Distributed computing with

Spark

Javier Santos

April,13 - 2015

About me

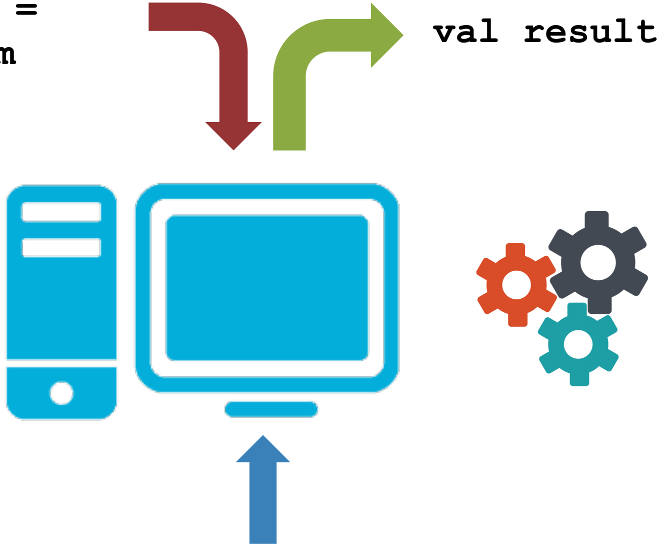
Javier Santos ***@jpaniego***

«Hay dos formas de programar sin errores; solo la tercera funciona»

Alan J Perlis

What’s Spark?

```
val action: Seq[Int] => Int =
  (seq: Seq[Int]) => seq.sum
```

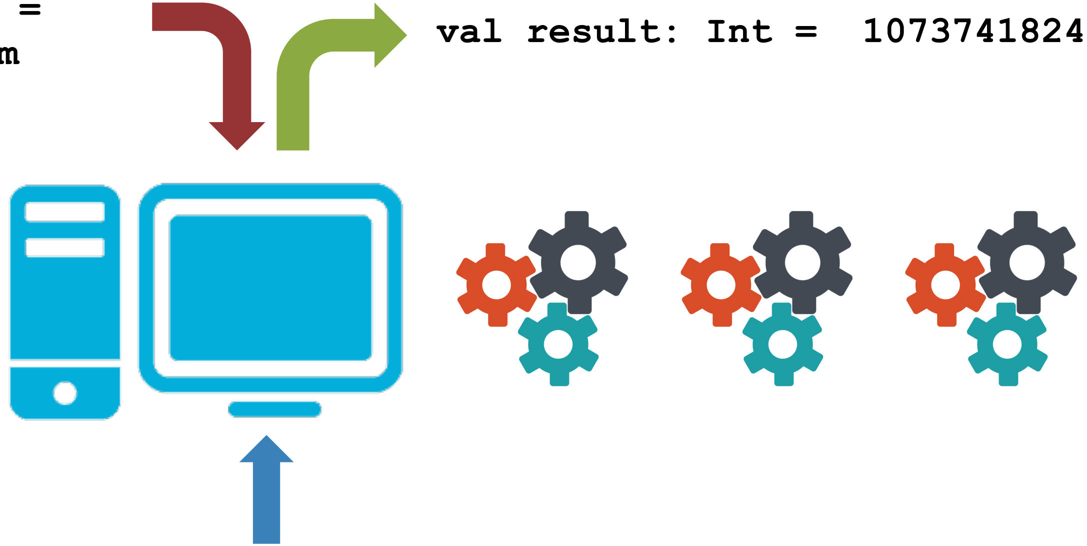


```
val result: Int = 15
```

```
val mySeq: Seq[Int] = 1 to 5
```

What's Spark?

```
val action: Seq[Int] => Int =
  (seq: Seq[Int]) => seq.sum
```



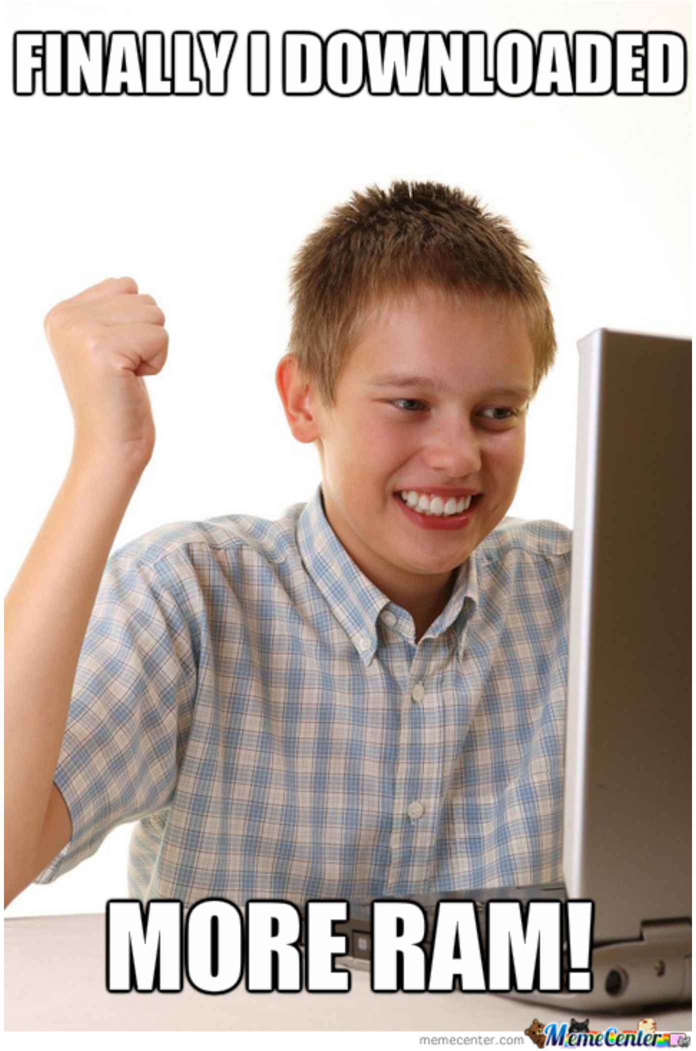
```
val result: Int =  1073741824
```

```
val mySeq: Seq[Int] = 1 to Int.MaxValue
```

What’s Spark?

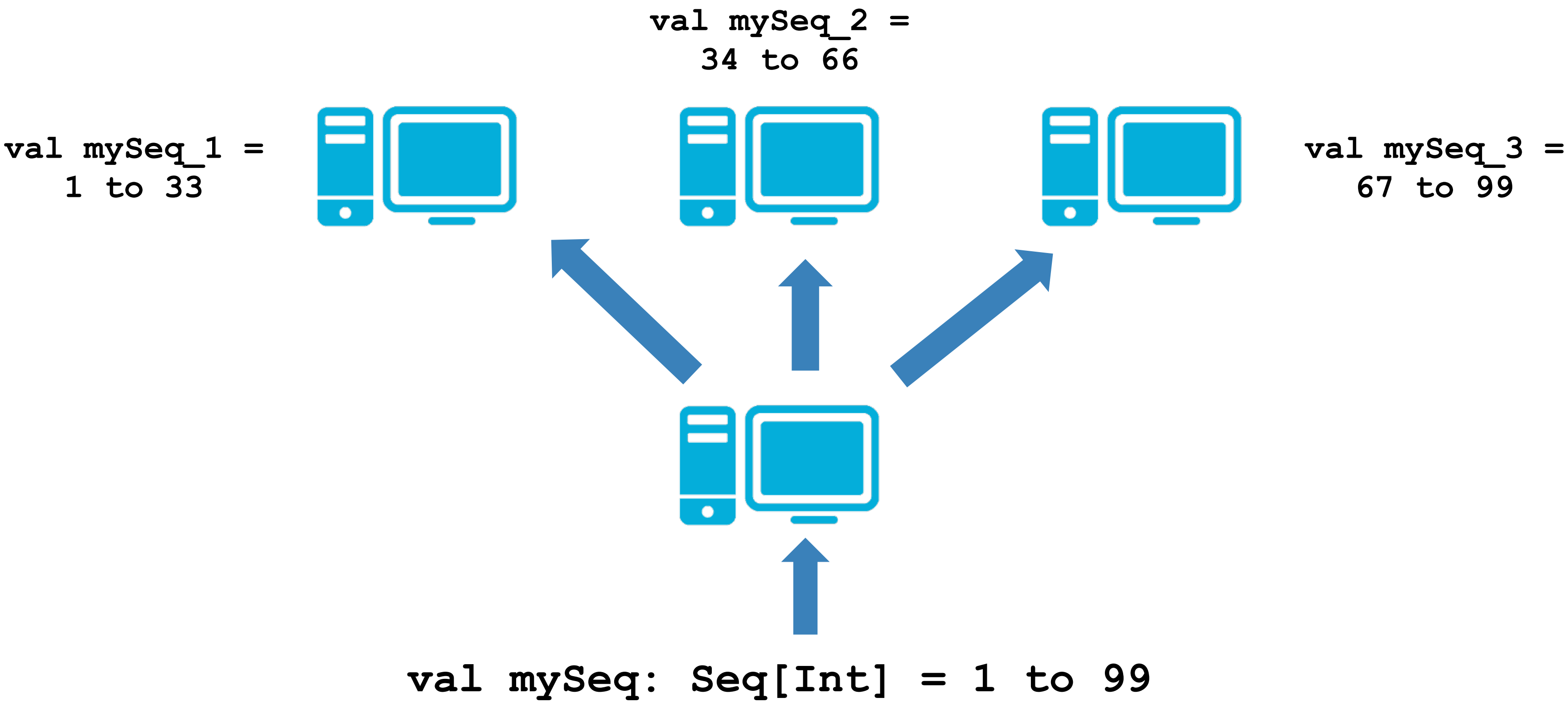
Computing problems:

- Time: Waiting for action results on tons of records
- Space: How do we allocate TB of data?

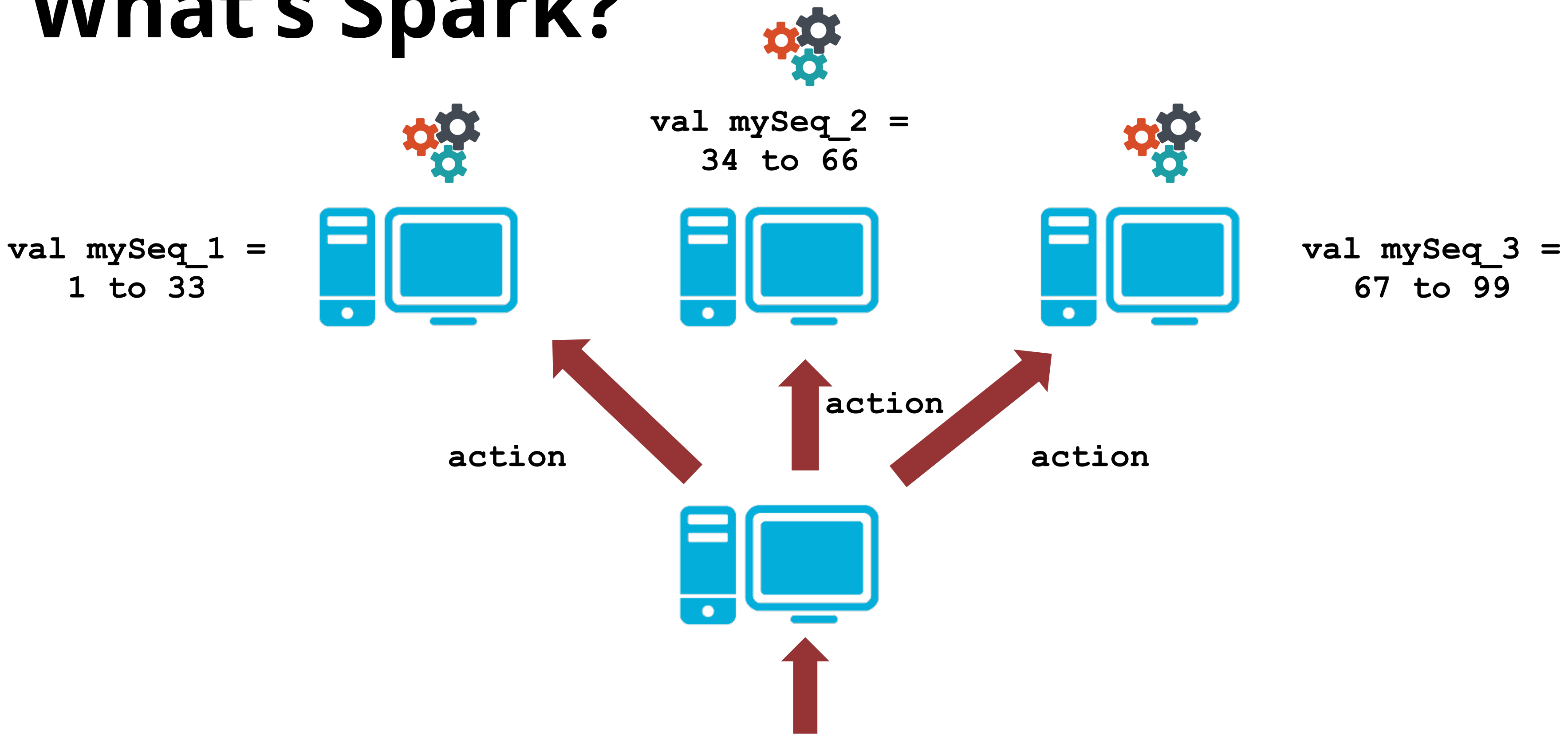


downloadmoreram.com

What’s Spark?

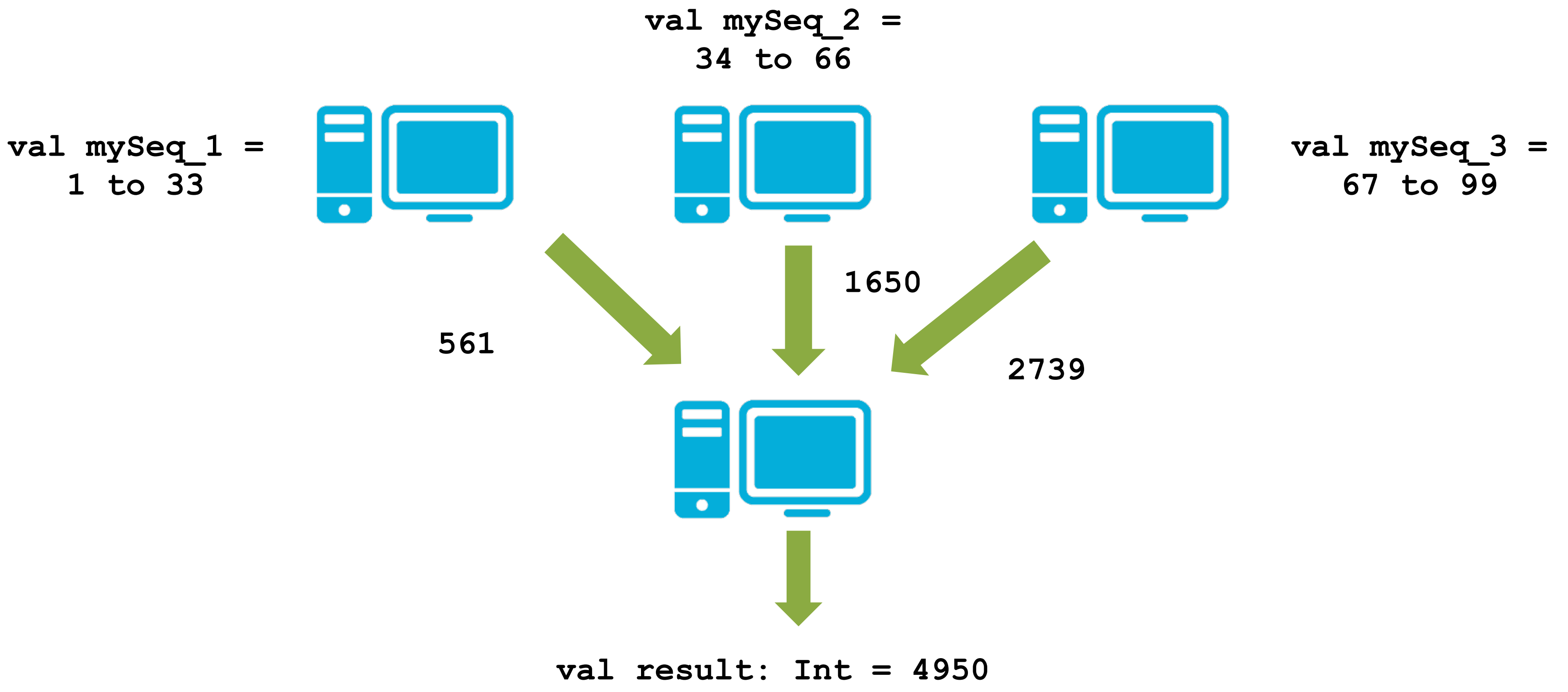

val mySeq_2 =
34 to 66
val mySeq_1 =
1 to 33
val mySeq_3 =
67 to 99
val mySeq: Seq[Int] = 1 to 99

What’s Spark?



val action: Seq[Int] => Int = (seq: Seq[Int]) => seq.sum

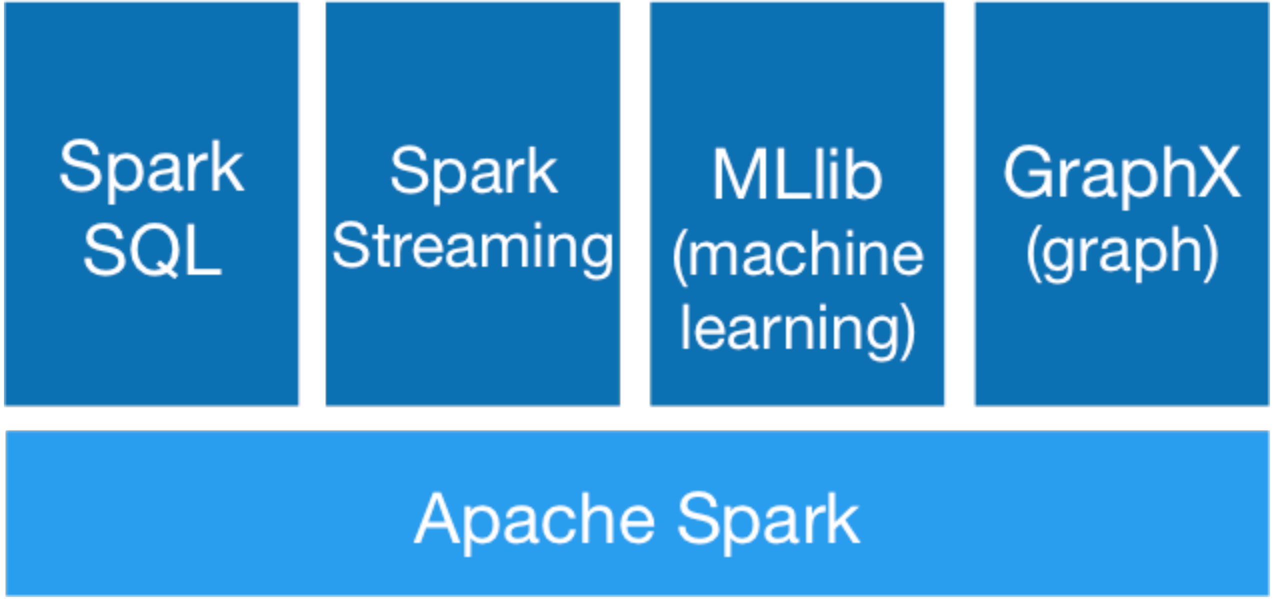
What's Spark?


val mySeq_2 =
34 to 66
val mySeq_1 =
1 to 33
val mySeq_3 =
67 to 99
1650
561
2739
val result: Int = 4950

What’s Spark?

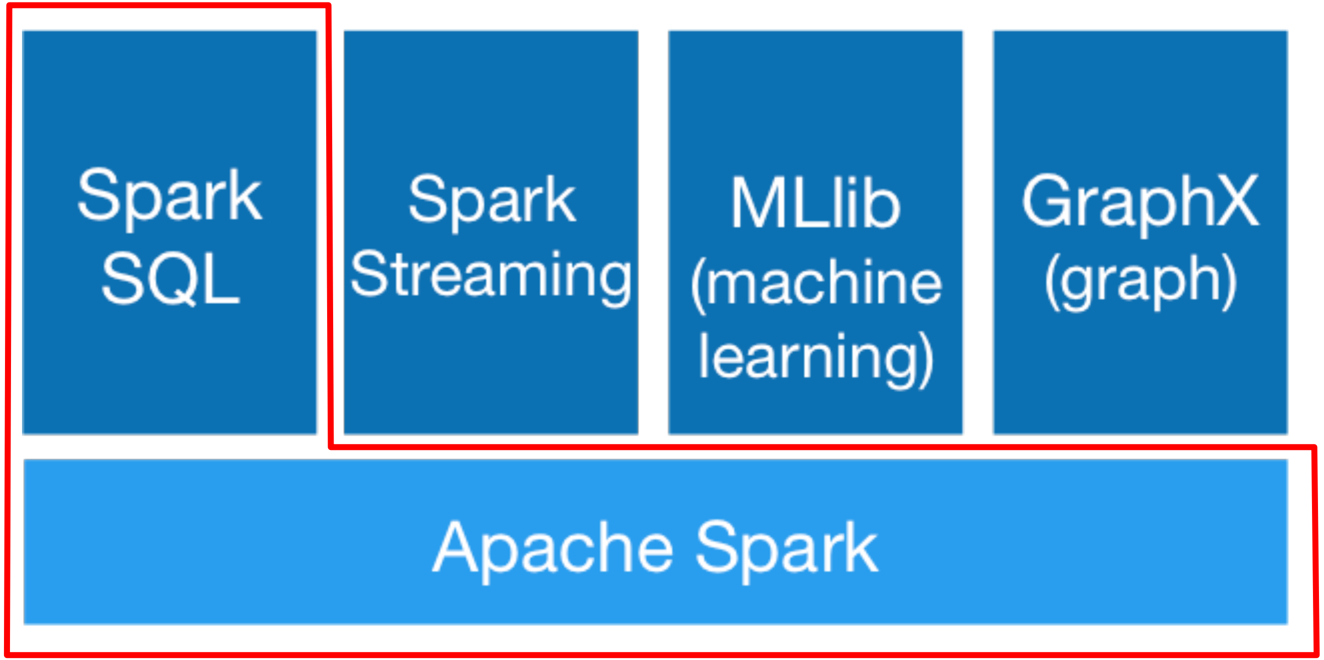
- General engine for large-scale data processing.
- “In-memory” data allocation.
- Target: **data** and **computing** parallelism.
- Mostly developed in Scala.
  It also provides support for Python and Java
- First developed by AMPLab in UC Berkeley
- Support by Databricks
- Latest stable version 1.3.0

Modules



Spark SQL
Spark Streaming
MLlib (machine learning)
GraphX (graph)
Apache Spark

Modules


Spark SQL
Spark Streaming
MLlib (machine learning)
GraphX (graph)
Apache Spark

MapReduce model

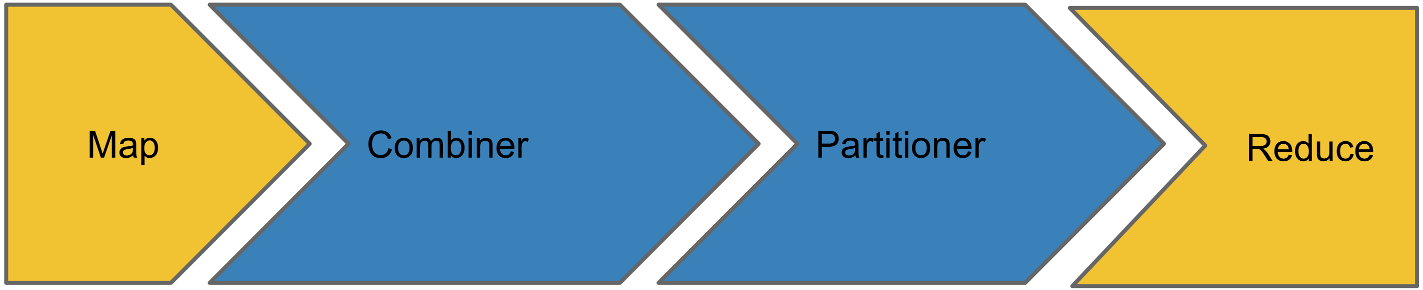
- Programming model used for supporting parallel-computing and big data sets distribution among computing groups

- Based on two main functional programming methods: Map and reduce.

- Hadoop
  - One of the first open source implementation
  - First implementation lead by Yahoo

MapReduce model

- Phases



MapReduce model

Partitions

- Example (**Map**)

"123;SamsungGalaxyIII;14:05;300;34612345678"
"124;LGNexus4;16:05;121;+34613455678"
"126;LGNexus4;12:05;23;+3463624678"
"131;NokiaLumia;14:05;300;+34613246778"

(SamsungGalaxyIII, 1)
(LGNexus4, 1)
(LGNexus4, 1)
(NokiaLumia, 1)

"125;MotorolaG2;14:05;300;+34612345678"
"127;NokiaLumia;14:05;300;+34612345678"
"130;NokiaLumia;14:05;300;+34612345678"

(MotorolaG2, 1)
(NokiaLumia, 1)
(NokiaLumia, 1)

MapReduce model

- Example (**Combiner**)

(SamsungGalaxyIII, 1)
(LGNexus4, 1)
(LGNexus4, 1)
(NokiaLumia, 1)

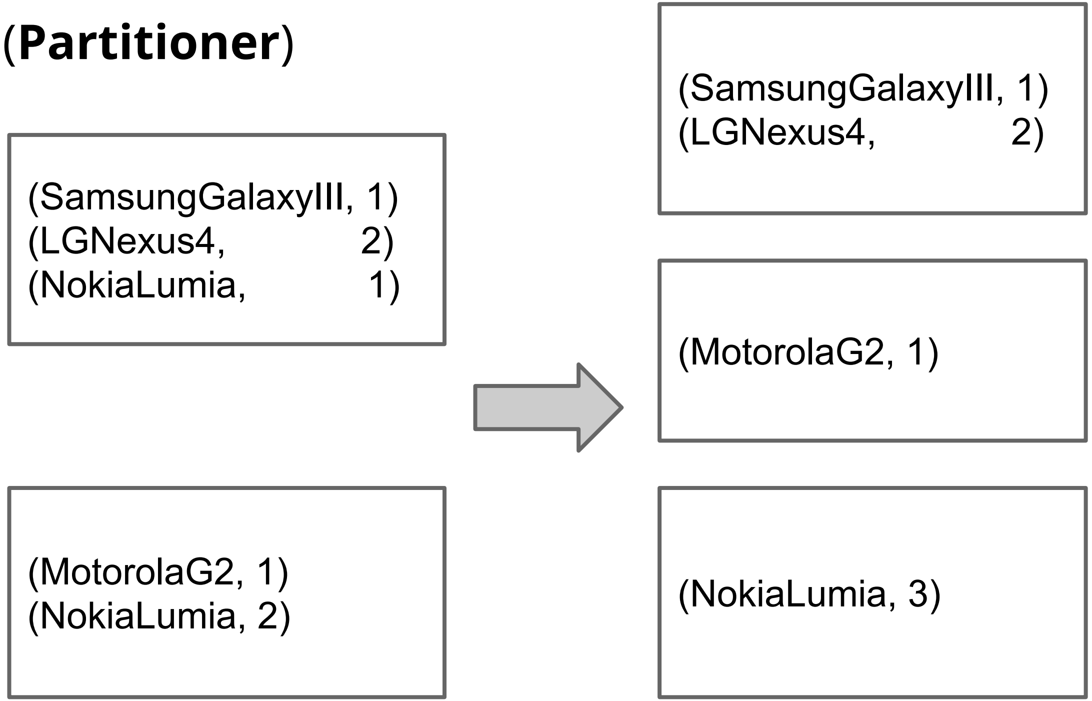
(SamsungGalaxyIII, 1)
(LGNexus4, 2)
(NokiaLumia, 1)

(MotorolaG2, 1)
(NokiaLumia, 1)
(NokiaLumia, 1)

(MotorolaG2, 1)
(NokiaLumia, 2)

MapReduce model

- Example (**Partitioner**)



MapReduce model

- Example (**Reduce**)

(SamsungGalaxyIII, 1)
(LGNexus4, 2)

(MotorolaG2, 1)

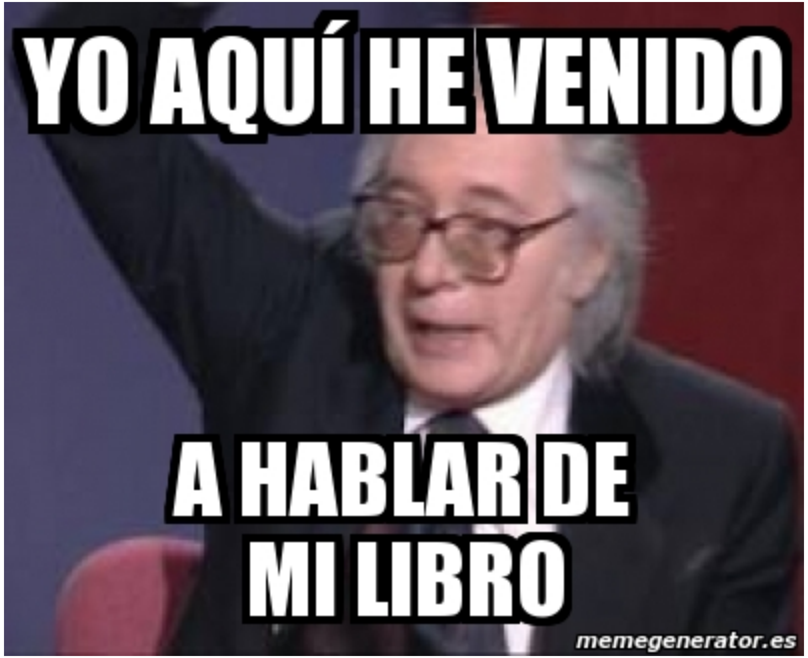
(NokiaLumia, 3)

ReduceFunction

```
f(t1:(String,Int),t2:(String,Int)) =
  if (t1._2 > t2._2) t1 else t2
```

(NokiaLumia, 3)

So...what about Spark?

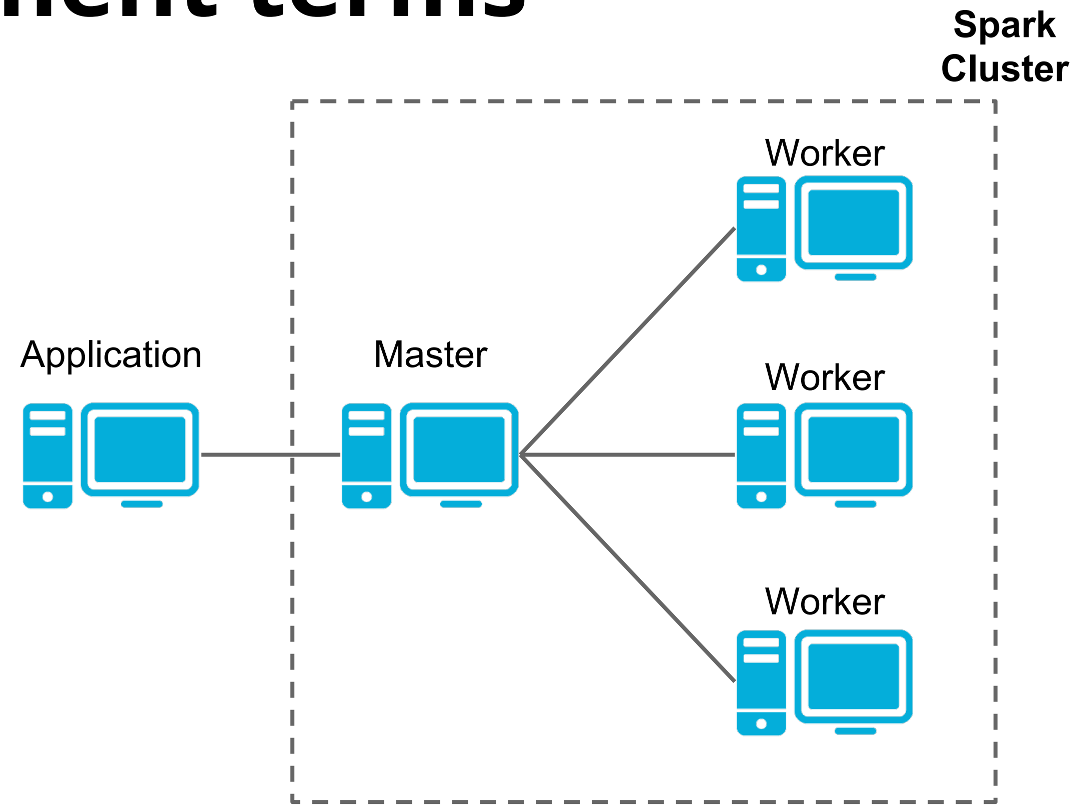


Deployment types

- Local
  - `master=”local[N]”` (N=Amount of cores)
  - Spark master is launched in the same process.
    It’s not accessible from web.
  - Workers are launched in the same process.

- Standalone
  - `master=”spark://master-url”`
  - Spark master is launched in a cluster machine.
  - For deploying workers, it’s used ‘start-all.sh’ script.
  - Submit JAR to worker nodes.

Deployment terms

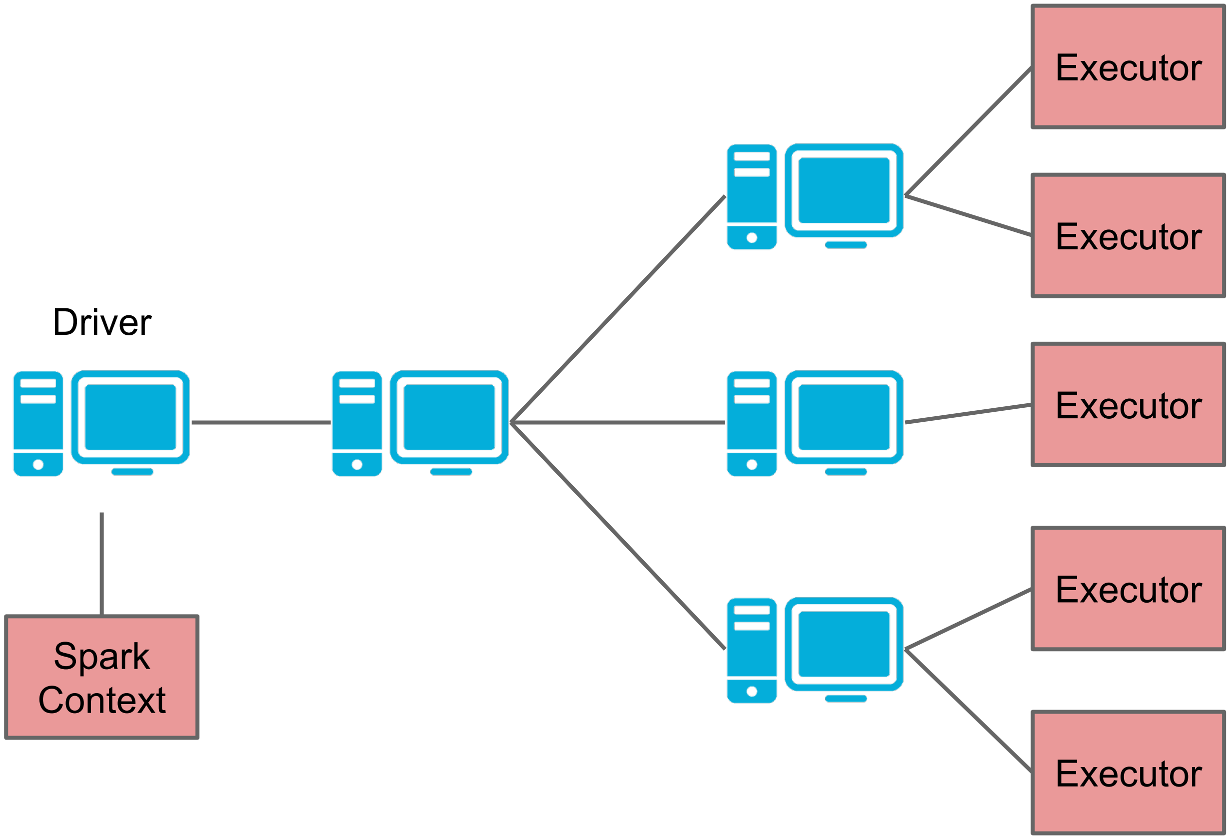


Deployment: Cluster managers

- YARN
  - It uses Hadoop Yarn Resource Manager
  - Spark app is launched inside a Yarn container
  - Several isolated Spark Masters running in cluster possibility.

- Mesos
  - Another cluster manager
  - Not so much used like Yarn with Spark.
  - Several isolated Spark Masters running in cluster possibility.

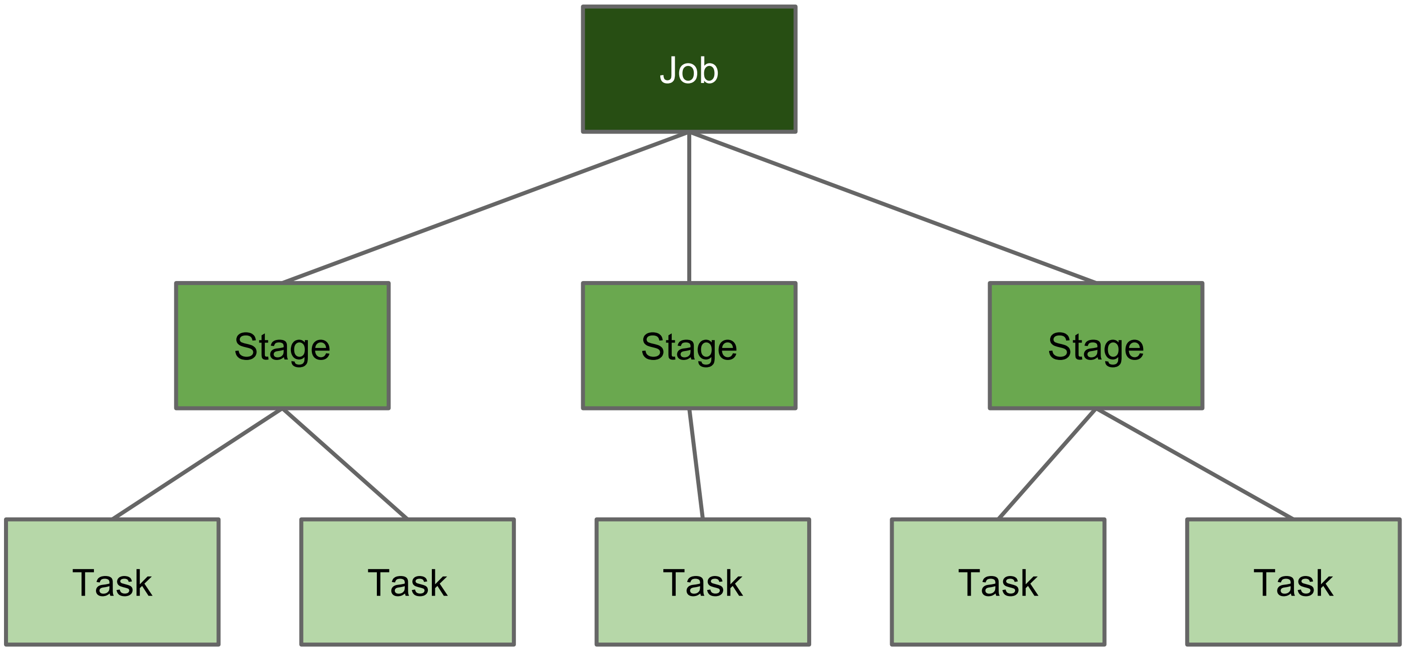
Execution terms



Execution terms : SparkContext & Executor

- SparkContext
  - Spark cluster connection
  - Necessary for SQLContext, StreamingContext, ...
  - Isolated: Two Spark Contexts cannot exchange data (without external help).
- Executor
  - Individual execution unit
  - 1 core ~ 2 executor
  - Each application has its owns.

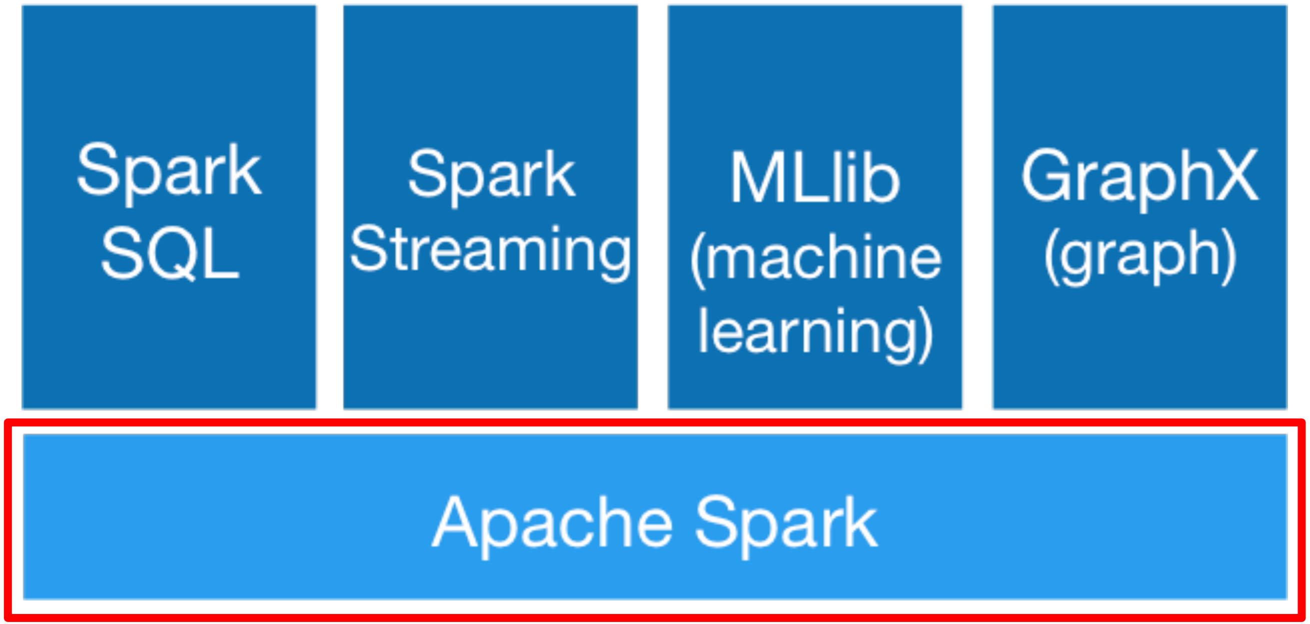
Job terms



Job terms

- Job
  - Individual executed action
  - It’s composed by a set of tasks arranged in stages
- Stage
  - Job split based on last cached action or shuffle event.
- Task
  - Minimum execution unit sent to executors.
  - Task ~ Partition correlation

Modules


Spark SQL
Spark Streaming
MLlib (machine learning)
GraphX (graph)
Apache Spark

RDD

- **R**esilient **D**istributed **D**ataset. Basic abstraction in Spark
- Think of it like a huge collection **partitioned** and distributed in different machines.
- Lazy evaluated
- Basically composed of
  - A list of partitions
  - A function for computing each split
  - A list of dependencies on other RDDs

RDD

- Basic example:

```
val myRdd: RDD[String] = sc.parallelize(
  “this is a sample string”.split(“ “).toList)
```

- For creating a new RDD is necessary to implement:
  - compute (How to get RDD population)
  - getDependencies (RDD lineage)
  - getPartitions (How to split the data)

RDD

- An RDD may be created from
  - A file / set of files:

    ```
    sc.textFile(“myFile”)
    sc.textFile(“file1,file2”)
    ```

  - A bunch of memory-storaged data:

    ```
    sc.parallelize(List(1,2,3))
    ```

    Another RDD:

    ```
    myRdd.map(_.toString)
    ```

RDD - Partitions

- Partition : RDD chunk
- A worker node may contain 1 or more partitions of some RDD.
- It’s important to choose a best-performance partitioning.
- Repartitioning
    - It implies shuffling all RDD data among worker nodes
    - There will surely be tons of network traffic among all worker nodes!

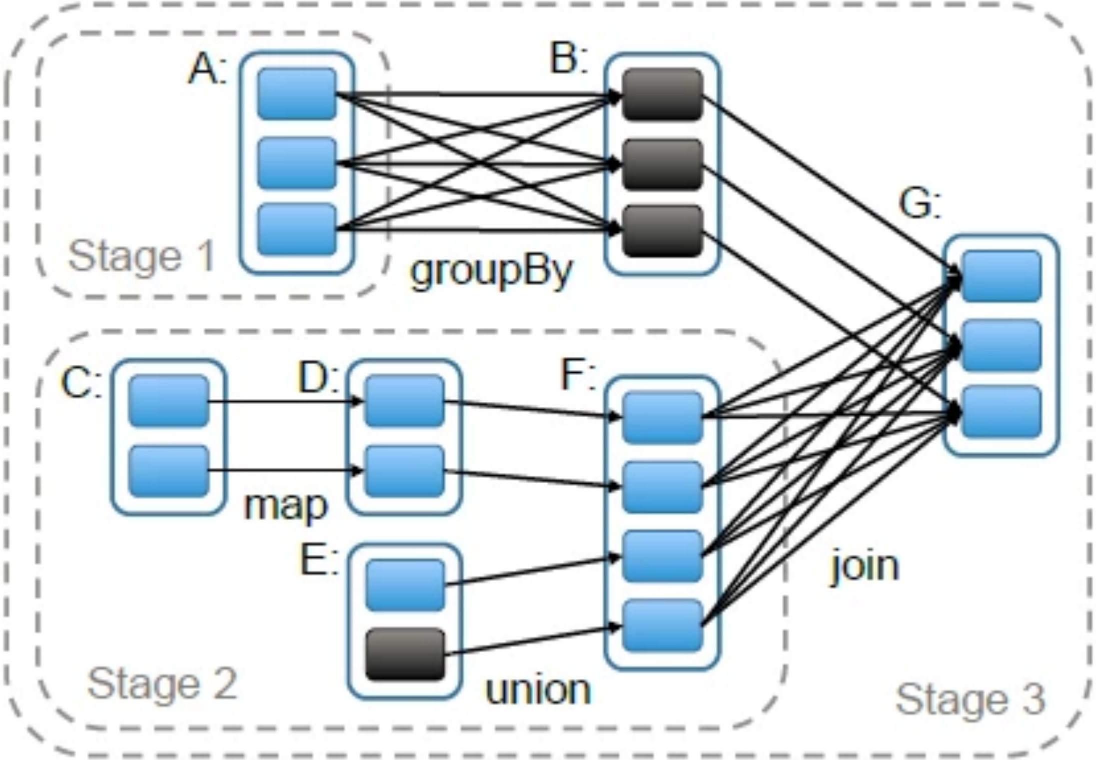
RDD - Lineage

- An RDD may be built from another one.
- Base RDD : Has no parent
- Ex:

```
val base: RDD[Int] = sc.parallelize(List(1,2,3,4))
val even: RDD[Int] = myRDD.filter(_%2==0)
```

- DAG (**D**irected **A**cyclic **G**raph) : Represents the RDD lineage (inheritance, mutations, ...).
- DAGScheduler: High level layer used that schedules stages among all related RDDs

RDD - Lineage



RDD - Transformations

- It applies a change to some RDD, returning a new one.
- It’s not immediately evaluated.
- Think of it like a “Call-by-name” function.
- Adds a new node in Lineage DAG.

RDD - Transformations

- Most frequently used:
    - **map**

```
val myRDD: RDD[Int] = sc.parallelize(List(1,2,3,4))
val myStrinRDD: RDD[String] = myRDD.map(_.toString)
```

    - **flatMap**

```
val myRDD: RDD[Int] =
  sc.parallelize(List(1,null,3,null))
val myNotNullInts: RDD[Int] =
  myRdd.flatMap(n => Option(n))
```

RDD - Transformations

- Most frequently used:
  - **filter**

```
val myRDD: RDD[Int] = sc.parallelize(List(1,2,3,4))
val myStrinRDD: RDD[Int] = myRDD.filter(_%2==0)
```

  - **union**

```
val odd: RDD[Int] = sc.parallelize(List(1,3,5))
val even: RDD[Int] = sc.parallelize(List(2,4,6))
val all: RDD[Int] = odd.union(even)
```

RDD - Actions

- It launches the RDD evaluation,
  - returning some data to the driver
  - or persisting to some external storage system

- It’s evaluated partially on each worker node, and results are merged in the Driver

- There are mechanisms to make cheaper computing several times the same RDD (like `persist()` or `cache()`)

RDD - Actions

- Examples:
  - **count**

    ```
    val myRdd: Rdd[Int] = sc.parallelize(List(1,2,3))
    val size: Int = myRdd.count
    ```

    Counting implies processing whole RDD
  - **take**

    ```
    val myRdd: Rdd[Int] = sc.parallelize(List(1,2,3))
    val List(1,2) = myRdd.take(2).toList
    ```
  - **collect**

    ```
    val myRdd: Rdd[Int] = sc.parallelize(List(1,2,3))
    val data: Array[Int] = myRdd.collect
    ```

    Beware! Executing 'collect' on big collections might end into a memory leak

Demo1 : Most retweeted

- Most Retweeted example
- Bunch of tweets (1000 json records)
- Find out which is the most retweeted tweet.

Key-Value RDDs

- Particular case of RDD[T] where T = (U,V)
- It allows grouping,combining,aggregating values by some key.
- In Scala it's only needed to
  `import org.apache.spark.SparkContext._`
- In Java, it's mandatory to use `PairRDD` class.

Key-Value RDDs

- keyBy: Generates a PairRDD

```
val myRDD: RDD[Int] =
  sc.parallelize(List(1,2,3,4,5))

val kvRDD: RDD[(String,Int)] =
  myRDD.keyBy(
    n => if (n%2==0) “even” else “odd”)
```

“odd” -> 1, “even” -> 2, “odd” -> 3, “even” -> 4, “odd” -> 5

Key-Value RDDs

- keys: Gets keys from PairRDD

```
val myRDD: RDD[(String,Int)] =
  sc.parallelize(“odd” -> 1,”even” -> 2,”odd” -> 3)
val keysRDD: RDD[String] = myRDD.keys
```

“odd”,”even”,”odd”

- values: Gets values from PairRDD

Key-Value RDDs

- mapValues: map PairRDD values, omitting keys.

```
val myRDD: RDD[(String,Int)] =
  sc.parallelize("odd" -> 1,"even" -> 2,"odd" -> 3)
val mapRDD: RDD[(String,String)] =
  myRDD.mapValues(_.toString)
```

"odd" -> "1", "even" -> "2", "odd" -> "3"

- flatMapValues: flatMap PairRDD values

Key-Value RDDs

- join: Return a new RDD with both RDD joined by key.

```
val a = sc.parallelize(List("dog", "salmon", "salmon", "rat", "elephant"), 3)
val b = a.keyBy(_.length)
val c = sc.parallelize(List("dog","cat","gnu","salmon","rabbit","turkey","wolf","
  bear","bee"), 3)
val d = c.keyBy(_.length)
b.join(d).collect
```

```
res0: Array[(Int, (String, String))] = Array((6,(salmon,salmon)), (6,(salmon,rabbit)),
(6,(salmon,turkey)), (6,(salmon,salmon)), (6,(salmon,rabbit)), (6,(salmon,turkey)), (3,
(dog,dog)), (3,(dog,cat)), (3,(dog,gnu)), (3,(dog,bee)), (3,(rat,dog)), (3,(rat,cat)),
(3,(rat,gnu)), (3,(rat,bee)))
```

Key-Value RDDs - combineByKey

- combineByKey:

```
def combineByKey[C](
    createCombiner: V => C,
    mergeValue: (C,V) => C,
    mergeCombiners: (C,C) => C): RDD[(K,C)]
```

- Think of it as *something-like-but-not* a `foldLeft` over each partition

Key-Value RDDs - combineByKey

- It’s composed by:
  - `createCombiner( V => C)`: Sets the way to mutate initial RDD [V] data into new data type used for aggregating values (C). This will be called Combinator.
  - `mergeValue((C,V) => C)`: Defines how to aggregate initial V values to our Combiner type C, returning a new combiner type C.
  - `mergeCombiners((C,C) => C)`: Defines how to merge two combiners into a new one.

Key-Value RDDs - combineByKey

- Example:

```
val a = sc.parallelize(List("dog","cat","gnu","salmon","rabbit","
  turkey","wolf","bear","bee"), 3)
val b = sc.parallelize(List(1,1,2,2,2,1,2,2,2), 3)
val c = b.zip(a)
val d = c.combineByKey(List(_), (x:List[String], y:String) => y :: x,
  (x:List[String], y:List[String]) => x ::: y)
d.collect
```

res16: Array[(Int, List[String])] = Array((1,List(cat, dog, turkey)), (2,List(gnu, rabbit, salmon, bee, bear, wolf)))

Key-Value RDDs - aggregateByKey

- **aggregateByKey** : Aggregate the values of each key, using given combine functions and a neutral “zero value”.
- *Beware*! Zero value is evaluated in each partition.

```
def aggregateByKey[U:ClassTag](zeroValue: U)(
  seqOp: (U,V) => U, combOp: (U,U) => U): RDD[(K,U)] =
  combineByKey(
    (v: V) => seqOp(zeroValue,v),
    seqOp,
    comboOp)
```

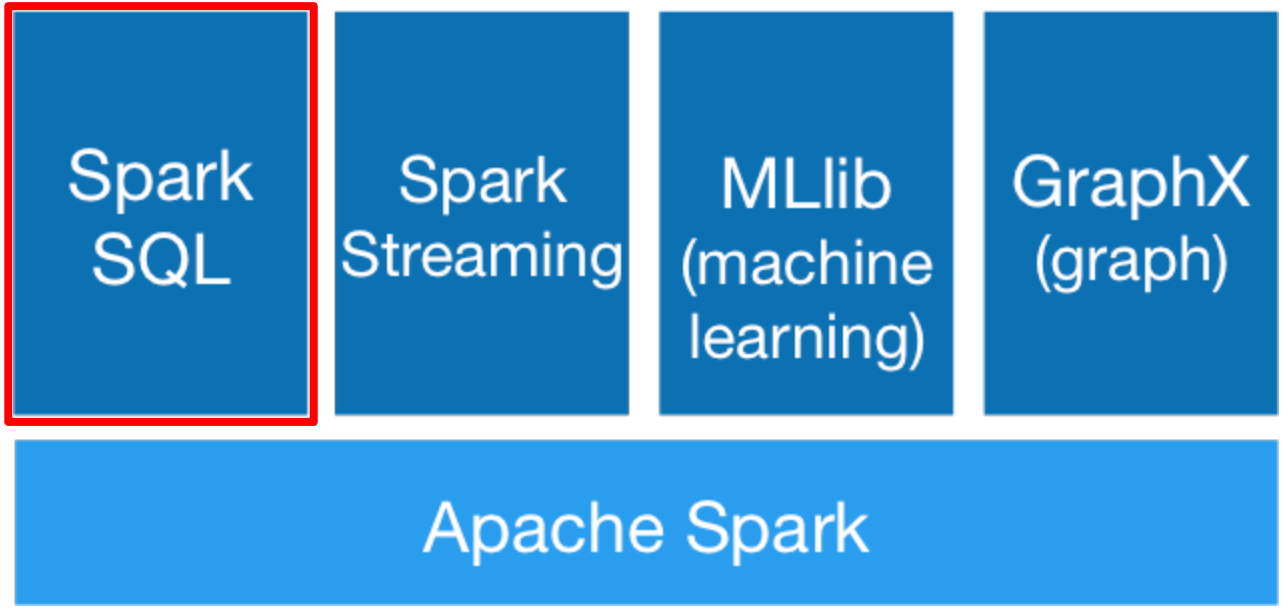
Key-Value RDDs - groupByKey

- **groupByKey**: Group all values with same key into a Iterable of values

```
def groupByKey(): RDD[(Key,Iterable[Value])] =
  combineByKey[List[Value]](
    (v: V) => List(v),
    (list: List[V],v: V) => list+= v,
    (l1: List[V],l2: List[V]) => l1 ++ l2)
```

Note: GroupByKey actually uses CompactBuffer instead of list.

Modules


Spark SQL
Spark Streaming
MLlib (machine learning)
GraphX (graph)
Apache Spark


INVENTS MAPREDUCE
DOESN'T SHARE HIGH-LEVEL LANGUAGE SUPPORT
SCUMBAGSTEVE.COM

SparkSQL

- Shark successor (Berkeley)
  - Alternative to Hive + Hadoop
- Query language
  - HiveQL
  - Future: Support full SQL92
- SQLContext

```
val sqlContext = new SQLContext(sparkContext)
val hiveContext = new HiveContext(sparkContext)
```

SparkSQL

- Different datasources
  - JSON
  - Parquet
  - CSV
- New implementations
  - Cassandra
  - ElasticSearch
  - MongoDB
- Unified API in Spark 1.3.0

```
val students: DataFrame = sqlContext.load(
  “students”,“org.apache.sql.parquet”,Map(...))
```

SparkSQL - DataFrame

- DataFrame = RDD[org.apache.spark.sql.Row] + Schema
- A Row holds both column values and their types.
- This allows unifying multiple datasources with the same API.
  i.e, we could join two tables, one declared on Mongo and another on ElasticSearch.

Demo2 : Most retweeted(SparkSQL)

- Most Retweeted Example (SparkSQL)
- Same bunch of tweets (1000 json records)
- Find out which is the most retweeted tweet using SparkSQL

Who uses it?

amazon

databricks

YAHOO!

ebay

Almaden Institute

IBM

Nokia Networks

JPL

NASA

National Aeronautics and Space Administration
Jet Propulsion Laboratory
California Institute of Technology

STRATIO

cloudera®

TREND MICRO™

Simba technologies

Distributed computing with

Spark

Javier Santos

April 2015