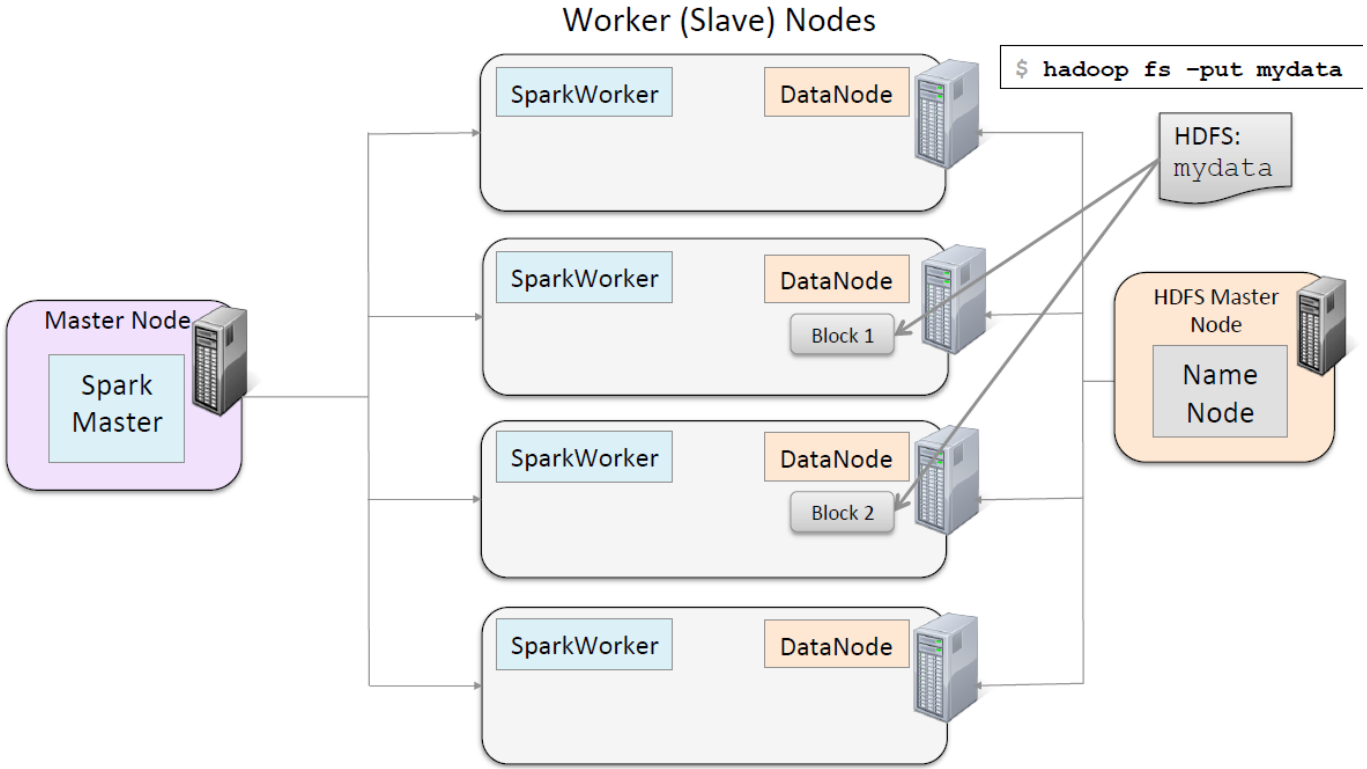
- Single script **spark-submit** with options to connect to clusters and control resources

1. The user submits an application using **spark-submit**.
2. **spark-submit** launches the driver program and invokes the main() method specified by the user.
3. The **driver program** contacts the **cluster manager** to ask for resources to launch executors.
4. The **cluster manager** launches executors on behalf of the driver program.
5. The **driver process** runs through the user application. Based on the RDD actions and transformations in the program, the driver sends work to executors in the form of tasks.
6. Tasks are run on executor processes to compute and save results.
7. If the driver's main() method exits or it calls **SparkContext.stop()**, it will terminate the executors and release resources from the cluster manager.

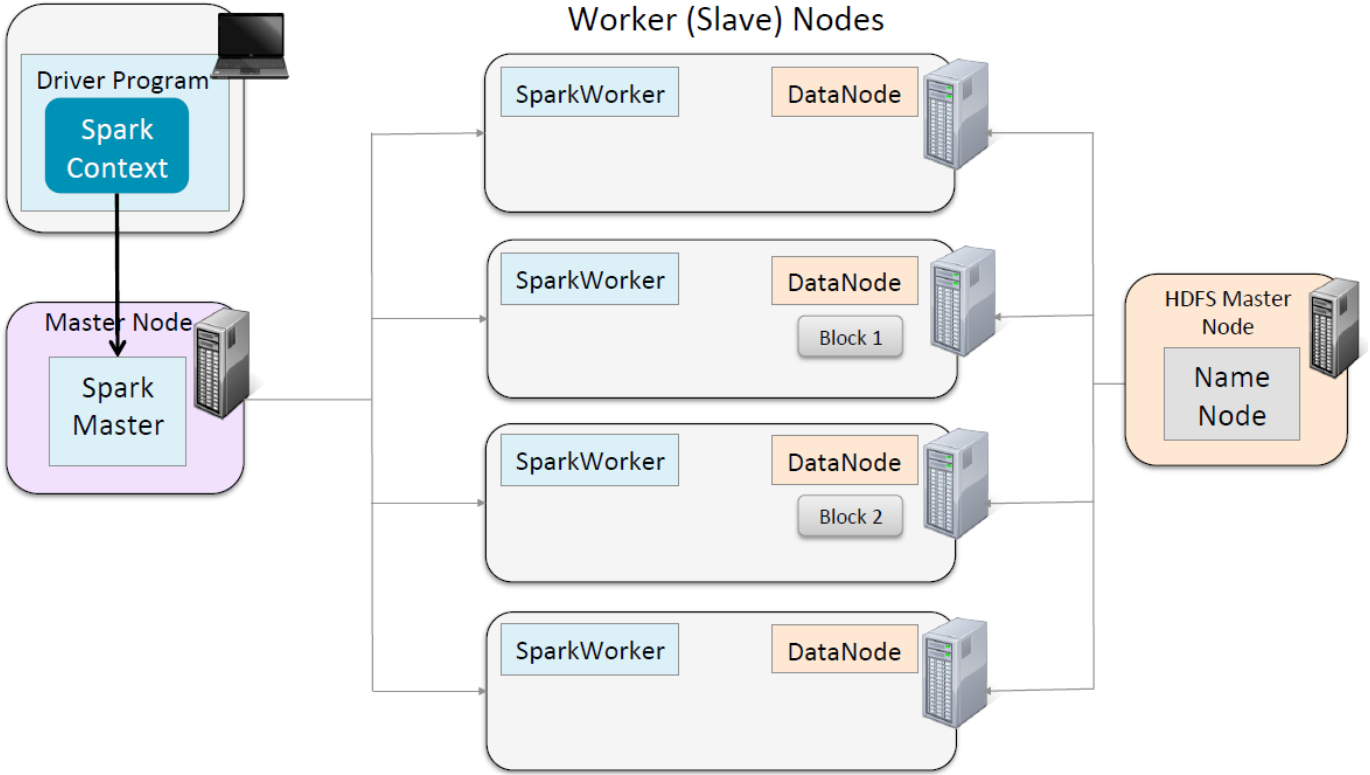
Spark Standalone Cluster Workflow (1/5)



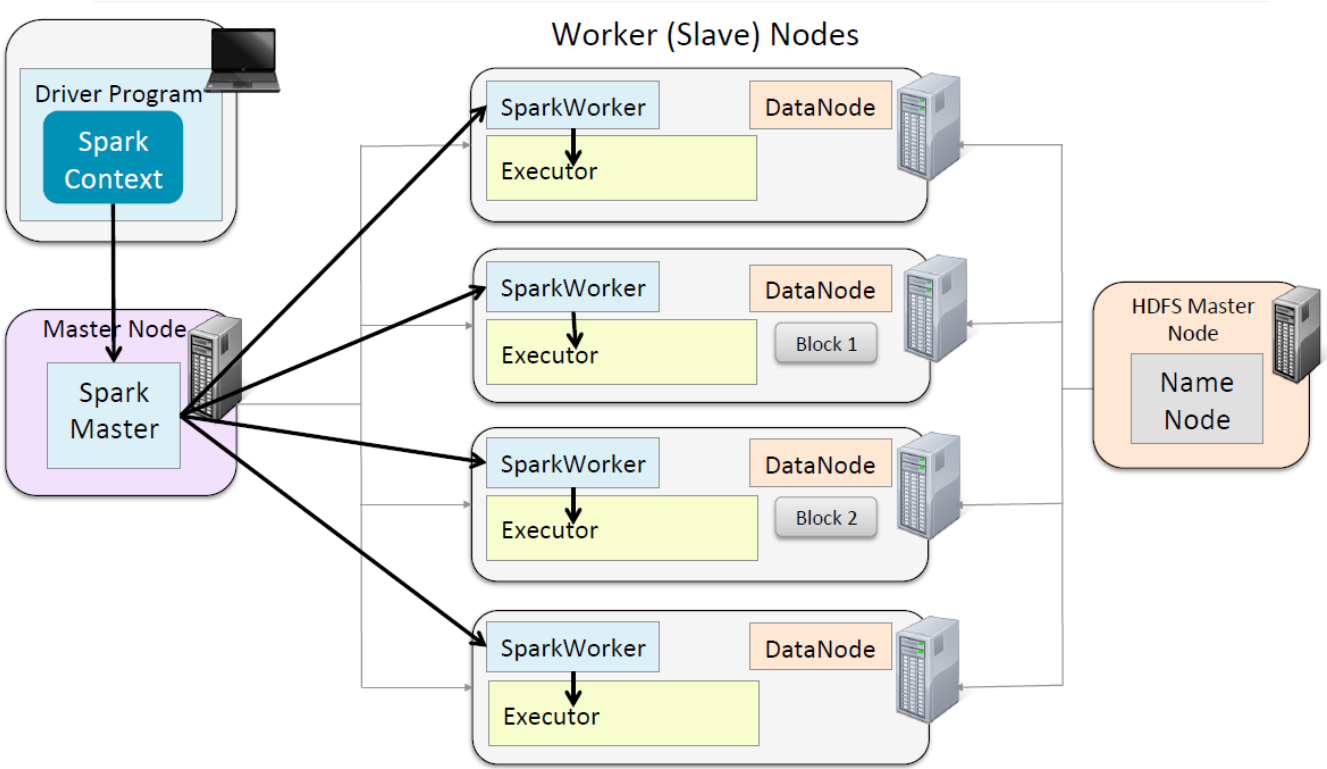
Spark Standalone Cluster Workflow (2/5)



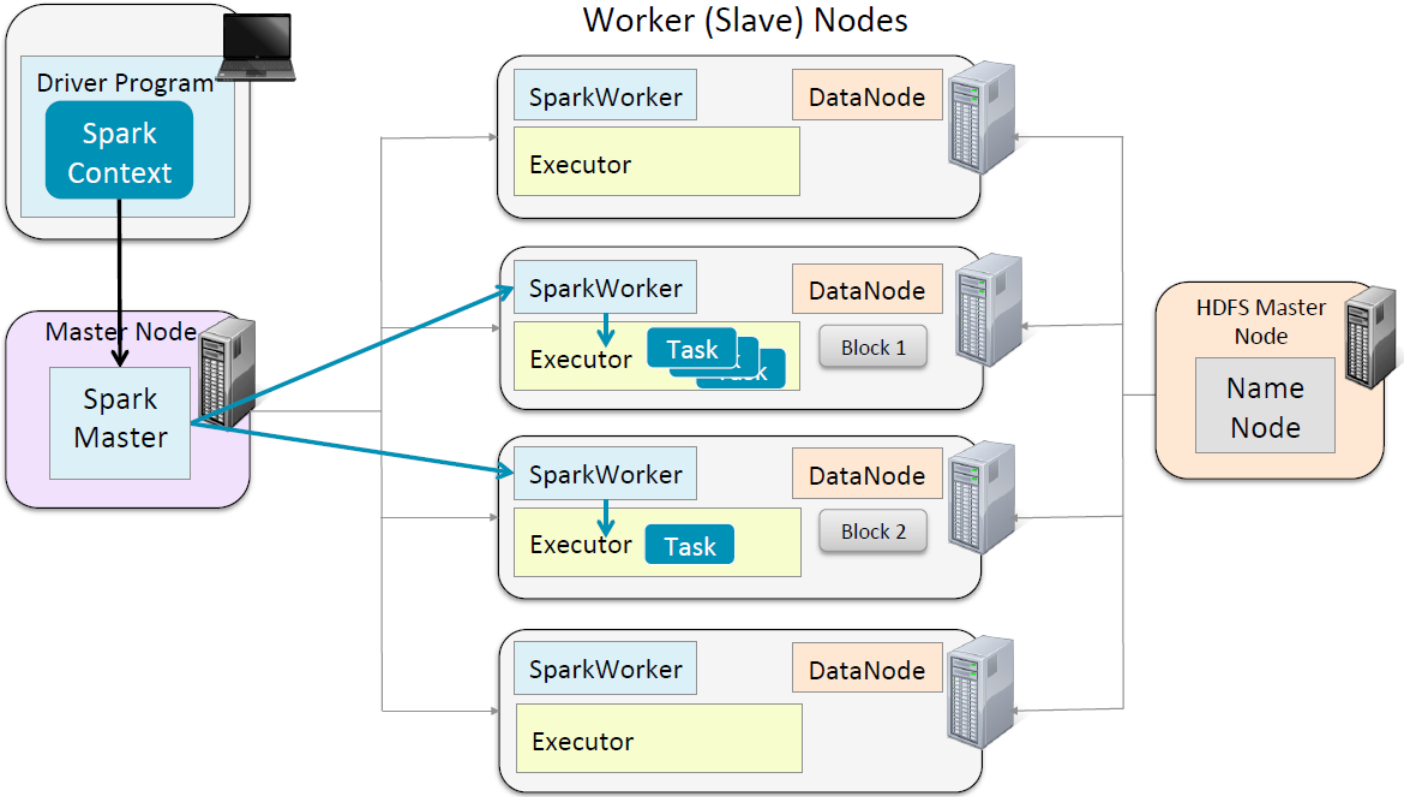
Spark Standalone Cluster Workflow (3/5)



Spark Standalone Cluster Workflow (4/5)

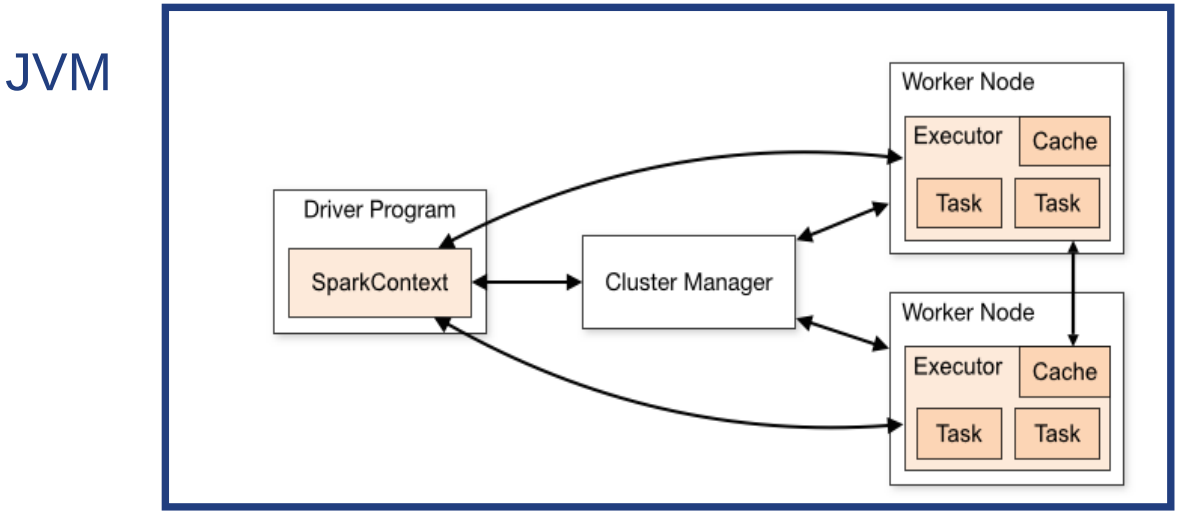


Spark Standalone Cluster Workflow (5/5)



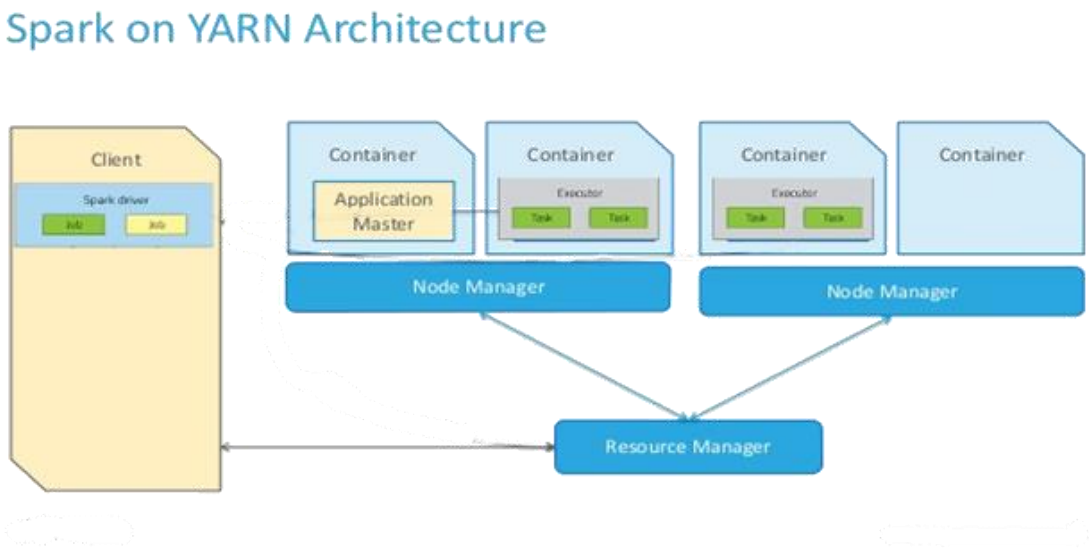
Spark on local mode

- The **Spark Context**, the **Spark Master** (cluster manager) and the **Worker node** are **threads** in a single JVM
- The number of workers (threads) are set at spark-shell invocation
  - `spark-shell –master local[#threads]`



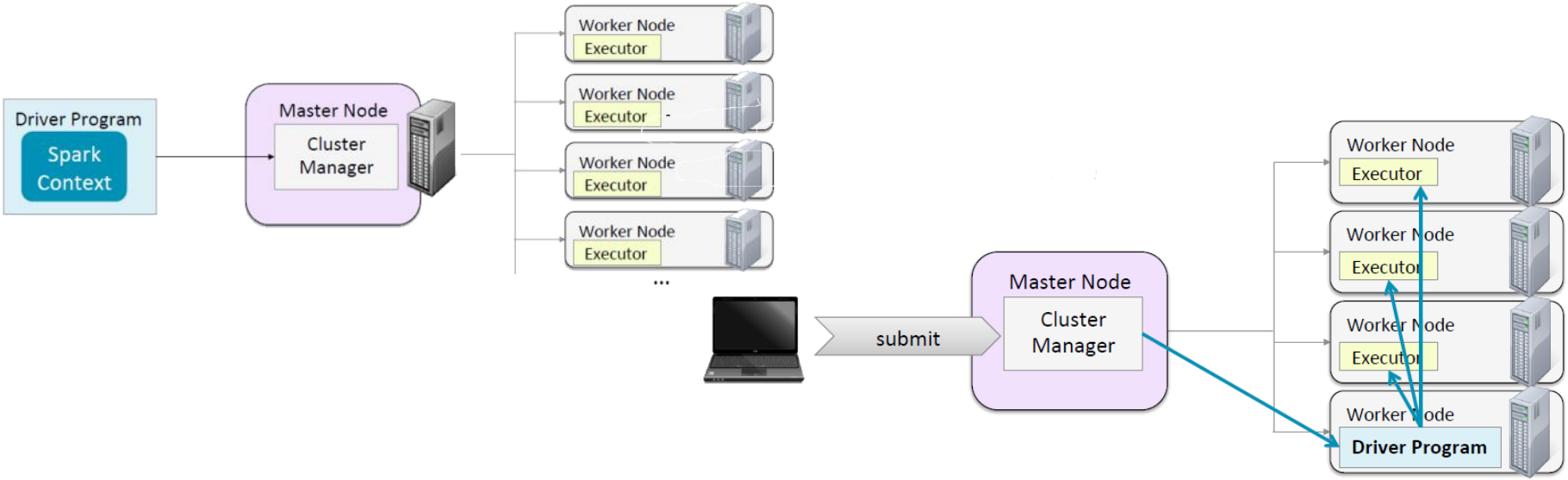
Spark with other cluster managers

- Spark can run the Executors and the Master with other cluster managers (YARN, Mesos)
- Executors and the Master are run as tasks managed by the cluster manager
- Cluster manager provides scalability, fault tolerance, multi-jobs



Spark modes with other cluster managers

- Two modes exist, depending where the driver is run
- **Client mode**: the driver is outside the cluster (spark-shell)
- **Cluster mode**: the driver is another task in the cluster (no spark-shell)

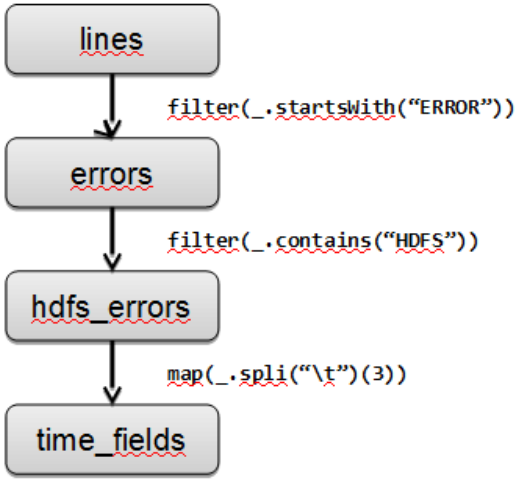


The Driver

- Is the process where the **main()** method of your program runs
- It is the process running the **user code** that creates **a SparkContext**, **creates RDDs**, and indicates transformations and actions

```
val lines = sc.textFile(…)
val errors = lines.filter(_.startsWith("ERROR"))
val hdfs_errors = errors.filter(_.contains("HDFS"))
val time_fields = hdfs_errors.map(_.split("\t")(3))
```

- Directed acyclic graph – DAG
  - Vertices – RDDs
  - Edges – transformations

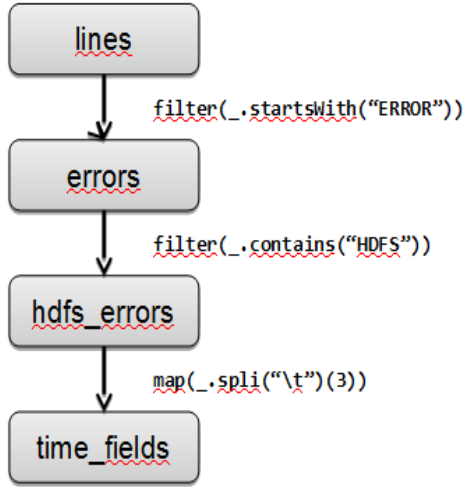


The Driver – duties

- **Converts a user program into tasks**:
  - Creates a logical **directed acyclic graph (DAG)** of operations.
  - Converts DAG into a **physical execution plan**
  - Creates a series of **stages** based on DAG optimizing transformations and merging them
  - Each stage consists of multiple **tasks** (smallest unit of work). They are bundled up and sent to the cluster
- **Schedules tasks on executors**
  - Uses **physical execution** plan
  - At start executors register themselves to the driver. Each executor can run task and store RDD slices
  - Based on data placement the driver schedule each task to a worker
  - Driver tracks the location of cached data

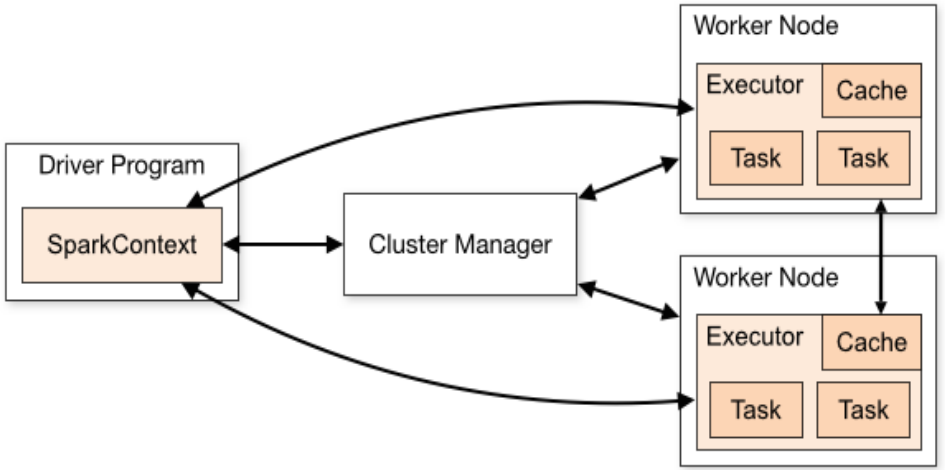
```
val lines = sc.textFile(…)
val errors = lines.filter(_.startsWith("ERROR"))
val hdfs_errors = errors.filter(_.contains("HDFS"))
val time_fields = hdfs_errors.map(_.split("\t")(3))
```

- Directed acyclic graph – DAG
  - Vertices – RDDs
  - Edges – transformations



Executors

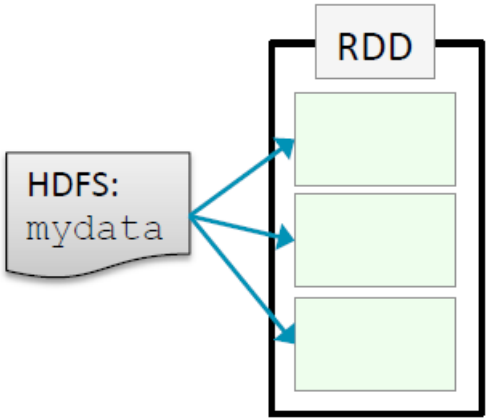
- Java processes responsible for running individual tasks in a Spark job
- Two roles
  - Run the tasks and return results to the driver
  - Provide in-memory storage for RDDs through a service called **Block Manager**



Execution Components – Example (1/5)

Average work length by letter

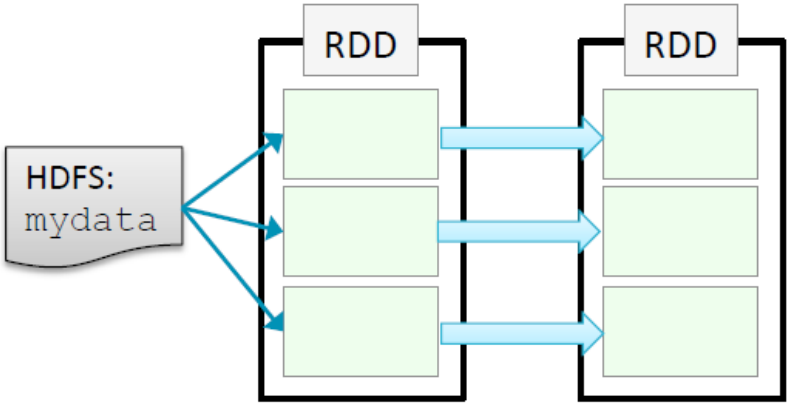
```
$ var avg_length = sc.textFile(fileName)
```



Execution Components – Example (2/5)

Average work length by letter

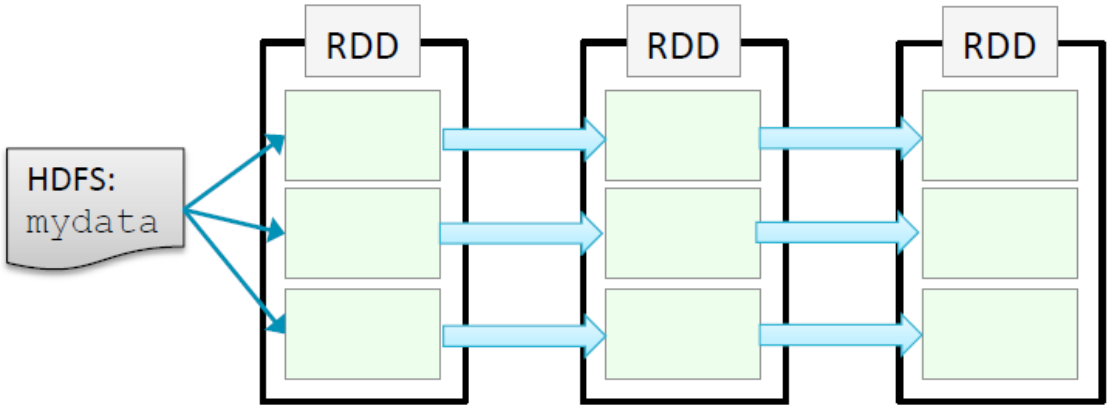
```
$ var avg_length = sc.textFile(fileName).flatMap(line => line.split(" ")
```



Execution Components – Example (3/5)

Average work length by letter

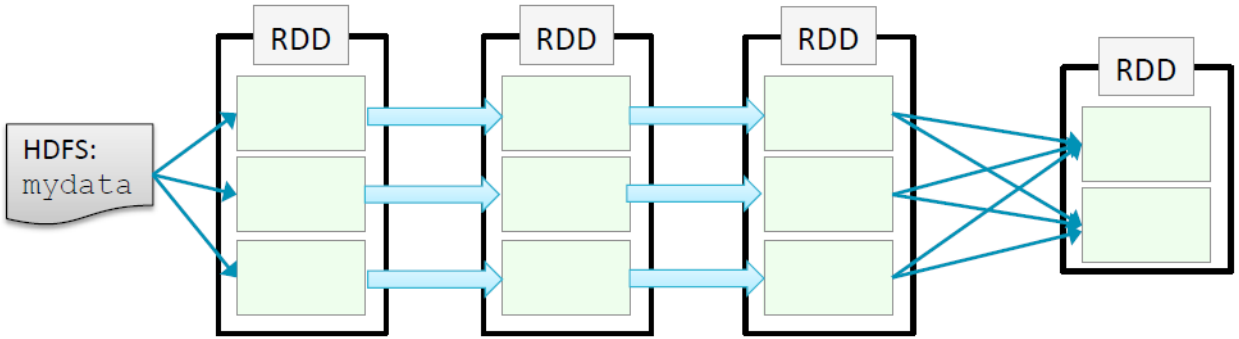
```
$ var avg_length = sc.textFile(fileName).flatMap(line => line.split(" ")
$ | .map( word => (word[0], word.length)
```



Execution Components – Example (4/5)

Average work length by letter

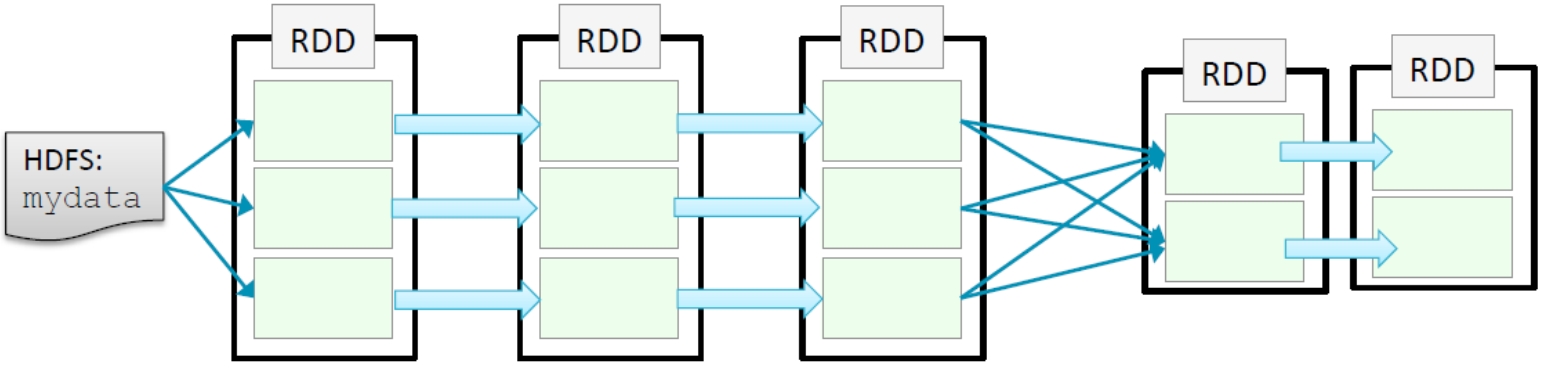
```
$ var avg_length = sc.textFile(fileName).flatMap(line => line.split(" ")
$ | .map( word => (word[0], word.length)
$ | .groupByKey()
```



Execution Components – Example (5/5)

Average work length by letter

```
$ var avg_length = sc.textFile(fileName).flatMap(line => line.split(" ")
$ | .map( word => (word[0], word.length)
$ | .groupByKey()
$ | .map((key,values) => sum(values)/values.length)
```

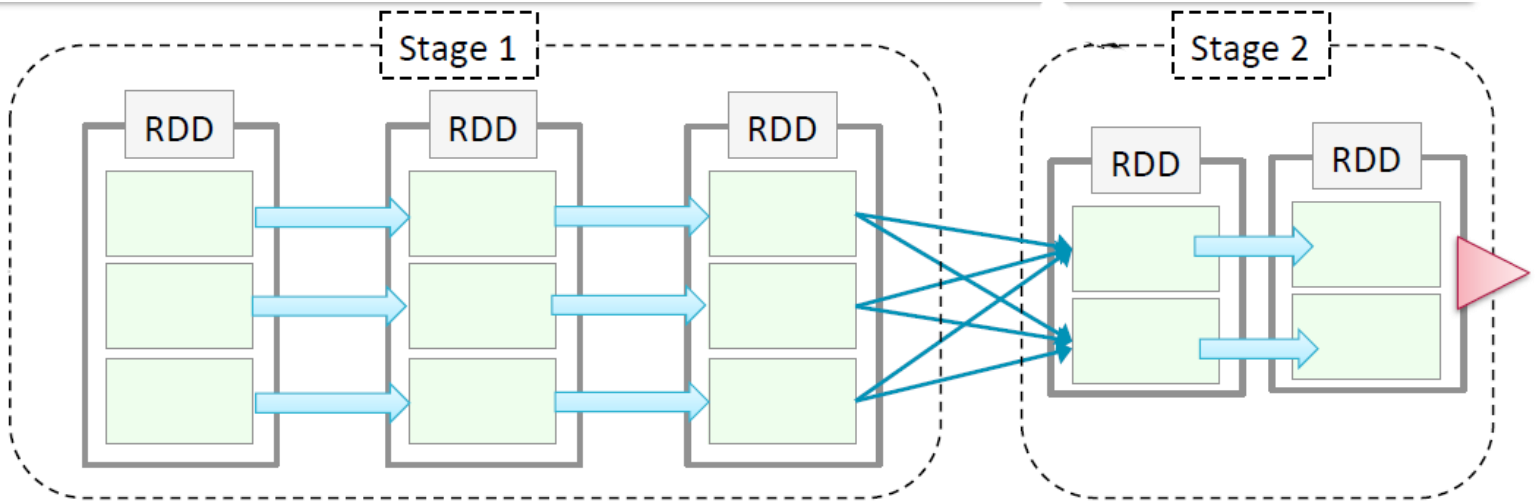


Components of Execution (1/4)

- At this time there are no actions. Only the DAG has been created.
- To display the lineage of the RDDs we can use **toDebugString**
- **Stages** and **Tasks** are identified
- **Stages:** group of operations that can be run on the same partition
- **Tasks:** parallel transformations inside a **stage**
- To **improve performance** should be aware of the **stages**

Components of Execution (2/4) – Stages

```
$ var avg_length = sc.textFile(fileName).flatMap(line => line.split(" ")
$ |  .map( word => (word[0], word.length)
$ |  .groupByKey()
$ |  .map((key,values) => sum(values)/values.length)
$ avg_length.count()
```

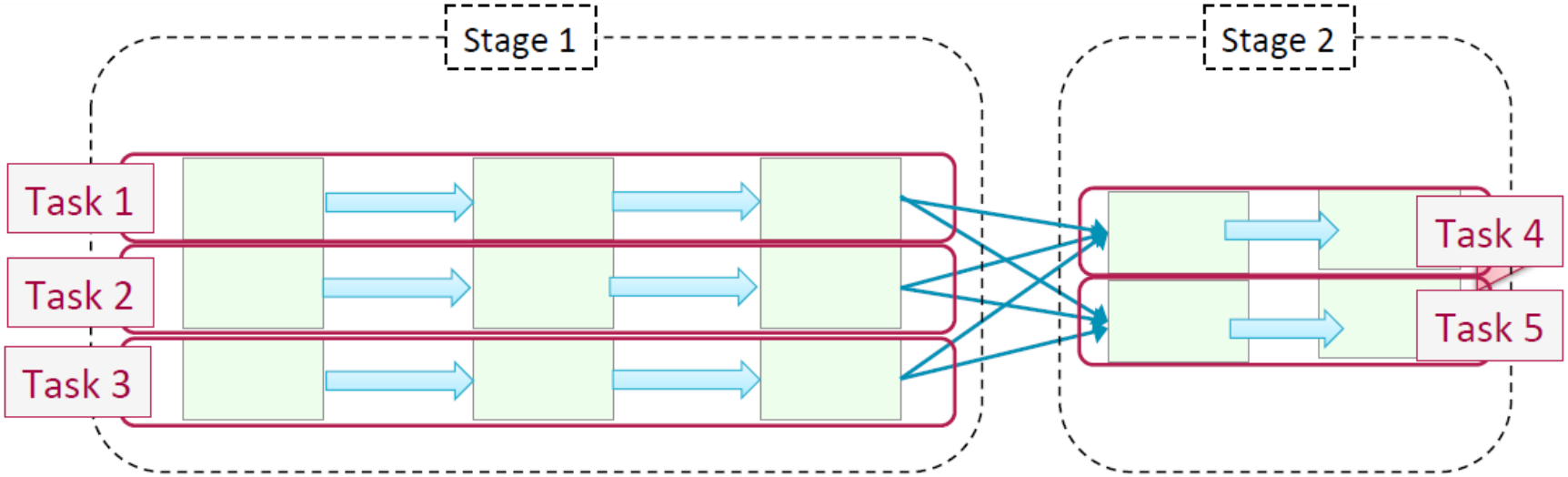


Components of Execution (3/4) – Scheduler

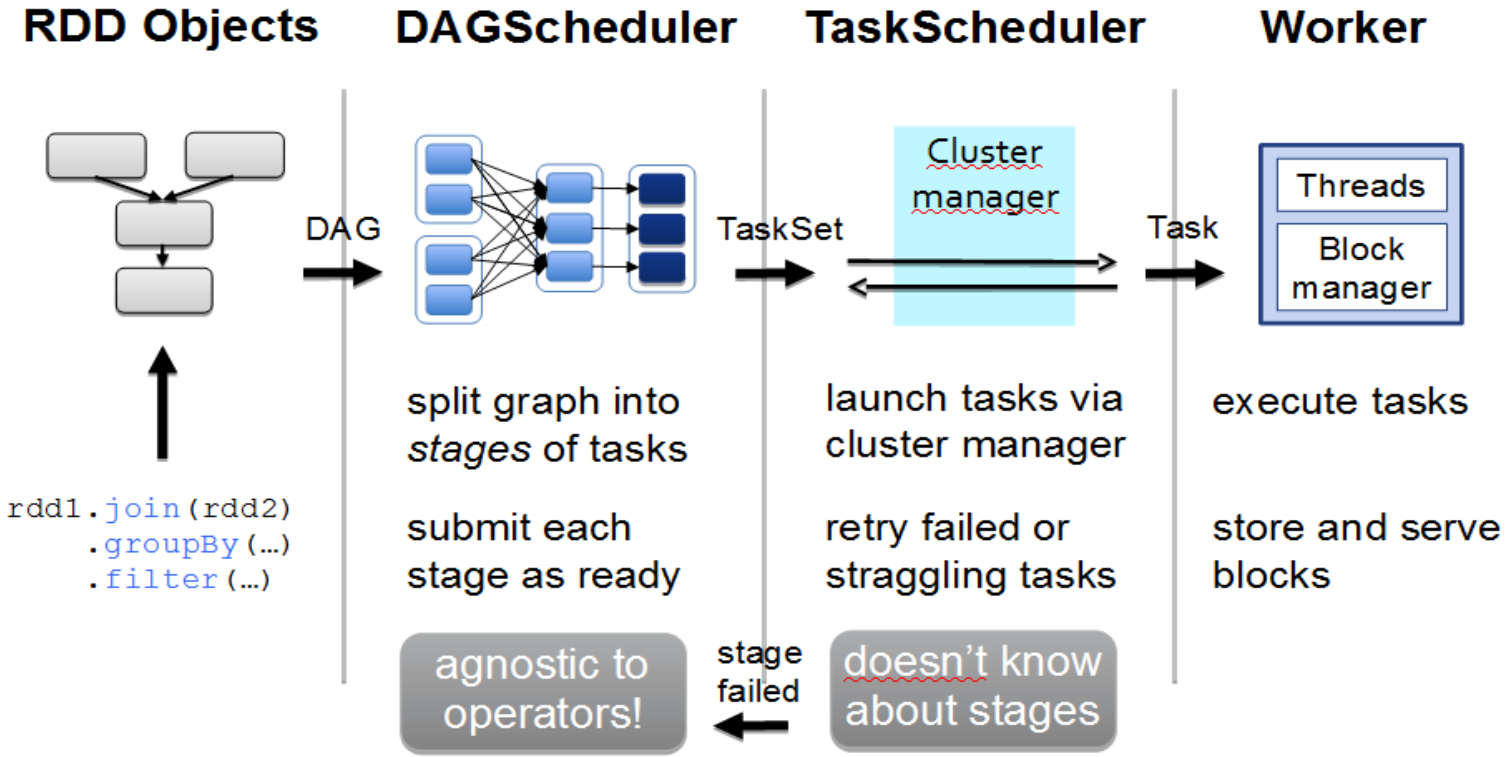
- The **Scheduler** performs *pipelining*, or the collapsing of computation of multiple RDDs into a single stage.
- Pipelining occurs when RDDs can be computed from their parents **without data movement**
- Each stage has *tasks* for each partition in that RDD
- Spark's internal scheduler may **truncate** the lineage of the **RDD graph** if an existing RDD has already been persisted

Components of Execution (4/4) – Tasks

```
$ var avg_length = sc.textFile(fileName).flatMap(line => line.split(" ")
$ |  .map( word => (word[0], word.length)
$ |  .groupByKey()
$ |  .map((key,values) => sum(values)/values.length)
$ avg_length.count()
```

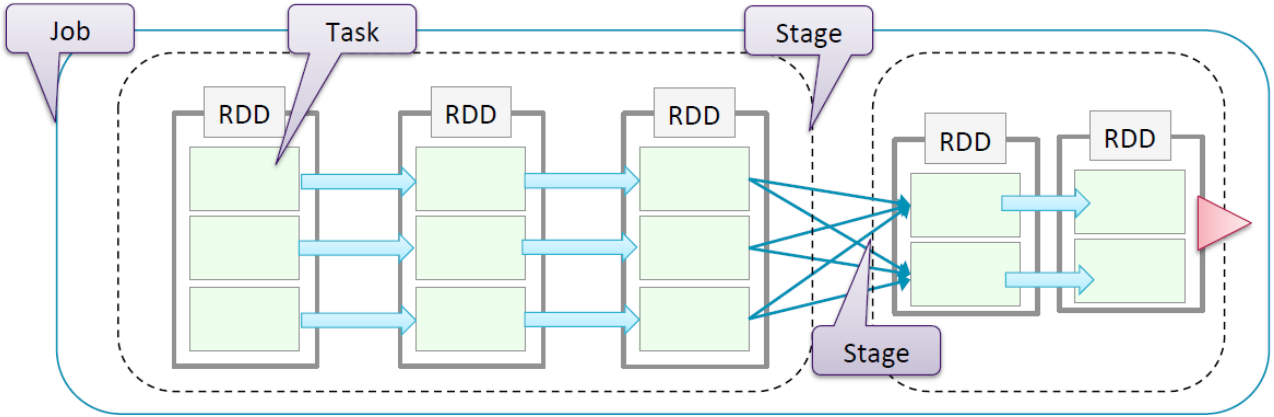


Driver / executor steps



Summary

- **Application:** represented by a SparkContext. Serves several jobs, RDD and shared variables
- **Job** – a set of tasks executed as a result of an action (arbitrary DAG)
- **Stage** – a set of tasks in a job that can be executed in parallel
- **Task** – an individual unit of work sent to one executor



Spark Configuration

1. Primary mechanism: SparkConf

```
val conf = new SparkConf()
conf.set("spark.app.name", "My Spark App")
conf.set("spark.master", "local[4]")
conf.set("spark.ui.port", "36000") // Override the default port
// Create a SparkContext with this configuration
val sc = new SparkContext(conf)
```

SparkConf provides methods to set common params like **setAppName()** and **setMaster ()**

2. Spark-submit provides flags for most common config and –conf to setup any configuration value

```
$ bin/spark-submit \
--class com.example.MyApp \
--master local[4] \
--name "My Spark App" \
--conf spark.ui.port=36000 \
myApp.jar
```

Spark Configuration

Spark-submit by default reads the conf/spark-default.conf file

- this file location can be customized

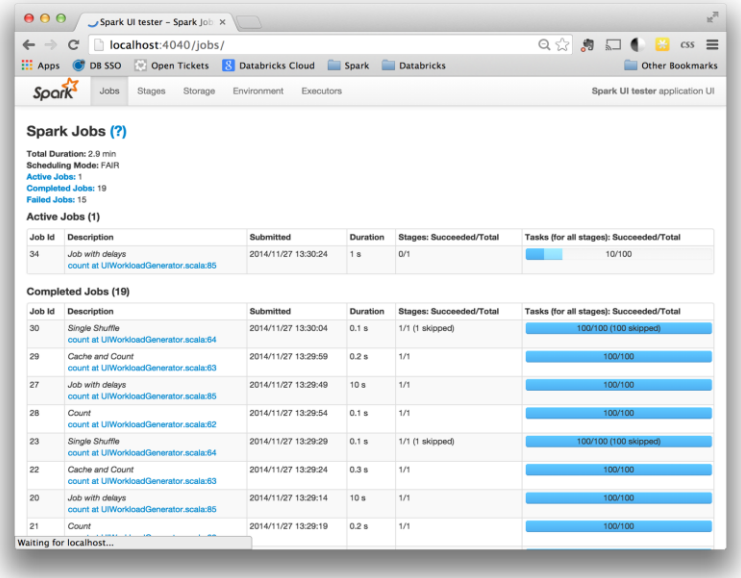
```
$ bin/spark-submit \
--class com.example.MyApp \
--properties-file my-config.conf \
myApp.jar
## Contents of my-config.conf ##
spark.master local[4]
spark.app.name "My Spark App"
spark.ui.port 36000
```

For a full list of configuration options see the Spark documentation
http://spark.apache.org/docs/latest/configuration.html

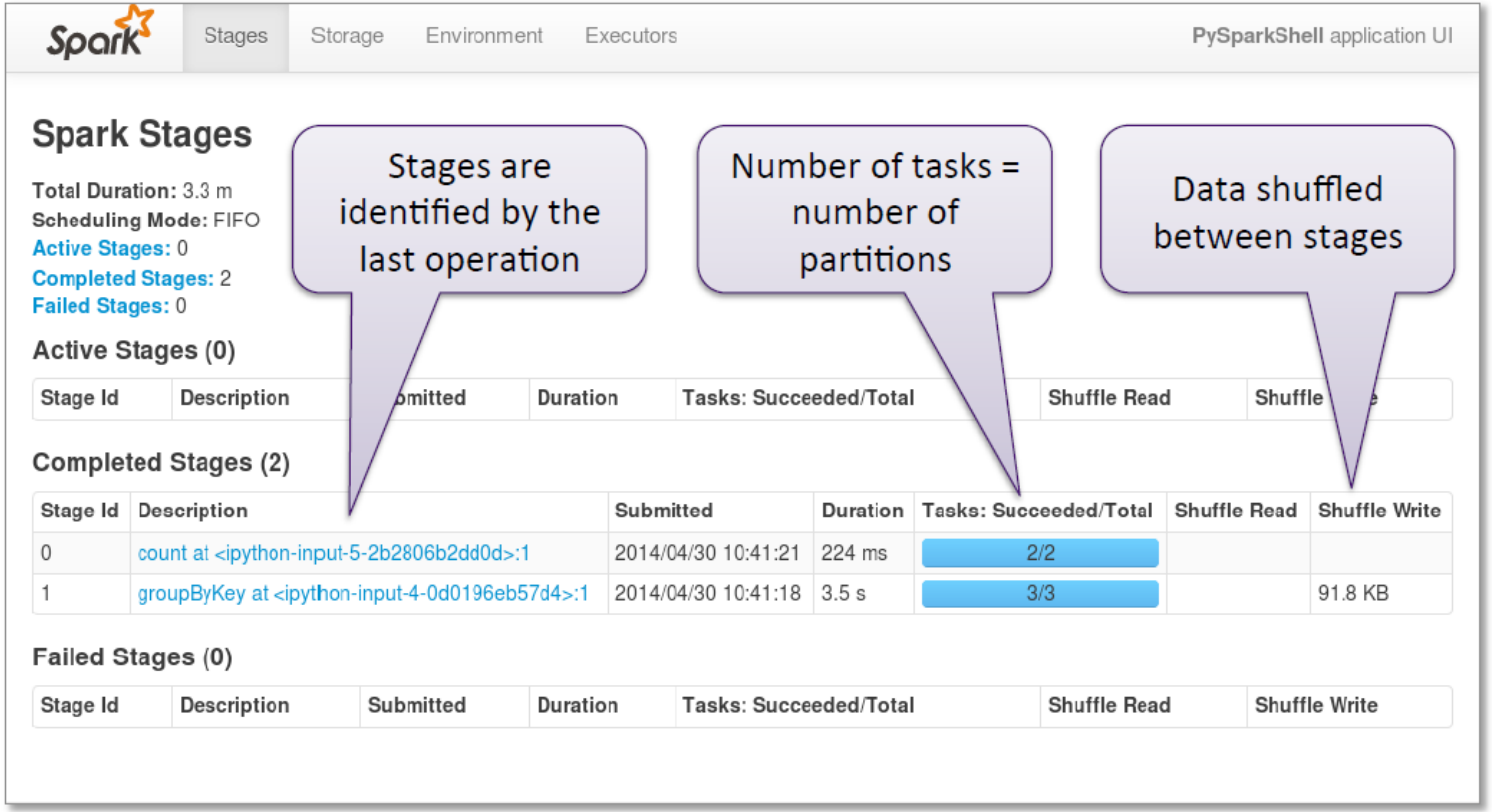
Spark web UI (1/2)

- Available at *http://<server-driver>:4040*
- *YARN: http://<ResourceManager>:4040*

- **Jobs**: Progress and metrics of stages, tasks, and more
- **Storage**: Information for RDDs that are persisted
- **Executors**: A list of executors present in the application
- **Environment**: Debugging Spark's configuration



Spark web UI (2/2)

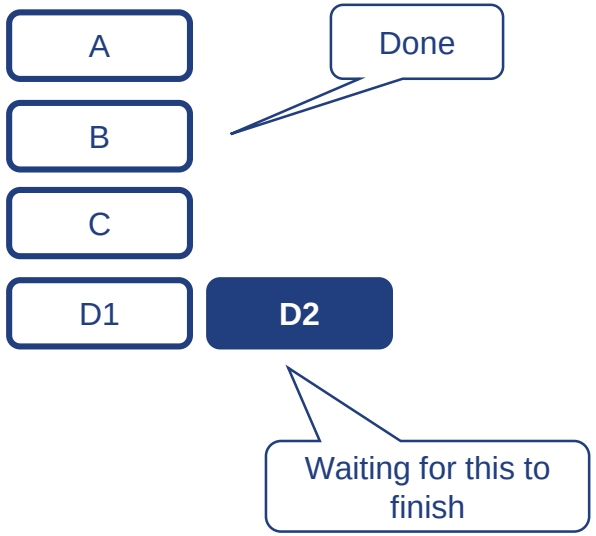


Drivers and Executor logs in cluster

- **YARN**
  - yarn logs -applicationId <*app ID*> when app has finished (logs are agregated)
- **In a Running Application**:
  - click through the ResourceManager UI to the Nodes page, then browse to a particular node, and from there, a particular container

Performance Tuning

- Level of Parallelism
  - Too little parallelism
    - Idle resources
    - Tasks to big. If more tasks than processors, waiting for the last task in the queue can be a problem
    - Be aware of memory/disk space for shuffle
  - Too much parallelism
    - Overhead for each partition computation
    - Network traffic



Performance Tuning

- What to tune
  - In operations that **shuffle data** you can set the degree of parallelism via parameter
  - A **RDD can be distributed** to have more or fewer partitions
    - **repartition()** shuffle a RDD into desired number of partitions
    - **coalesce()** to shrinking partitions and avoid shuffling

```
# Wildcard input that may match thousands of files
>>> input = sc.textFile("s3n://log-
files/2014/*.log")
>>> input.getNumPartitions()
35154
# A filter that excludes almost all data
>>> lines = input.filter(lambda line:
                        line.startswith("2014-10-17"))
>>> lines.getNumPartitions()
35154
# We coalesce the lines RDD before caching
>>> lines = lines.coalesce(5).cache()
>>> lines.getNumPartitions()
5
# Subsequent analysis can operate on the coalesced
RDD...
>>> lines.count()
```

Serialization

- Use Kryo (faster than the default java serialization)

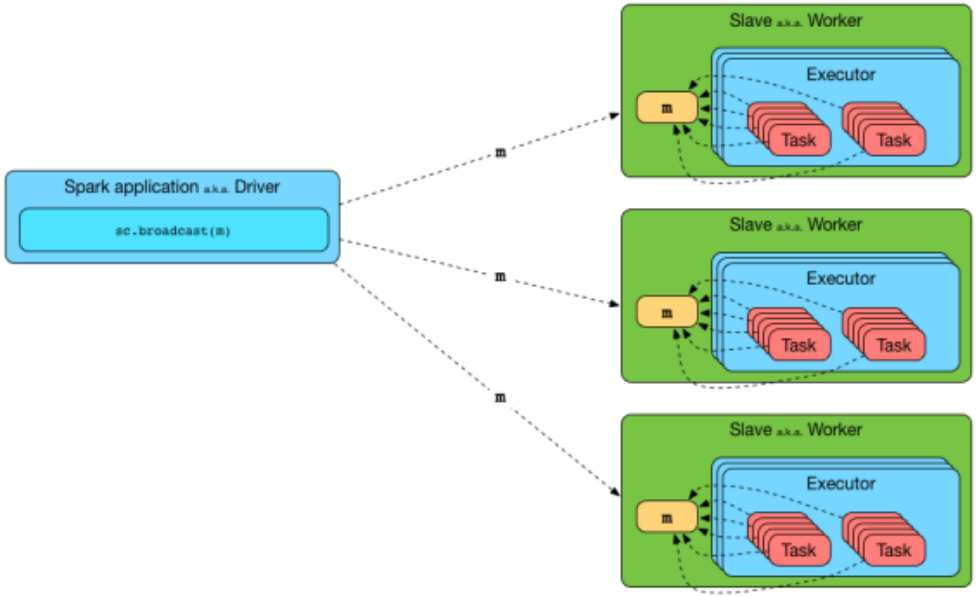
```
val conf = new SparkConf()
conf.set("spark.serializer",
"org.apache.spark.serializer.KryoSerializer")
// Be strict about class registration
conf.set("spark.kryo.registrationRequired", "true")
conf.registerKryoClasses(Array(classOf[MyClass],
classOf[MyOtherClass]))
```

Exercises

- **Exercise 9: Viewing Stages in the Spark Application UI**

Shared Variables

- Functions (map, filter,…) can use variables defined outside **them** in the driver program
- But each task running on the cluster gets a its own copy of each variable
- **Updates** from these copies **not return to driver**
- **Accumulators & Broadcast** variables relax this restriction for aggregation and broadcasts



Shared variables – Accumulators

- Aggregation values from **workers to the driver**
- For example count some kind of events during job execution
- Accumulator of empty line counts in Scala:

```
val file = sc.textFile("file.txt")

// Create an Accumulator[Int] initialized to 0
val blankLines = sc.accumulator(0)
val callSigns = file.flatMap(line => {
      if (line == "") {
        // Add to the accumulator
        blankLines += 1
      }
      line.split(" ")
})
callSigns.saveAsTextFile("output.txt")
println("Blank lines: " + blankLines.value)
```

- we will see the **right count** only ***after*** we run the saveAsTextFile() action, because the transformation above it, map(), is lazy
- accumulators are ***write-only*** variables. This allows accumulators to be implemented efficiently

Shared variables – Accumulators

- **Accumulators and Fault tolerance**

  - The same function may run **multiple times** on the same data
    - An operation crashes
    - A node is slow and a “speculative” copy of the task is launched on other node
    - Rebuild of a cached value that falls out of memory
  - For **accumulators used in actions**, Spark applies each task’s update to each accumulator **only once** (can be used actions like foreach() and is **OK**)
  - But for **accumulators used in RDD transformations** instead of actions, this guarantee does not exist (use only for debugging, **NOT OK**)

Shared Variables – Broadcast

- ***Broadcast variables***, allow the program to efficiently send a large, **read-only** value to all the worker nodes for use in one or more Spark operations
- Like send a large, read-only lookup table to all the nodes
- Sparks send **all variables in your closures** to the worker nodes but can be **inefficient**:
  - The default task launching mechanism is optimized for small task sizes
  - You might, in fact, use the same variable in *multiple* parallel operations, but Spark will **send it separately for each operation**
- Broadcast variables are read-only
- Example

```
val signPrefixes = sc.broadcast(loadCallSignTable())
val countryContactCounts = contactCounts.map{ case(sign, count) =>
          val country = lookupInArray(sign, signPrefixes.value)
          (country, count)
}.reduceByKey((x, y) => x + y)
countryContactCounts.saveAsTextFile(outputDir + "/countries.txt")
```

Exercises

- Exercise 10: Using Broadcast Variables
- Exercise 11: Using Accumulators
- Exercise 12: Average WordLength


INTRODUCTION TO SPARK
4. Spark Architecture
Load and Save Data

Load and Save Data

- Spark can access data used by Hadoop (S3, HDFS, Cassandra Hbase, etc…) and local filesystem
- Common supported file formats

Format name	Structured	Comments
Text files	No	Plain old text files. Records are assumed to be one per line.
JSON	Semi	Common text-based format, semistructured; most libraries require one record per line.
CSV	Yes	Very common text-based format, often used with spreadsheet applications.
SequenceFiles	Yes	A common Hadoop file format used for key/value data.
Protocol buffers	Yes	A fast, space-efficient multilanguage format.
Object files	Yes	Useful for saving data from a Spark job to be consumed by shared code. Breaks if you change your classes, as it relies on Java Serialization.

Load and Save Data – Text Files

```
val input = sc.textFile(“/user/gftbigdata/input-spark/core-site.xml")
```

- **Multiples Files** (in pair RDD, key name of file)
  - Support directories & wildcards
  - Average value per file (key – filename; value – file contents)

```
val input = sc.wholeTextFiles("file:///home/holden/salesFiles")
val result = input.mapValues{ y =>
    val nums = y.split(" ").map(x => x.toDouble)
    nums.sum / nums.size.toDouble
}
```

Load and Save Data

- **Loading CSV with textFile() in Scala and opencsv Library**
  - No newline inside a field

```
import java.io.StringReader
import au.com.bytecode.opencsv.CSVReader
...
val input = sc.textFile(inputFile)
val result = input.map( line =>
     val reader = new CSVReader(new StringReader(line));
     reader.readNext();
)
```

  - New lines inside a field

```
case class Person(name: String, favoriteAnimal: String)
val input = sc.wholeTextFiles(inputFile)
val result = input.flatMap( case (_, txt) =>
     val reader = new CSVReader(new StringReader(txt));
     reader.readAll().map(x => Person(x(0), x(1)))
)
```

(FileName, content)

Load and Save Data

- Saving RDD as a text file

```
result.saveAsTextFile(outputFile)
```

- For CVS: Write a function that converts the fields to given positions in an comma separated line and save as text file.

Load and Save Data -- Sequence Files

- Flat files with key/value pairs
- Sync markers that allow Spark read SequenceFiles in parallel from multiples nodes
- The elements implement Hadoop's Writable Interface Like IntWritable, Text …
- Add dependencies for hadoop to your sbt project
- Always map the loading to Scala types for doing computations.

```
val data = sc.sequenceFile(inFile, classOf[Text],
classOf[IntWritable])
.map{case (x, y) => (x.toString, y.get())}
```

- Or use sequenceFile[Key, Value](path) that returns native Scala types

Load and Save Data -- Sequence Files

- Saving sequenceFile

```
val data =
   sc.parallelize(List(("Panda", 3),
   ("Kay", 6), ("Snail", 2)))
data.saveAsSequenceFile(outputFile)
```

Load and Save Data – Hadoop

- Spark also can interact with Hadoop input and output formats
- Example loading with new KeyValueTextInputFormat for reading key/value data from text files

```
val input = sc.newAPIHadoopFile[Text, Text,
KeyValueTextInputFormat](inputFile).map {
case (x, y) => (x.toString, y.toString)
}
```

- Example saving with a sequence file output format

```
result.saveAsNewAPIHadoopFile(fileName, Text.class,
IntWritable.class, SequenceFileOutputFormat.class);
```

Load and Save data – File Systems

- **Local filesystem:**
  - Requires that the files are available at the same path on all nodes in your cluster. If not load locally on the driver and then parallelize

```
val rdd =   sc.textFile("file:///home/gftbigdata/hola.txt")
```

  - Amazon S3:
    - Set AWS_ACCESS_KEY_ID and AWS_SECRET_ACCESS_KEY environment variables in each worker
    - Fast if nodes are in EC2

```
val rdd = sc.textFile("s3n://bucket/pathwithin-bucket")
```

- **HDFS**
  - Take advantage of this data locality to avoid network overhead

```
val rdd = sc.textFile("hdfs://master:port/path")
```

Load and Save Data – SQL

- **Spark SQL** ready for working with structured data in a relational way
- Has its own context (see later)
- Can load **Hive tables** (SQL like abstraction over Hadoop map-reduce)
  - Copy hive-site.xml to Spark conf directory
  - And use HiveContext which is a specialization of SQLContext

```
import org.apache.spark.sql.hive.HiveContext
val hiveCtx = new HiveContext(sc)
val rows = hiveCtx.sql("SELECT name, age FROM users")
val firstRow = rows.first()
println(firstRow.getString(0)) // Field 0 is the name
```

Load and Save Data – Databases

- Loading data from a database

```
def createConnection() = {
   Class.forName("com.mysql.jdbc.Driver").newInstance();
   DriverManager.getConnection("jdbc:mysql://localhost/test?user=holden");
}
def extractValues(r: ResultSet) = { (r.getInt(1), r.getString(2)) }
val data = new JdbcRDD(sc, createConnection,
         "SELECT * FROM panda WHERE ? <= id AND id <= ?", lowerBound = 1,
         upperBound = 3, numPartitions = 2, mapRow = extractValues)
println(data.collect().toList)

override def readFields(r: ResultSet) = {
      // blank since only used for writing
}
```

Load and Save Data – Databases

- Saving data to a database (using Hadoop DBOutputFormat)

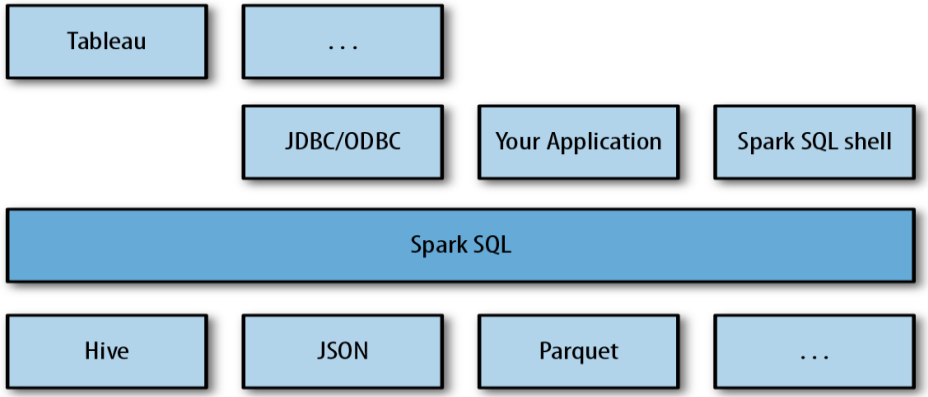
```
val records = data.map(e => (catRecord(e._1, e._2), null))
val tableName = "table"
val fields = Array("name", "age")
val jobConf = new JobConf()
DBConfiguration.configureDB(jobConf, "com.mysql.jdbc.Driver",
   "jdbc:mysql://localhost/test?user=holden")
DBOutputFormat.setOutput(jobConf, tableName, fields:_*)
records.saveAsHadoopDataset(jobConf)
[…]
case class catRecord(name: String, age: Int) extends DBWritable {
    override def write(s: PreparedStatement) {
      s.setString(1, name)
      s.setInt(2, age)
    }
    override def readFields(r: ResultSet) = {
      // blank since only used for writing
    }
}
```

INTRODUCTION TO SPARK

5. Spark SQL and Data Frames

Spark SQL

- Interface for working with structured (schema) and semi-structured data
- Three main capabilities:
  - **DataFrame** abstraction for structured datasets. Similar to tables in Relational database.
  - Read & write in structured formats (JSON, Hive, Parquet)
  - Query data using SQL inside Spark program & from external tools using JDBC/ODBC



DataFrame

- Extension of the RDD model
- A DataFrame contains
  - RDD of **Row** objects, each representing a record
  - Schema (i.e., data fields) of its rows
  - Data is stored in a more efficient manner than RDDs (due the schema)
  - Provide new operations and can run SQL queries
  - Creation from:
    - External data sources
    - Result of queries
    - Regular RDDs

Linking with Spark SQL

- Add dependencies:
  - If you want **Hive support** (hive tables, UDFS, SerDes & HiveQL). **Recommended**. Does not require a Hive installation

```
libraryDependencies += "org.apache.spark" % "spark-hive_2.10" % "1.5.0
```

  - If you want only **SQL support**

```
libraryDependencies += "org.apache.spark" % "spark-sql_2.11" % "1.5.0"
```

- Entry points
  - SQLContext: extends standard context
  - HiveContext: extends SQLContext (recommended since **HiveQL** is the **standard query language** in Spark)
- If you have a Hive installation copy **hive-site.xml** to Spark conf directory.
- If not, Spark SQL will create its own Hive metastore in your program's working directory **metastore_db** and tables will be placed in **/user/hive/ware-house** in local filesystem or HDFS if you have **hdfs-site.xml** on your classpath

Using SparkSQL in Applications

- Easy Data load and query combining with “regular” code
- Import Contexts

```
import org.apache.spark.sql.hive.HiveContext    // Import Spark SQL
//import org.apache.spark.sql.SQLContext // Or if you can't have hive
```

- Create a SQL context

```
val sc = new SparkContext(...)
val hiveCtx = new HiveContext(sc)
```

- Import implicits to convert RDDs into specialized RDDs for querying

```
import hiveCtx.implicits._ // Import the implicit conversions
```

- Basic Query: Load some JSON twitter data. Register it as “temporay table” for SQL query

```
val input = hiveCtx.jsonFile(inputFile)
input.registerTempTable("tweets") // Register the input schema RDD
// Select tweets based on the retweetCount
val topTweets = hiveCtx.sql("SELECT text, retweetCount FROM tweets ORDER BY
retweetCount                                    LIMIT 10")
```

Using SparkSQL in Applications

- **DataFrames** provide an access to the RDD: **.rdd()** so you can user RDD transformations like map() or filter. Example access first column:

```
val topTweetText = topTweets.rdd().map(row => row.getString(0))
```

- With **registerTempTable()** method, Hive context can query the DataFrame (as in the example)
- Tables created are local to HiveContext. When the app ends the go away
- *Basic DataFrame operations*

```
Name, Age {Row("Bear",None),
Row("Databricks", 1)}
```

Function name	Purpose	Example
`show()`	Show the contents of the DataFrame	`df.show()`
`select()`	Select the specified fields/ functions	`df.select("name", df("age")+1)`
`filter()`	Select only the rows meeting the criteria	`df.filter(df("age") > 19)`
`groupBy()`	Group together on a column, needs to be followed by an aggergation like `min()`, `max()`, `mean()` or `agg()`.	`df.groupBy(df("name")).min()`

Types stored in DataFrames

- All of these types can also be nested within each other
- For example, you can have arrays of **structs**, or maps that contain **structs**
- **structs** are simply represented as other Rows in Spark SQL

Spark SQL/HiveQL type	Scala type
TINYINT	Byte
SMALLINT	Short
INT	Int
BIGINT	Long
FLOAT	Float
DOUBLE	Double
DECIMAL	scala.math.BigDecimal
STRING	String
BINARY	Array[Byte]
BOOLEAN	Boolean
TIMESTAMP	java.sql.Timestamp
ARRAY<DATA_TYPE>	Seq
MAP<KEY_TYPE, VAL_TYPE>	Map
STRUCT<COL1: COL1_TYPE, ...>	Row

Caching in Spark SQL

- Caching in Spark SQL works different:
  - For an efficient memory use

    ```
    hiveCtx.cacheTable("tableName")
    ```

  - Memory cache can be also achieved using HiveSQL statements

    ```
    CACHE TABLE tableName or UNCACHE TABLE tableName.
    ```

- When caching a table, Spark SQL represents the data in an **in-memory columnar format**.
- The **cached table** will **remain in memory only** for the **life** of our driver program.

Loading and Saving Data

- In addition to Hive, JSON and Parquet, Spark SQL has a **DataSource Api** for integration
- Some notable implementations include Avro, Hbase, ES, Cassandra…
- It is possible **convert a regular RDDs to DataFrames** by assigning them a schema

Exercises

- **Exercise 13: Data Mining Using SparkSQL**

Loading and Saving Data

- Apache Hive (seen before):

```
import org.apache.spark.sql.hive.HiveContext
val hiveCtx = new HiveContext(sc)
val rows = hiveCtx.sql("SELECT key, value FROM mytable")
val keys = rows.map(row => row.getInt(0))
```

Loading and Saving Data

- Parquet, column-oriented storage formats can store records with nested fields efficiently

```
// Load some data in from a Parquet file with field's name and favouriteAnimal
rows = hiveCtx.load(parquetFile, "parquet")
names = rows.map(row => row.name)
println ("Everyone“)
names.collect().foreach (println)
tbl = rows.registerTempTable("people") // Find the panda lovers
pandaFriends = hiveCtx.sql("SELECT name FROM people WHERE
favouriteAnimal = ‘panda’")
println ("Panda friends“)
pandaFriends.map(row => row.name).collect().foreach (println)
pandaFriends.save("hdfs://...", "parquet")
```

Loading and Saving Data

- **Parquet**, column-oriented storage formats can store records with nested fields efficiently
- **Avro** a data serialization system which relies on schemas
  - Include in sbt project:

```
libraryDependenicies += "com.databricks" %%
"spark-avro" %    "2.0.1"
```

  - In previous example, replace “parquet” by “com.databricks.spark.avro”

Loading and Saving Data

- JSON: Spark SQL can infer the schema by scanning the file and let you access fields by name

```
{"name": "Holden"}
{"name":"Sparky The Bear", "lovesPandas":true, "knows":{"friends":
["holden"]}}
```

```
val input = hiveCtx.jsonFile(inputFile)
input.printShema()
root
|-- knows: struct (nullable = true)
| |-- friends: array (nullable = true)
| | |-- element: string (containsNull = false)
|-- lovesPandas: boolean (nullable = true)
|-- name: string (nullable = true)
```

- Access with SQL

```
select knows.friends(0) from table_input limit 1;
```

Loading and Saving Data

- From RDDs: Any RDD with case classes is implicitly converted into a DataFrame

```
case class HappyPerson(handle: String, favouriteBeverage: String)
// Create a person and turn it into a Schema RDD
val happyPeopleRDD = sc.parallelize(List(HappyPerson("holden",
"coffee")))
// Note: there is an implicit conversion
// that is equivalent to
// sqlCtx.createDataFrame(happyPeopleRDD)
happyPeopleRDD.registerTempTable("happy_people")
```

- You can also use .toDF

```
val df = sc.parallelize(List(HappyPerson("holden",
"coffee"))).toDF
```

JDBC /ODBC Server

- Useful for connection BI tools to Spark Cluster
- JDBC server runs a standalone Spark driver that can be shared with multiple clients
- JDBC server corresponds to HiveServer2 in Hive (is a Thrift server) listen in port**:10000**

```
./sbin/start-thriftserver.sh --master sparkMaster
```

- The Server can share cached tables between multiples programs since is a single driver program

Working with Beeline

- Beeline: SQL shell client can connect to the server via JDBC

```
spark$ ./bin/beeline –u jdbc:hive2://localhost:10000
0: jdbc:hive2://localhost:10000> show tables;
+----------+
| result   |
+----------+
| pokes    |
+----------+
1 row selected (1.182 seconds)
```

Working with Beeline

- You can use standard HiveQL

```
> CREATE TABLE IF NOT EXISTS mytable (key INT, value STRING)
ROW FORMAT DELIMITED FIELDS TERMINATED BY ',';
> LOAD DATA LOCAL INPATH 'int_string.csv'
INTO TABLE mytable;
> SHOW TABLES;
Mytable
```

- To See the query plan

```
spark-sql> EXPLAIN SELECT * FROM mytable where key = 1;
== Physical Plan ==
Filter (key#16 = 1)
HiveTableScan [key#16,value#17], (MetastoreRelation default, mytable,
None), None
```

- Spark SQL also support a simple shell for local development

```
./bin/spark-sql
```

Spark SQL UDFs

- UDFs allow registering custom functions to be called within SQL
- Spark SQL offers an easy method to register UDFs

```
hiveCtx.udf.register("strLenScala", (_:
String).length)
val tweetLength = hiveCtx.sql("SELECT
strLenScala('tweet') FROM tweets LIMIT 10")
```

- Using existing Hive UDFS
  - Standard Hive UDFs are already included
  - The jars of your UDFs have to be included in the project. If using JDBC server add - -jar command-line flag
  - To make UDFs available

```
hiveCtx.sql("CREATE TEMPORARY FUNCTION name AS
class.function")
```

Exercises

- Exercise 14: Running spark jobs inside IntelliJ IDE
- Exercise 15: Dataframes
- Exercise 16: Dataframes
- Exercise 17: Spark RDD testing
- Exercise 18: Spark DataFrame testing exercise


INTRODUCTION TO SPARK
6. Spark Streaming
and Machine Learning

Spark Streaming

Consume data in real-time

- Micro-batching

Stateful exactly-once semantics out of the box

- recovers both lost work and operator state (*e.g.* sliding windows) without any extra code

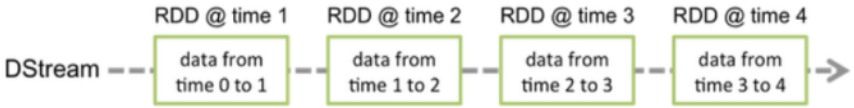
Same API than Spark core

- Combine streaming with batch and interactive queries
- Write streaming applications the same way you write batch jobs

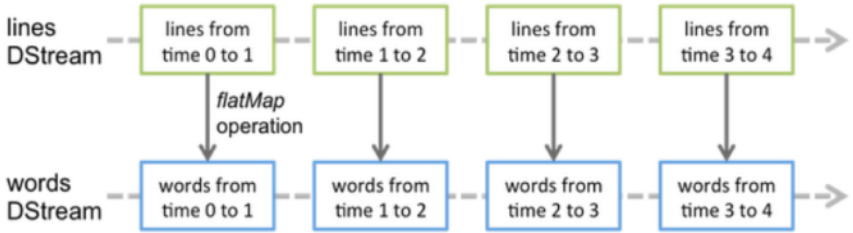
```
stream.join(historicCounts).filter {
  case (word, (curCount, oldCount)) =>
    curCount > oldCount
}
```

Spark Streaming

DStream as a sequence of RDDs



Same RDD operations apply to DStreams



Spark Streaming

Data sources

- Also, custom receivers

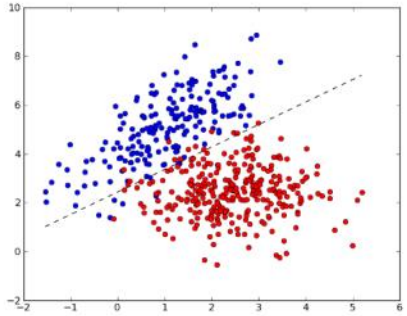
Example

- Each second (batch) get the number of tweets about “Spark” of the last 5 seconds (window)

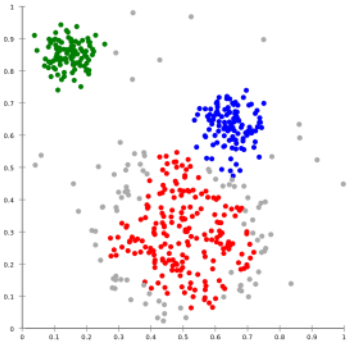
```
val streamCtx = new StreamingContext(sparkCts, Seconds(1))
val tweets_stream = TwitterUtils.createStream(streamCtx, …)
val spark_tweets = tweets_stream.filter(_.getText.contains("Spark"))
val counts_per_window = spark_tweets.countByWindow(Seconds(5))
val counts_per_window = counts_per_window.print()
```

MLib

- Algorithms
  - **Classification** (discrete) and **regression** (continue)
    - *e.g*: credit scoring, marketing campaigns, …
    - Linear models: SVMs, logistic regression, linear regression
    - Naive Bayes
    - Decision trees



  - **Collaborative filtering**
    - *e.g*: recommenders
    - Alternating least squares – ALS



  - **Clustering**
    - *e.g*: social network analysis (communities)
    - K-Means

Exercises

- Exercise 14: Recommender Algorithms
- Exercise 15: Deploy program in a Spark Cluster

Thank you

LinkedIn
https://es.linkedin.com/in/chicochica10