Introduction to Apache Spark

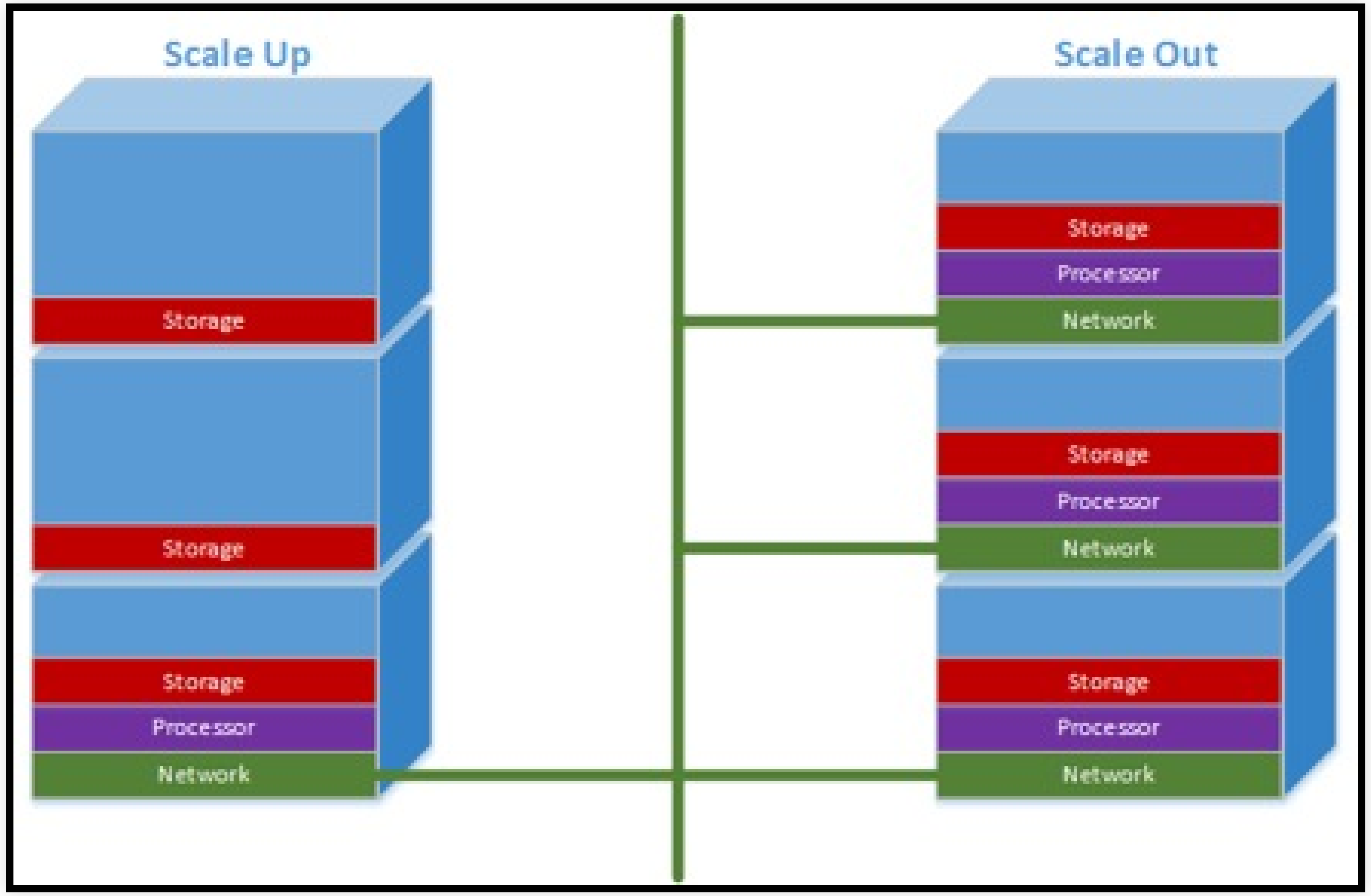
Valentin Bergeron, LesFurets.com, tp-bigdata@lesfurets.com

Plan

Plan

- Distributed programming models
    - Data flow models (*MapReduce*, ***Apache Spark***, *Apache Flink*, *Apache Beam*, *Google Dataflow*)

Scale-Up / Scale-Out



- ***Scale-Up***: "single powerfull computer"($$$)
- ***Scale-Out***: network of commodity hardware

Distributed programming

- unreliable ***network***
- hardware/software ***failures***
- synchronous ⇒ **asynchronous**
- ***consistency and ordering*** are expensive
- ***time***

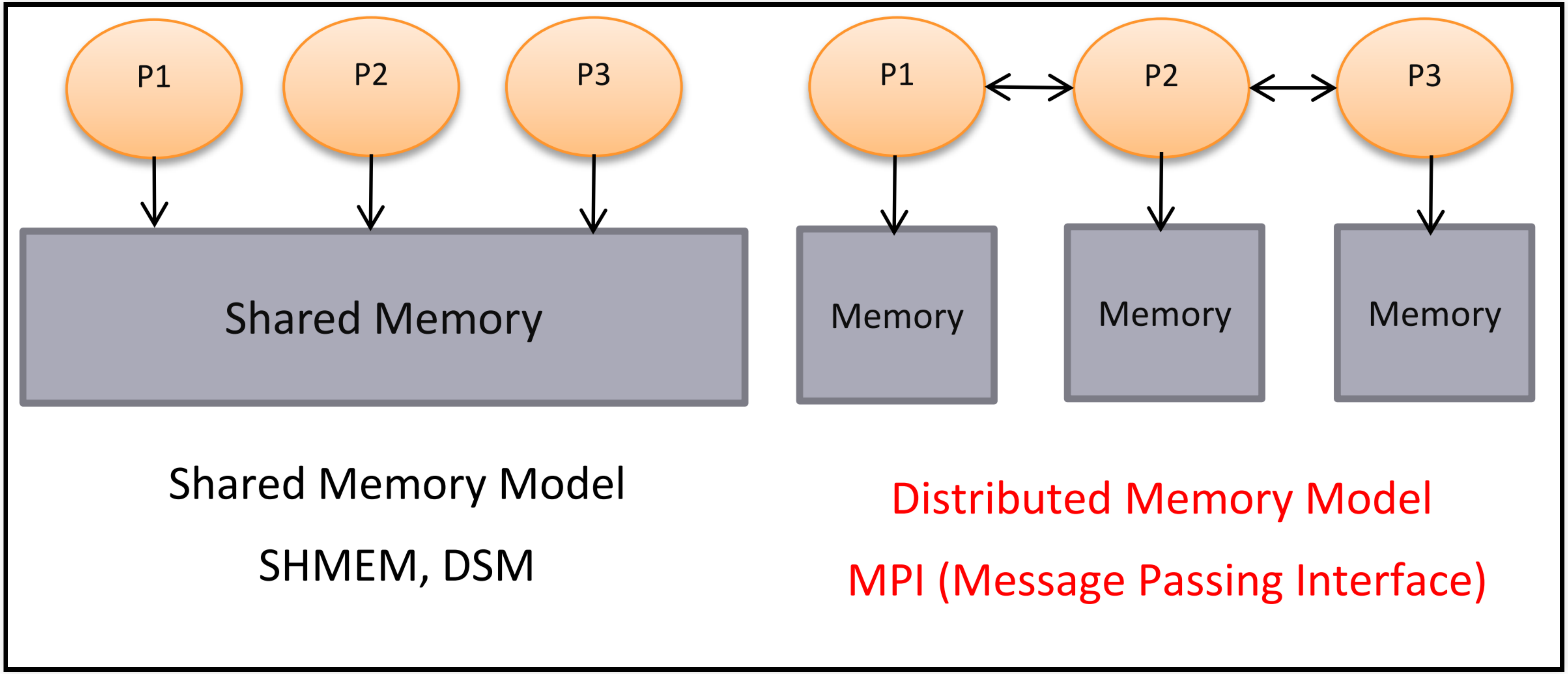
(Distributed) Databases vs Distributed programming

- ***Databases:*** used as a shared ressource
  - data ***persistence***
  - standardized access to data (API / ***query language***)
  - ***guaranteed properties (CAP)*** via *sharding and replication*
- ***Distributed programming:***
  - express ***general computation***
  - *pipelines of tasks*

Databases vs Distributed programming

- ***More and more difficult to distinguish between them !***
- **Distributed Databases:**
  - *User Defines Types, User Defined Functions, MapReduce support*
- ***Distributed programming frameworks:***
  - ***SQL*** like access (SparkSQL, Dataframes on top of RDDs), ***DSLs*** as query languages, DB inspired ***optimizations***
  - Data ***persistence***, ***partitioning and replication*** (MEMORY_ONLY_2)

Distributed programming models



Shared Memory Model

SHMEM, DSM

Distributed Memory Model

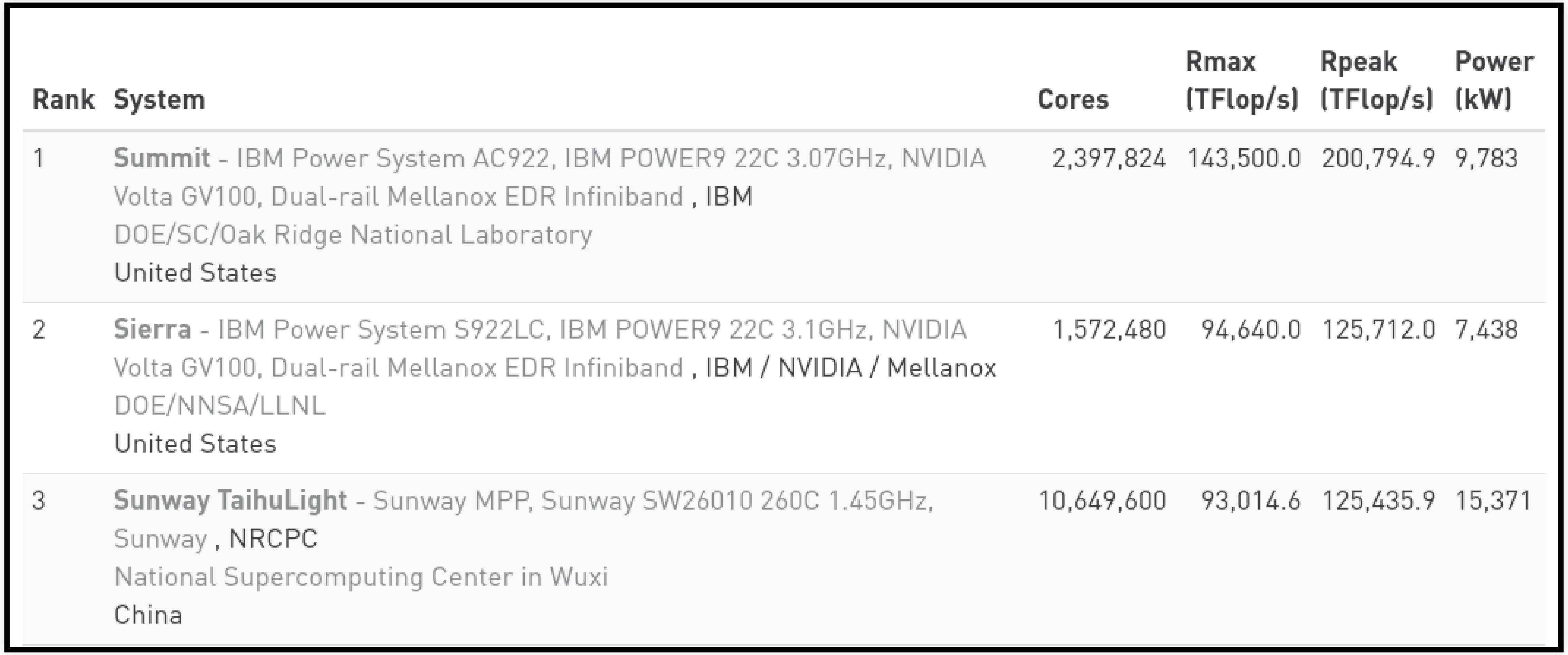
MPI (Message Passing Interface)

Shared Memory / Shared Nothing

- ***Von Neuman architecture*** ⇒ *instructions executed sequentially by a worker (CPU) and data does not move*
- ***Shared Memory***
  - *multiple workers* (CPU/Cores) executing computations in *parallel*
  - use **locks, semaphores and monitors** to synchronise access to the shared memory
- ***Shared Nothing: Divide&Conquer*** (*ForkJoin/ScatterGather*)
  - ***distribute data***
  - ***distribute the computations*** (**the code**)
  - ***launch and manage parallel computations***
  - ***collect*** results

Shared Nothing hegemony

Since 2006, no Shared-Memory system in the first 10 places on TOP500



Rank	System	Cores	Rmax (TFlop/s)	Rpeak (TFlop/s)	Power (kW)
1	**Summit** - IBM Power System AC922, IBM POWER9 22C 3.07GHz, NVIDIA Volta GV100, Dual-rail Mellanox EDR Infiniband , IBM DOE/SC/Oak Ridge National Laboratory United States	2,397,824	143,500.0	200,794.9	9,783
2	**Sierra** - IBM Power System S922LC, IBM POWER9 22C 3.1GHz, NVIDIA Volta GV100, Dual-rail Mellanox EDR Infiniband , IBM / NVIDIA / Mellanox DOE/NNSA/LLNL United States	1,572,480	94,640.0	125,712.0	7,438
3	**Sunway TaihuLight** - Sunway MPP, Sunway SW26010 260C 1.45GHz, Sunway , NRCPC National Supercomputing Center in Wuxi China	10,649,600	93,014.6	125,435.9	15,371

Share Nothing Architectures

1. ***Message Passing*** *(MPI, Actors, CSP)*
2. ***DataFlow systems***

Share Nothing Architectures → Message Passing

Computing grid : *MPI (message passing interface)* (1993-2012)

- launch the same code on multiple nodes

```
mpirun -np 4 simple1
```

```
// determining who am I
int myrank;
MPI_Comm_rank(MPI_COMM_WORLD, &myrank); (1)
// determining who are they :
int nprocs;
MPI_Comm_size(MPI_COMM_WORLD, &nprocs);  (2)
// sending messages
MPI_Send(buffer, count, datatype, destination, tag, MPI_COMM_WORLD); (3)
//receiving Messages
MPI_Recv(buffer, maxcount, datatype, source, tag, MPI_COMM_WORLD, &status); (4)
```

”Traditional” distributed programming

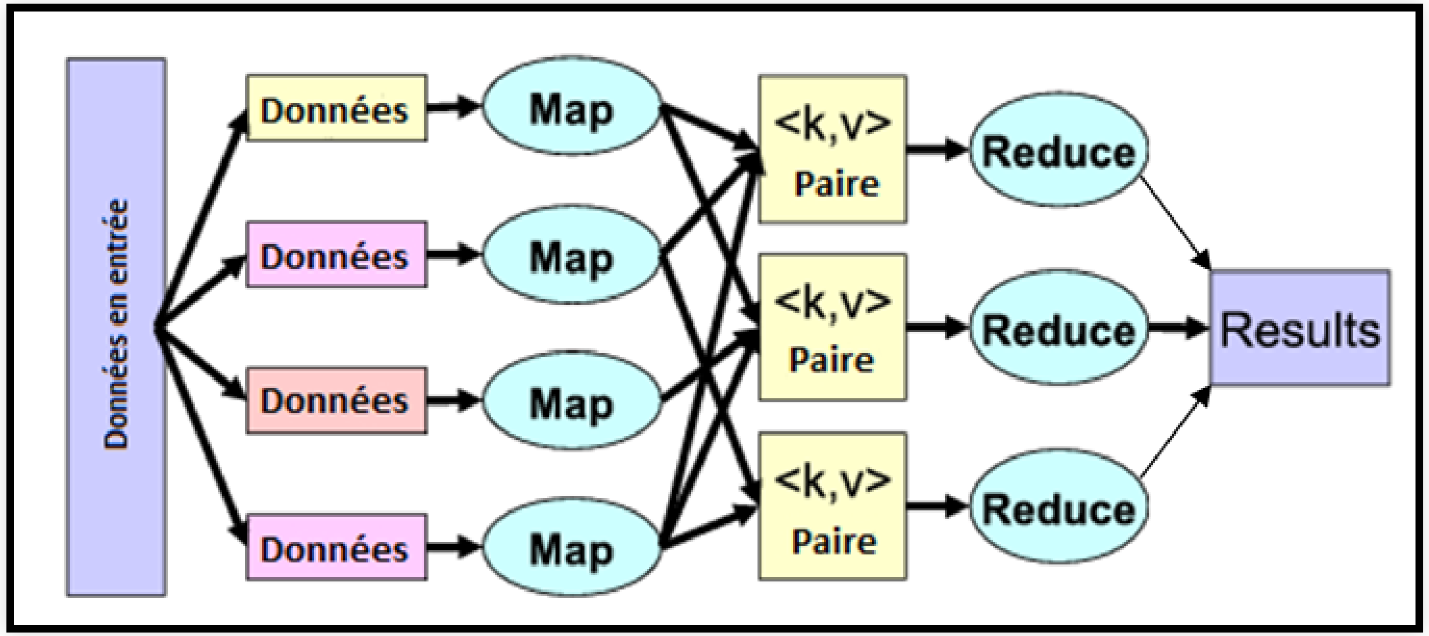
- ***Low level***
  - manual data decomposition
  - manual load balancing
- ***difficult to do at scale***:
  - how to efficiently split data/code across nodes?
    - must consider network & data locality
  - how to deal with failures? (inevitable at scale)

The Present: Data flow models

Restrict the programming interface so that the **system** can do more automatically

- Express jobs as **graphs of high-level operators**
  - **System** picks how to *split each operator into tasks* and *where to run each task*
  - Tasks executed on **workers** as soon as soon as input available

Data flow models: Map-Reduce

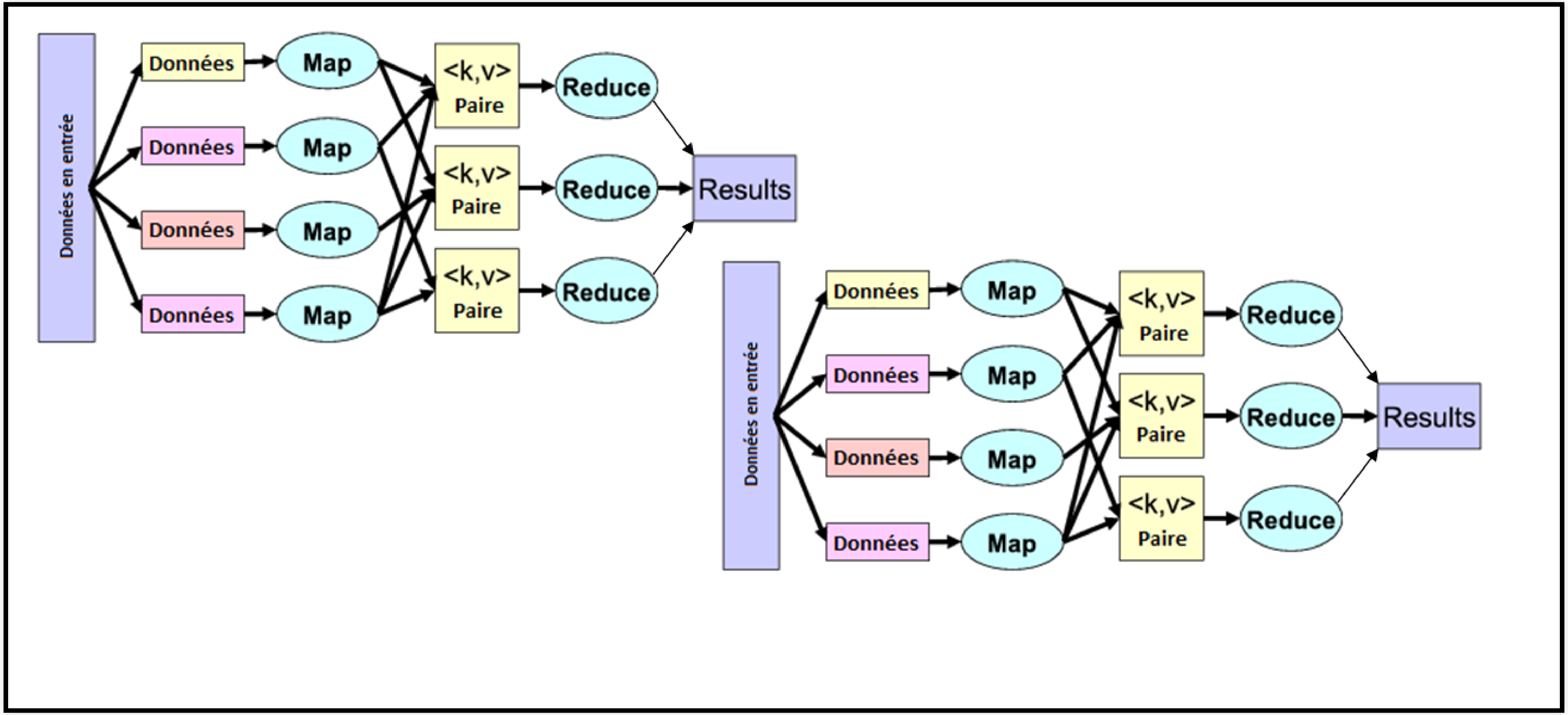


© http://commons.wikimedia.org/wiki/File:Mapreduce.png

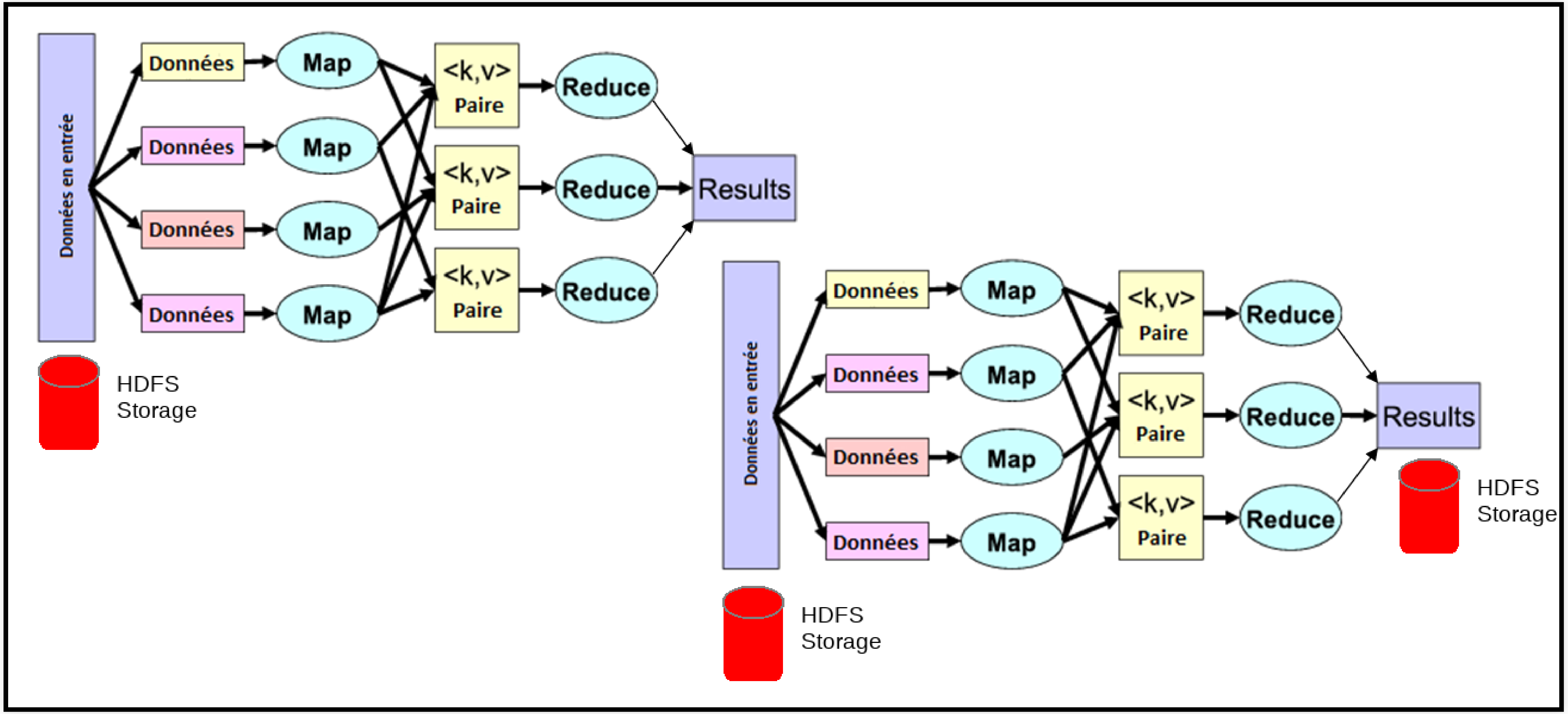
Map Reduce: **strong points**

- ***Simple*** programming model
  - High-level functions instead of message passing
- ***Scalability*** to very largest clusters
  - very good performance for simple jobs (one pass algorithms)
    - Run parts twice fault recovery

Map Reduce iterations



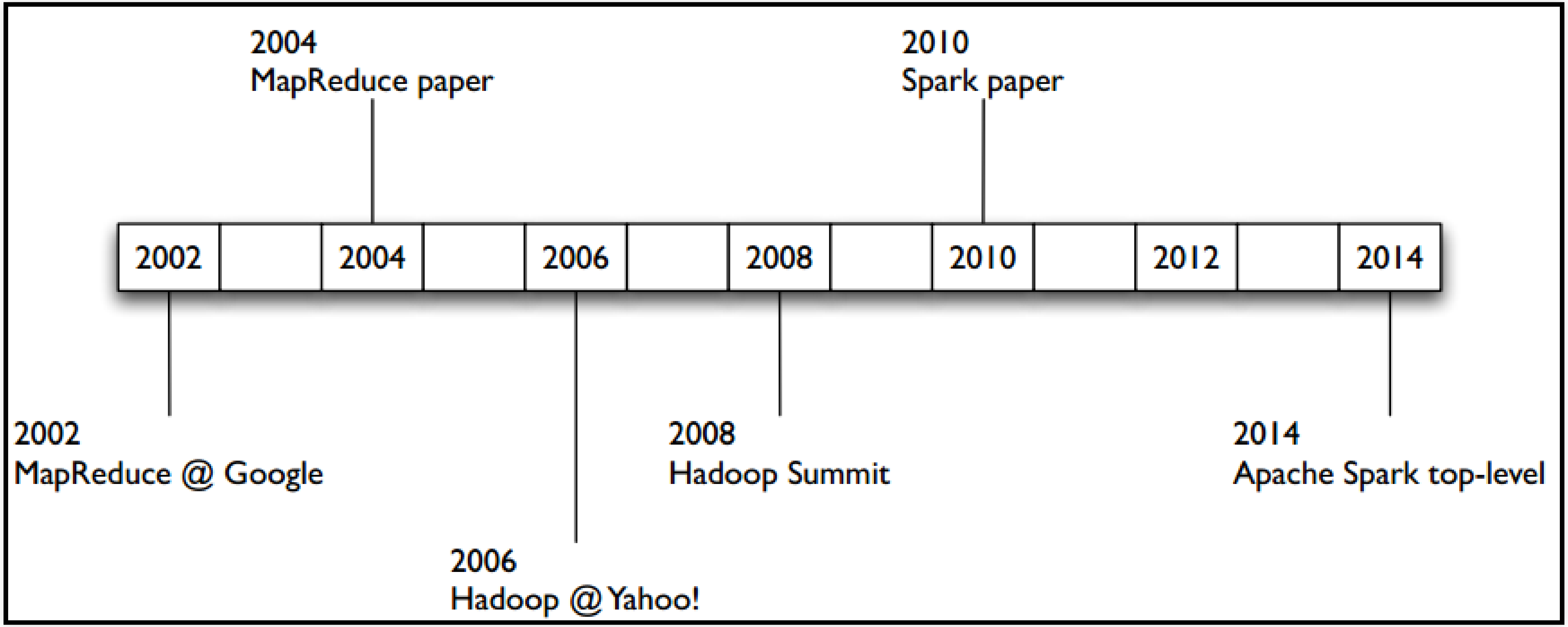
Map Reduce iterations



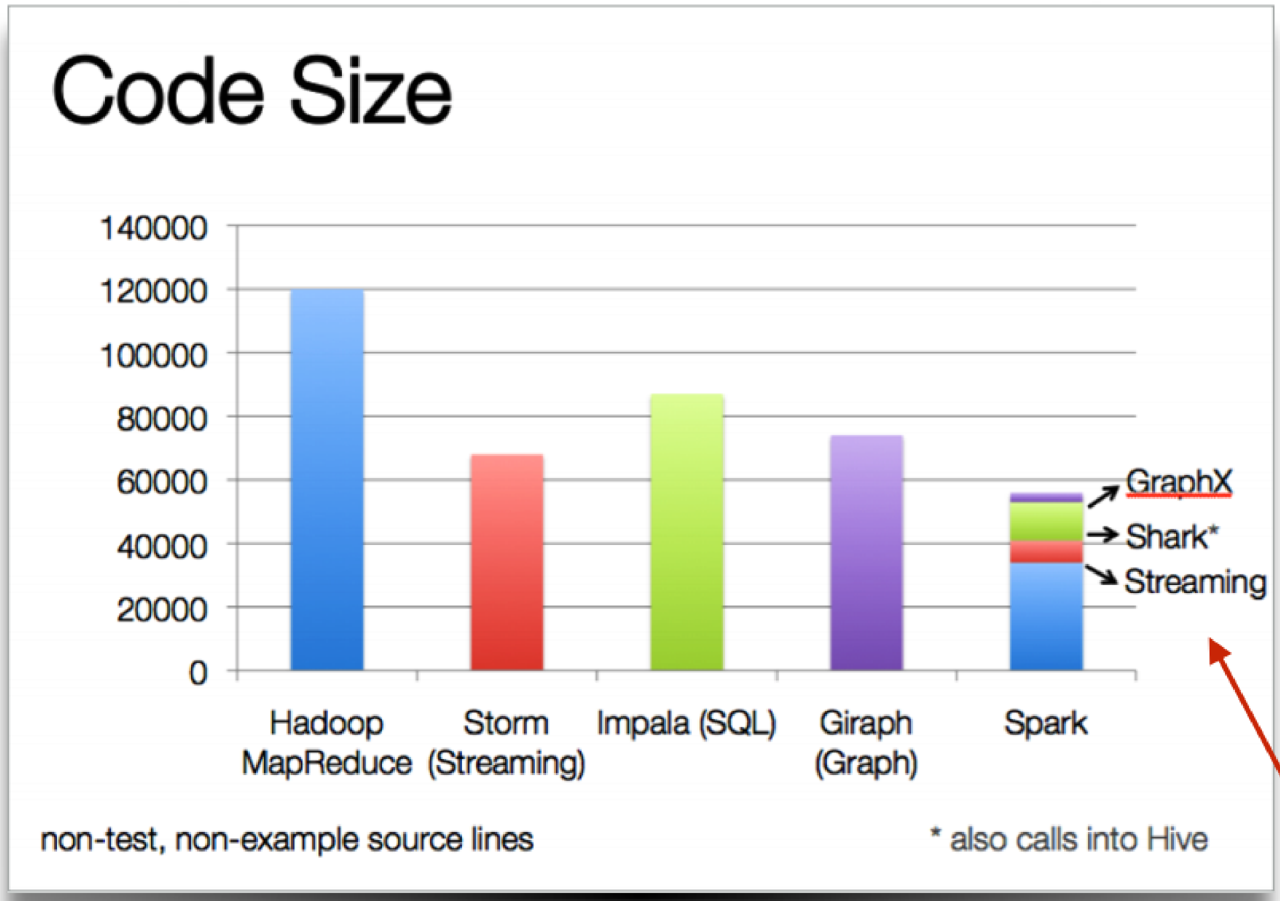
Map Reduce: **weak points:**

- not apropriate for ***multi-pass /iterative algorithms***
  - no efficient primitives for ***data sharing***
  - *state* between steps goes to distributed ***file system***
- ***cumbersome to write***

Apache Spark

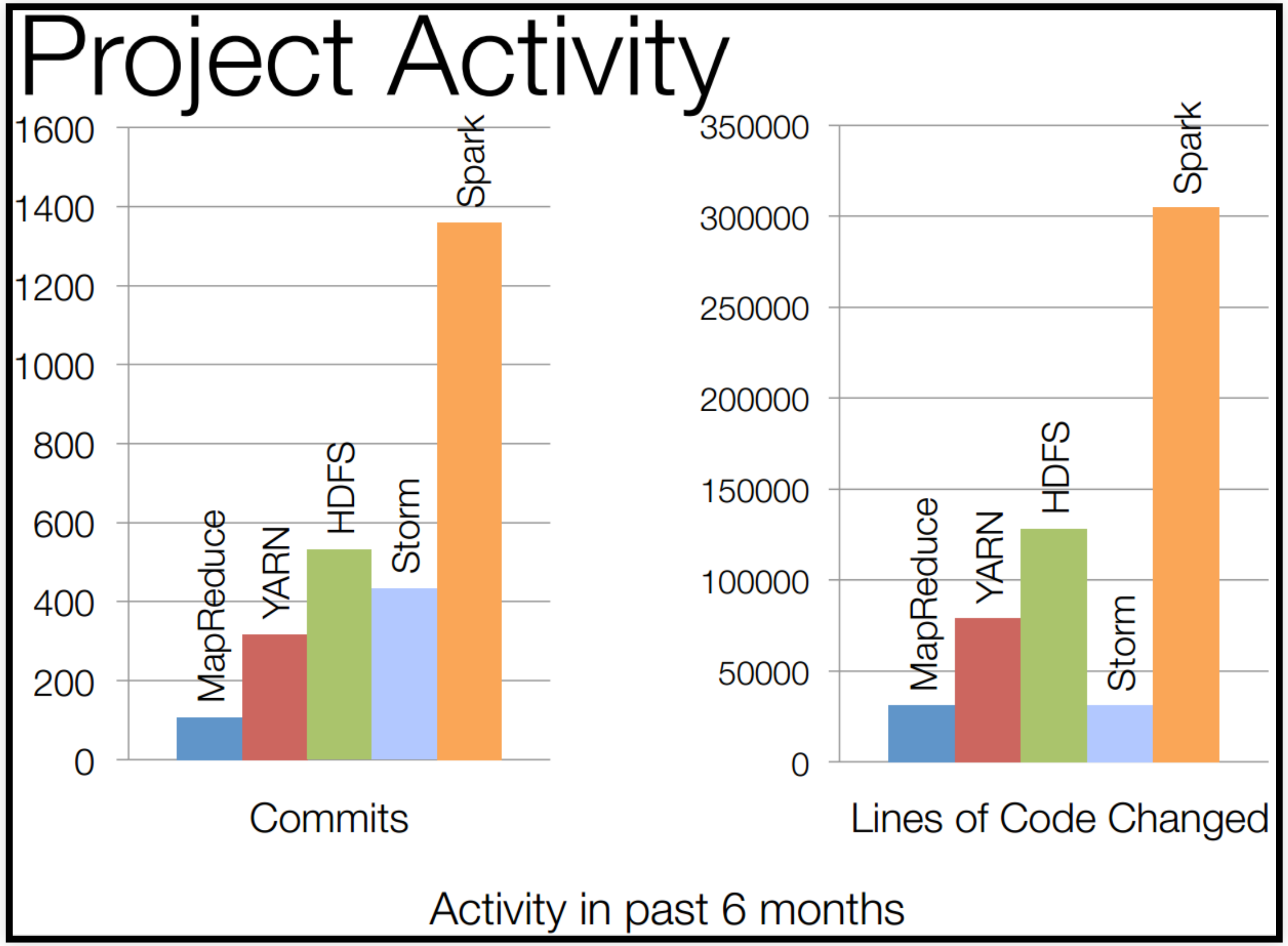


Spark code size (2015)

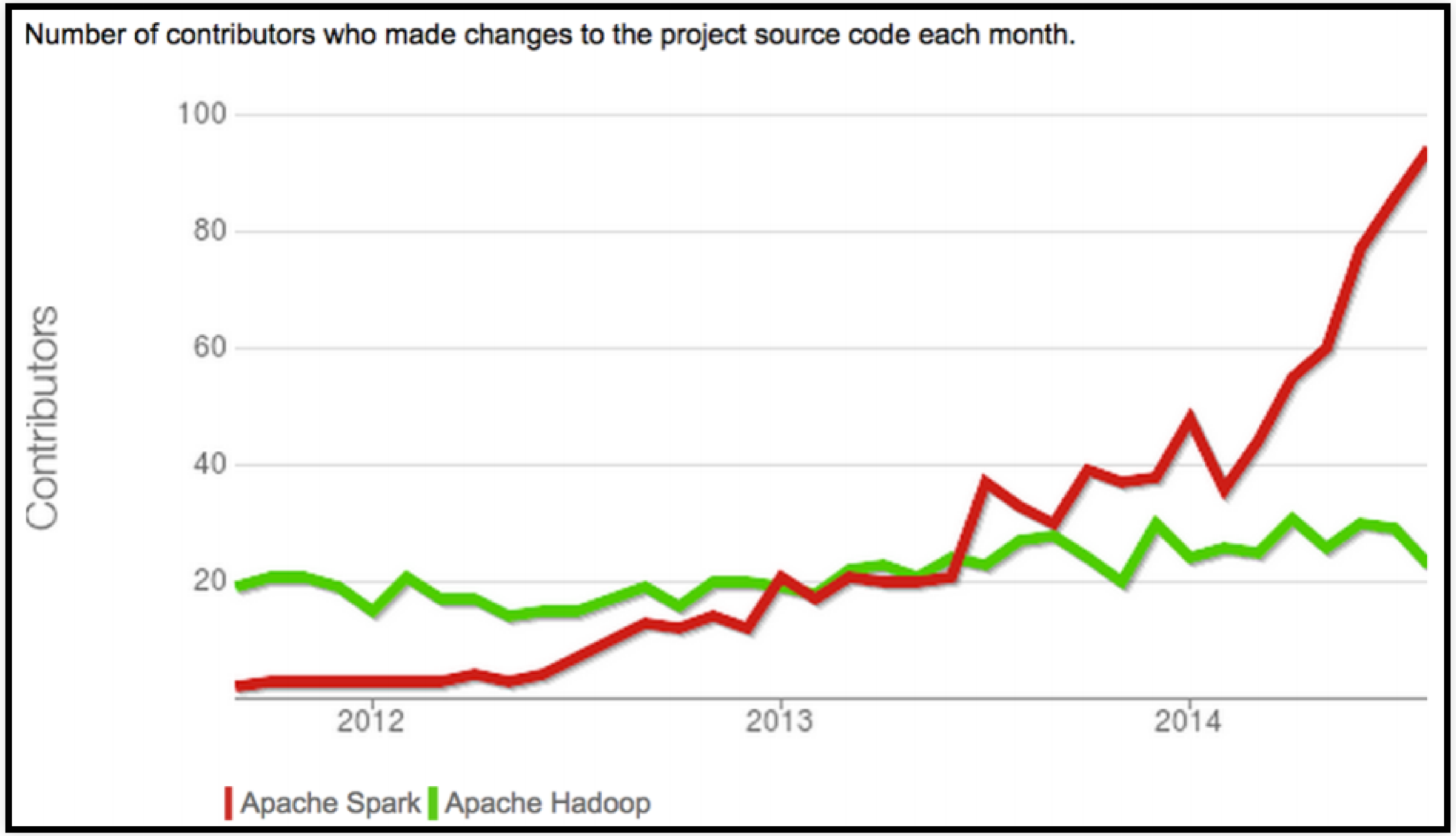


used as libs, instead of specialized systems

Spark project activity (2015)

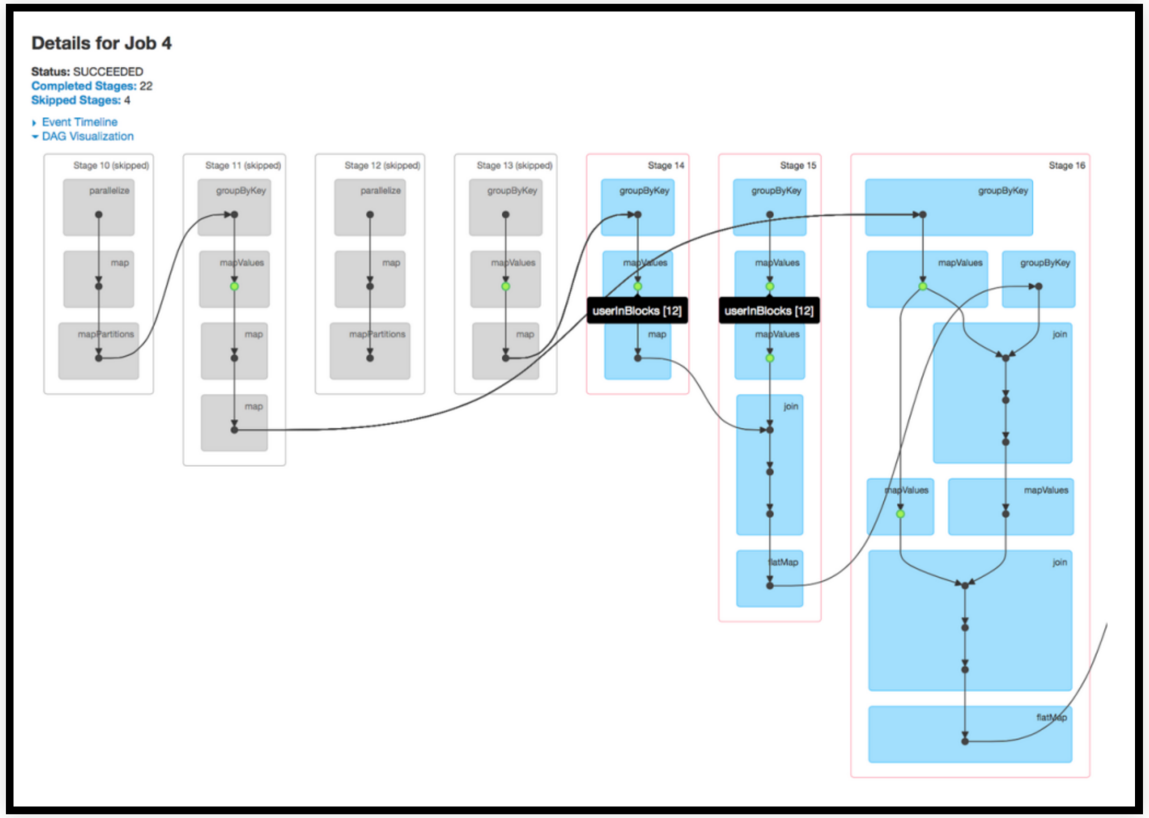


Spark contributors



Apache Spark: general dataflow model

- express general computations in a DAG form
- using a simple distributed list paradigm



Details for Job 4

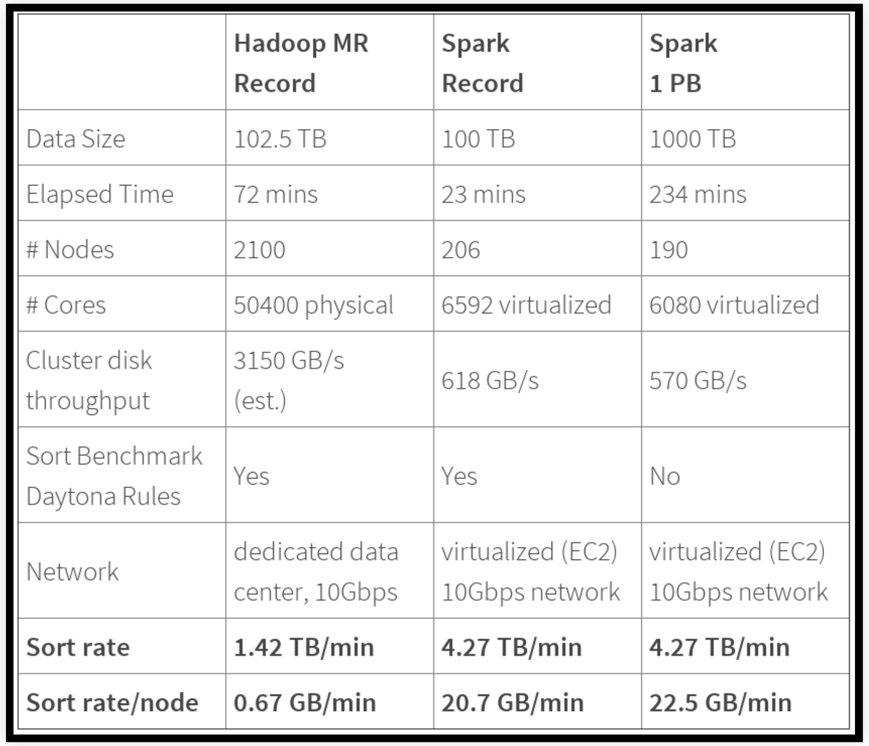
Status: SUCCEEDED
Completed Stages: 22
Skipped Stages: 4

▸ Event Timeline
▾ DAG Visualization

Spark vs MapReduce

- ***Performance***:
  - *RDDs* ***in memory*** *Resilient Distributed Datasets*
  - custom *caching* on nodes
  - *lazy evaluation* of the lineage graph ⇒ reduces wait states, better pipelining
  - generational differences in hardware ⇒ off-heap use of large memory spaces
  - lower overhead for starting jobs / less expensive shuffles
- ***Expressiveness***:
  - generalized *patterns* ⇒ unified engine for many use cases
  - *functional programming* ⇒ maintenance cost for large apps

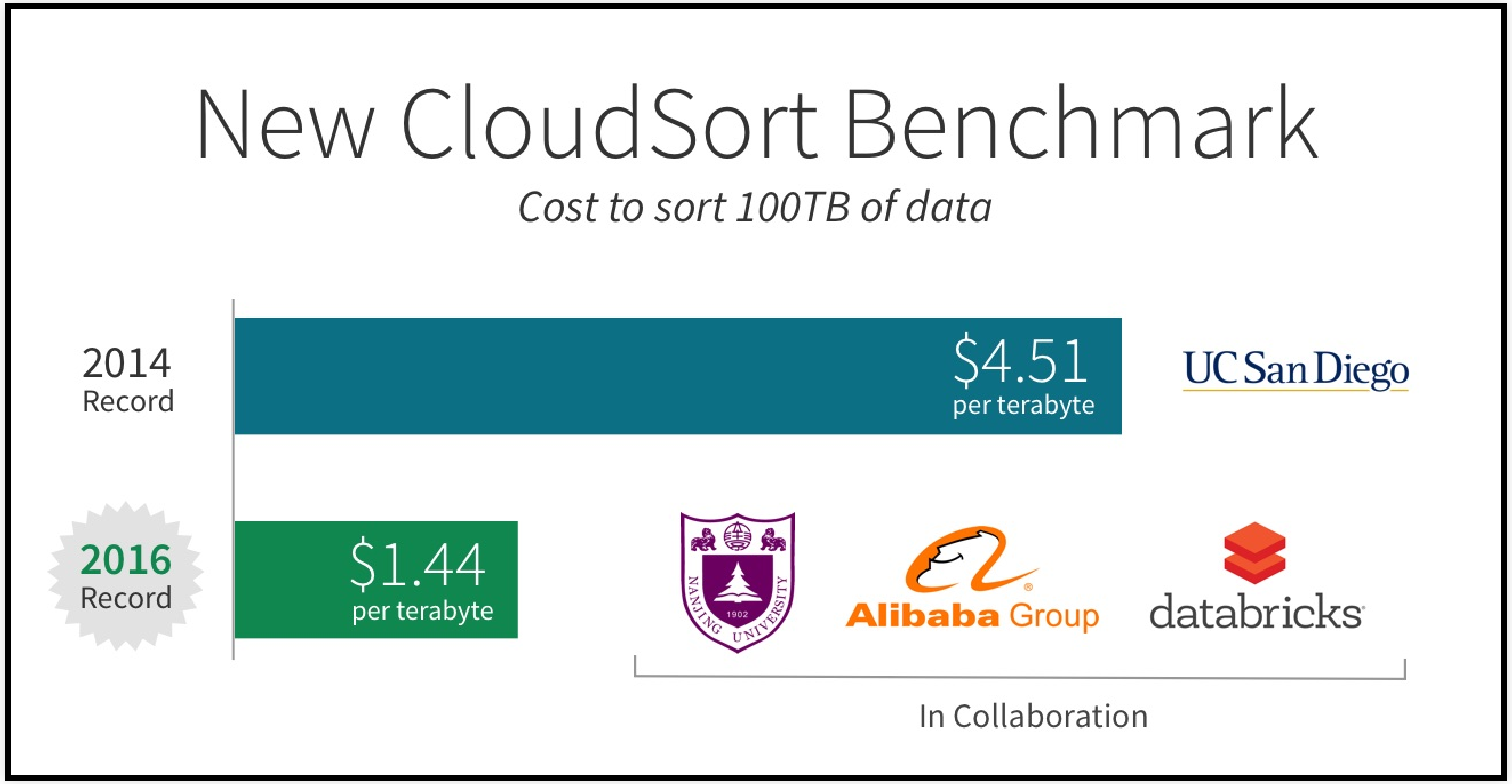
Performance: 3X faster using 10X fewer machines



	Hadoop MR Record	**Spark Record**	**Spark 1 PB**
Data Size	102.5 TB	100 TB	1000 TB
Elapsed Time	72 mins	23 mins	234 mins
# Nodes	2100	206	190
# Cores	50400 physical	6592 virtualized	6080 virtualized
Cluster disk throughput	3150 GB/s (est.)	618 GB/s	570 GB/s
Sort Benchmark Daytona Rules	Yes	Yes	No
Network	dedicated data center, 10Gbps	virtualized (EC2) 10Gbps network	virtualized (EC2) 10Gbps network
Sort rate	**1.42 TB/min**	**4.27 TB/min**	**4.27 TB/min**
Sort rate/node	**0.67 GB/min**	**20.7 GB/min**	**22.5 GB/min**

Spark officially sets a new record in large scale sorting

Performance vs cost



https://databricks.com/blog/2016/11/14/setting-new-world-record-apache-spark.html

Plan

1. Context: Distributed programming
2. **Spark data model**
    1. RDDs
3. Spark execution model
4. Spark cluster
5. Spark ecosystem

Spark Idea

Distributed programming should be no different than standard programming

```
val data = 1 to 1000
val filteredData= data.filter( _%2==0)
```

Spark Idea

Distributed programming should be no different than standard programming

```
val data = 1 to 1000
val filteredData = sc.parallelize(data).filter(_%2==0) 1 2
filteredData.collect 3
```

1	distribute data to nodes
2	distributed filtering
3	collect the result from nodes

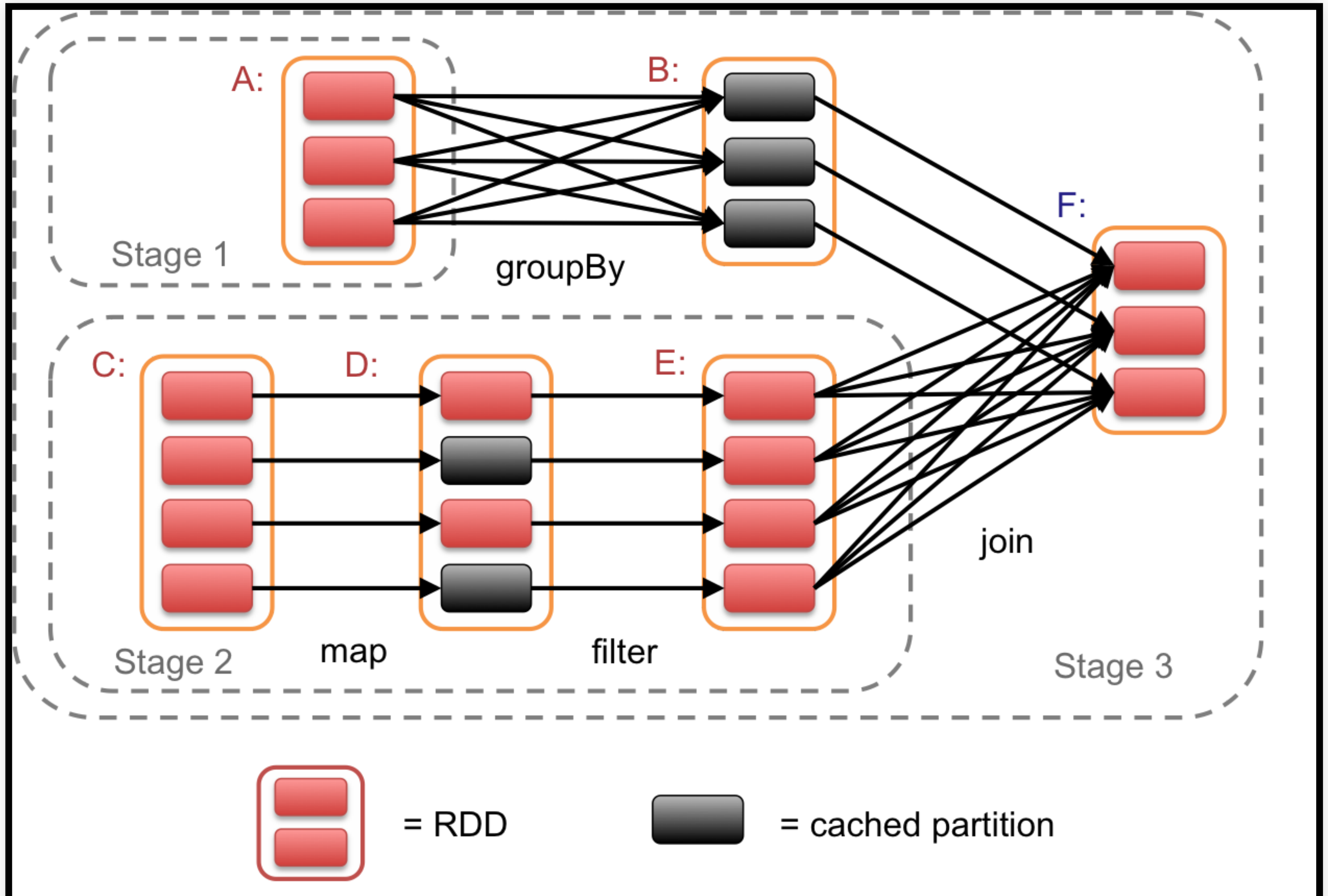
Resilient Distributed Datasets (RDDs)

- Collections of objects across a cluster with user controlled partitioning & storage (memory, disk, …)

Building RDDs

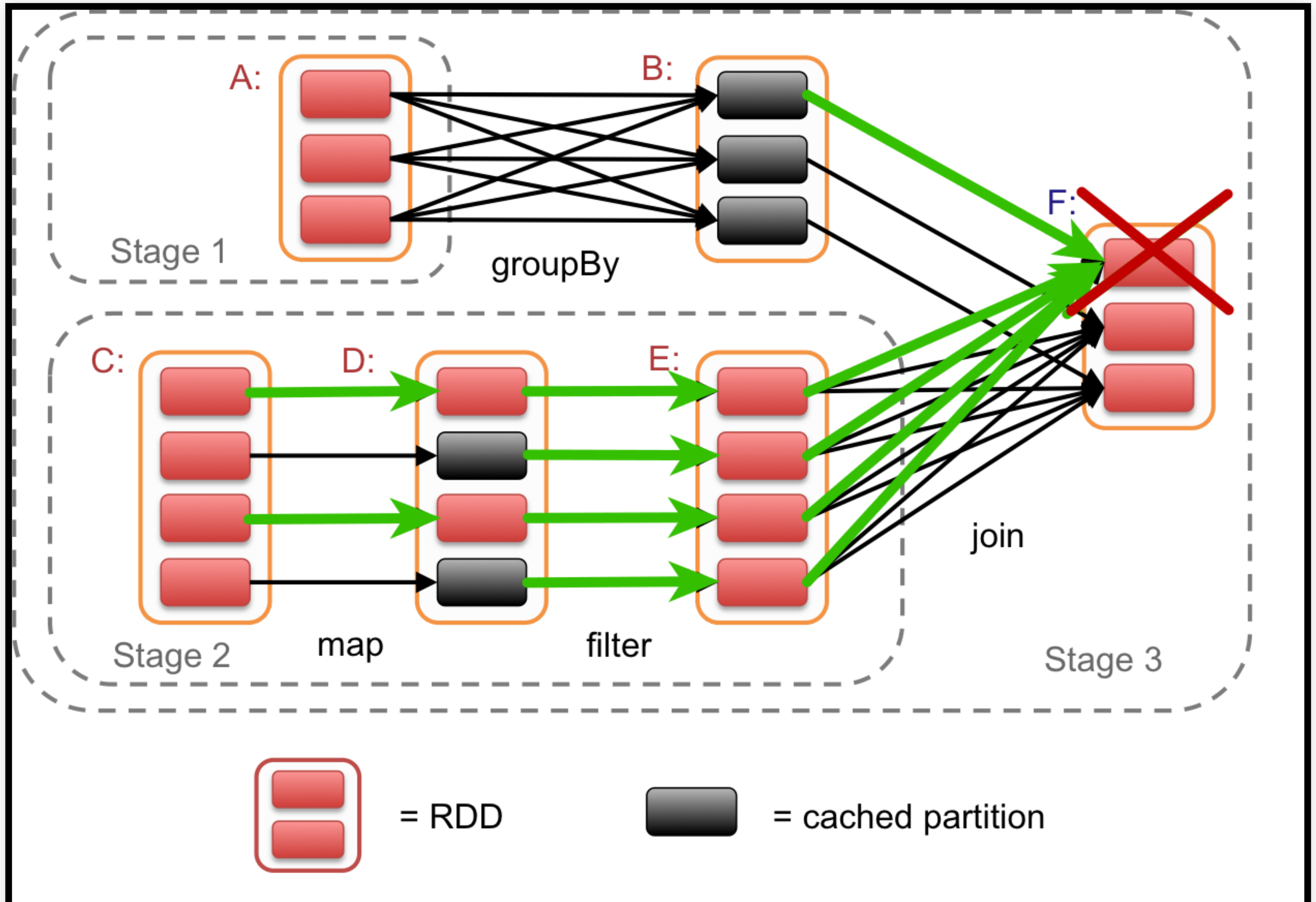
- Built via :
  - paralelization of data (**sc.parallelize**)
  - reading from parallel data sources (hdfs, s3, Cassandra)
  - parallel transformations of other RDDs(map, filter, …)
- Automatically rebuilt on failure (lineage)

Resilient Distributed Datasets (RDDs)



Source: Matei Zaharia slide deck on Spark at Strata conference Feb 2013

Resilient Distributed Datasets (RDDs)



Source: Matei Zaharia slide deck on Spark at Strata conference Feb 2013

Plan

1. Distributed programming
2. Spark data model
3. **Spark programming**
4. Spark execution model
5. Spark ecosystem: SparkSQL et Cassandra

First program in Spark

```
val data = 1 to 1000 (1)
val firstRDD = sc.parallelize(data) (2)
val secondRDD = firstRDD.filter( _ < 10) (3)
secondRDD.collect (4)
```

First program in Spark

```
val data = 1 to 1000 ①
```

① Local data generation

First program in Spark

```
val firstRDD = sc.parallelize(data) (1)
```

(1) Dispatch

```
def parallelize[T](seq: Seq[T], numSlices: Int): rdd.RDD[T]
```

First program in Spark

- (distributed) filtering

```
val secondRDD = firstRDD.filter( _ < 10)
```

First program in Spark

- collect the results

```
secondRDD.collect 4
```

Spark Word Count

```
val input = sc.textFile("s3://...") ①
val words = input.flatMap(x => x.split(" ")) ②
val result = words.map(x => (x, 1))③
                  .reduceByKey((x, y) => x + y) ④
```

1	construit un input RDD pour lire le contenu d'un fichier depuis AWS S3
2	transformer le RDD input pour construir un RDD qui contient tous les mots du fichier
3	construit un RDD qui contient des paires (mots,1)
4	construit un RDD qui fait le comptage des occurences

Spark Word Count - optimisation

```
val input = sc.textFile("s3://...") 1
val result = lines.flatMap(x => x.split(" ")).countByValue() 2
```

1	construit un input RDD pour lire le contenu d’un fichier depuis AWS S3
2	utilisation de countByValue

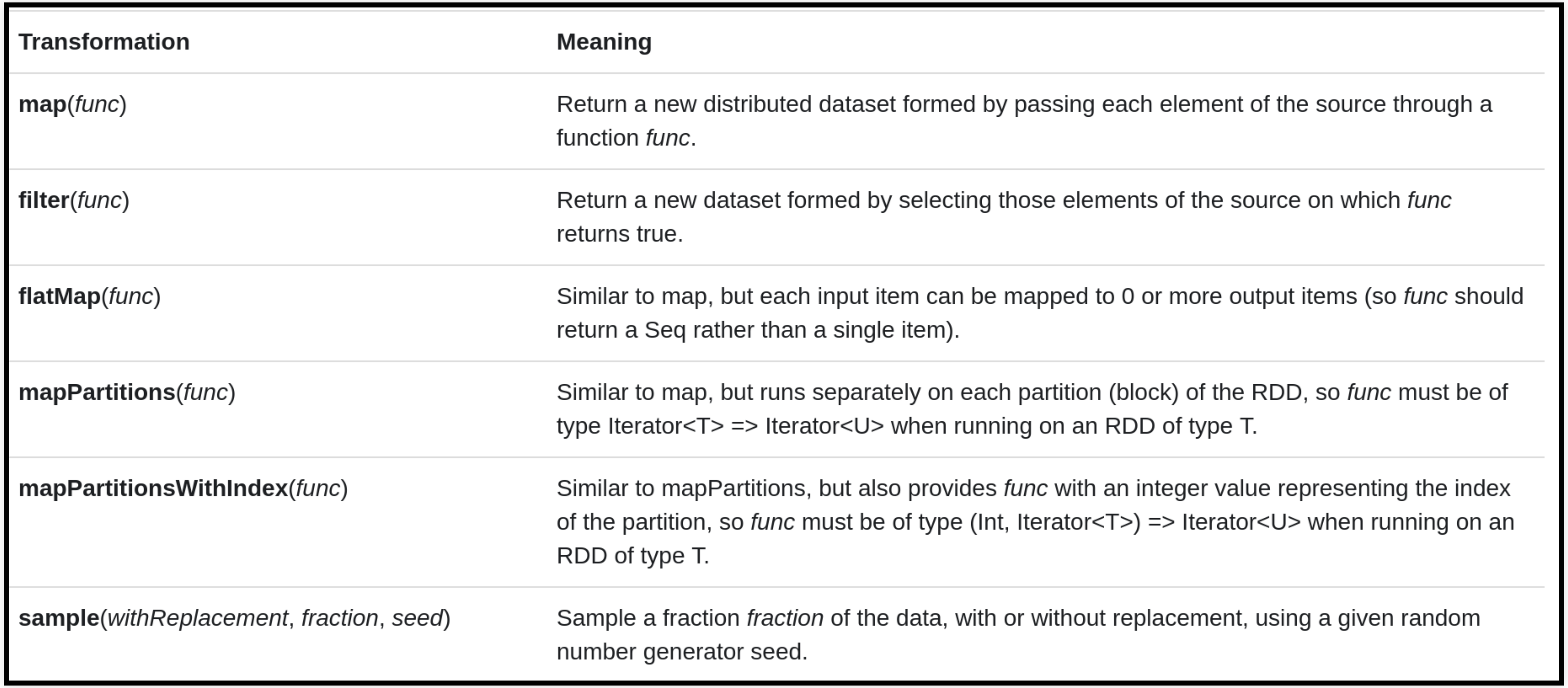
Spark programming in 3 steps

Create RDDs

- Parallelize: *sc.parallelize(Array)*
- Reading from file/HDFS/S3: *sc.textFile(FileURL)*

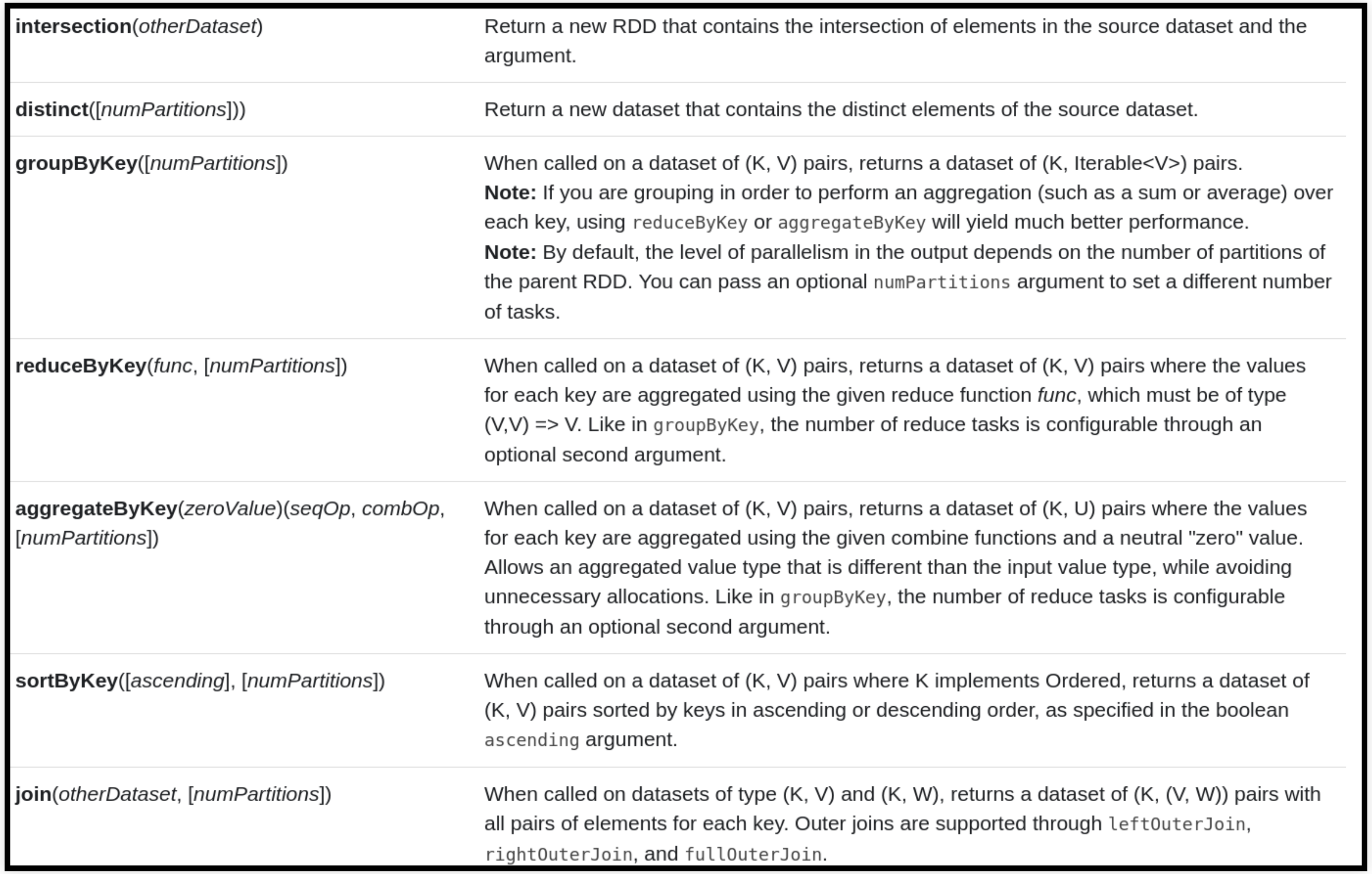
Transform RDDs

http://spark.apache.org/docs/latest/programming-guide.html#transformations



Transformation	Meaning
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).
mapPartitions(*func*)	Similar to map, but runs separately on each partition (block) of the RDD, so *func* must be of type Iterator<T> => Iterator<U> when running on an RDD of type T.
mapPartitionsWithIndex(*func*)	Similar to mapPartitions, but also provides *func* with an integer value representing the index of the partition, so *func* must be of type (Int, Iterator<T>) => Iterator<U> when running on an RDD of type T.
sample(*withReplacement*, *fraction*, *seed*)	Sample a fraction *fraction* of the data, with or without replacement, using a given random number generator seed.

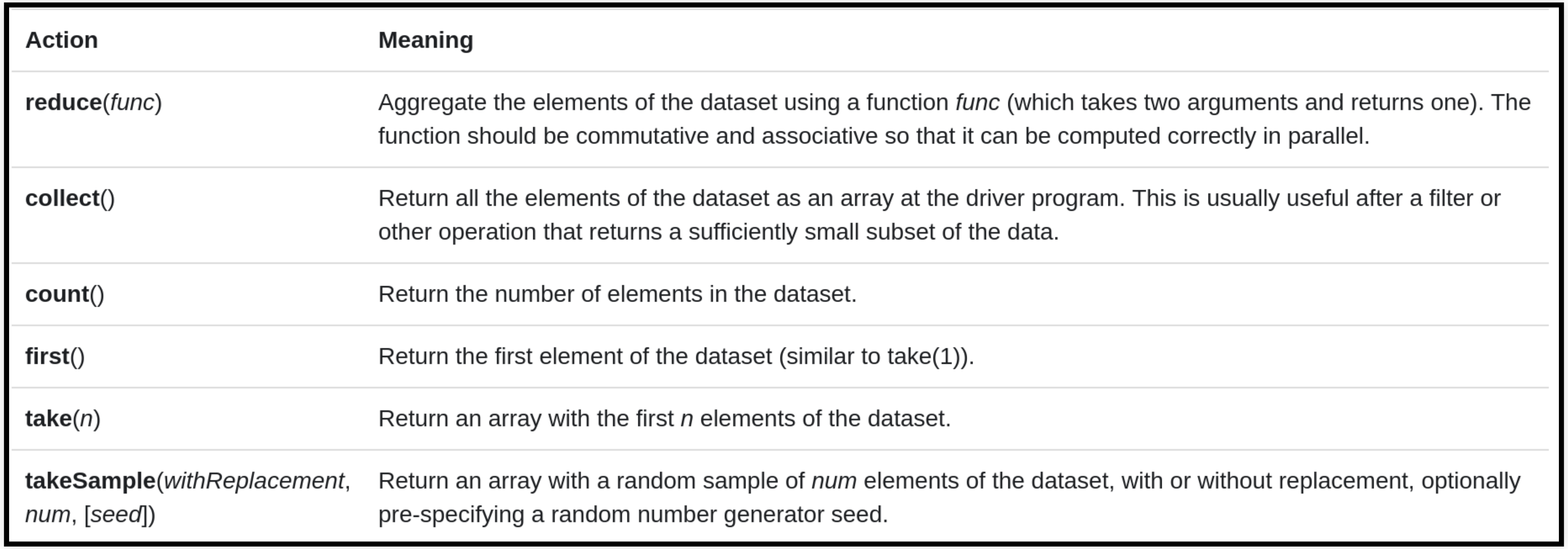
Transform RDDs



intersection(*otherDataset*)	Return a new RDD that contains the intersection of elements in the source dataset and the argument.
distinct([*numPartitions*]))	Return a new dataset that contains the distinct elements of the source dataset.
groupByKey([*numPartitions*])	When called on a dataset of (K, V) pairs, returns a dataset of (K, Iterable<V>) pairs. **Note:** If you are grouping in order to perform an aggregation (such as a sum or average) over each key, using `reduceByKey` or `aggregateByKey` will yield much better performance. **Note:** By default, the level of parallelism in the output depends on the number of partitions of the parent RDD. You can pass an optional `numPartitions` argument to set a different number of tasks.
reduceByKey(*func*, [*numPartitions*])	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 *func*, which must be of type (V,V) => V. Like in `groupByKey`, the number of reduce tasks is configurable through an optional second argument.
aggregateByKey(*zeroValue*)(*seqOp*, *combOp*, [*numPartitions*])	When called on a dataset of (K, V) pairs, returns a dataset of (K, U) pairs where the values for each key are aggregated using the given combine functions and a neutral "zero" value. Allows an aggregated value type that is different than the input value type, while avoiding unnecessary allocations. Like in `groupByKey`, the number of reduce tasks is configurable through an optional second argument.
sortByKey([*ascending*], [*numPartitions*])	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*, [*numPartitions*])	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. Outer joins are supported through `leftOuterJoin`, `rightOuterJoin`, and `fullOuterJoin`.

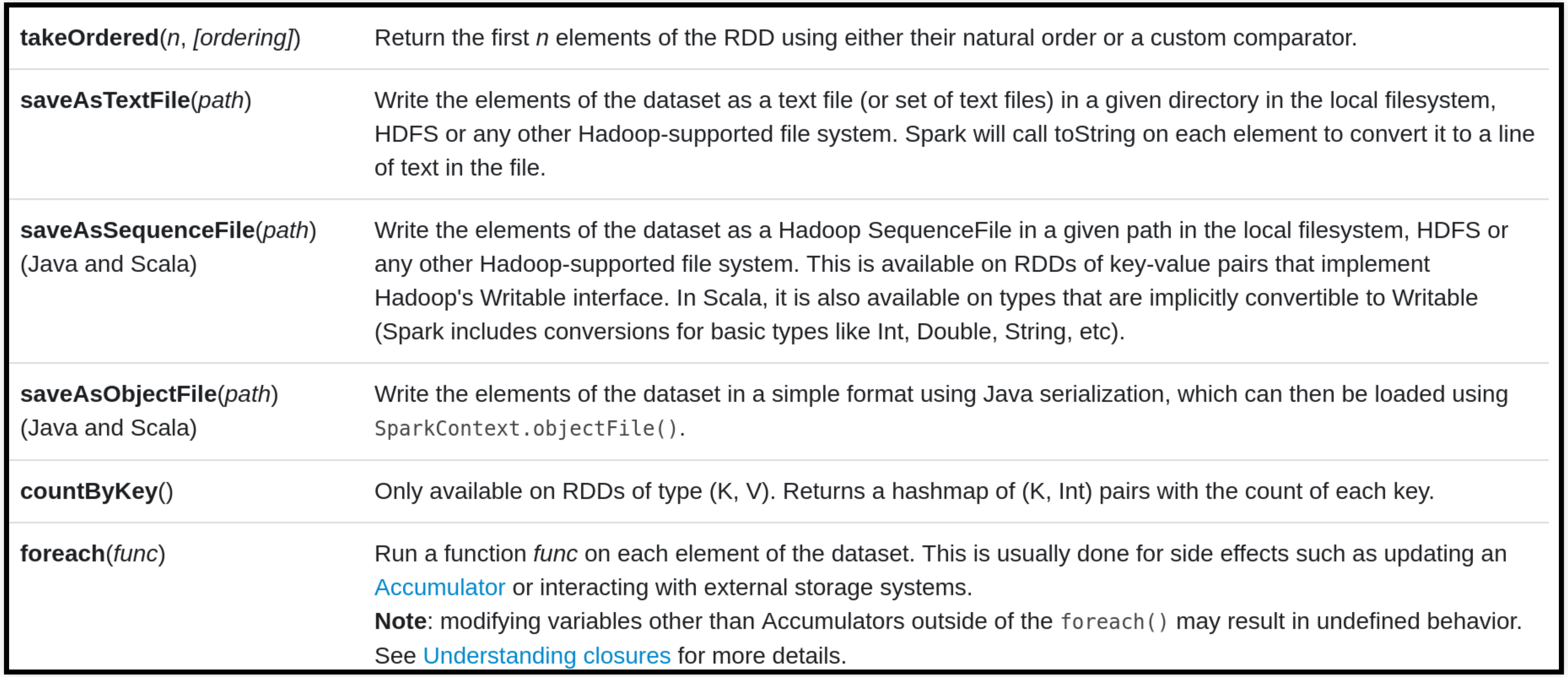
Actions

http://spark.apache.org/docs/latest/programming-guide.html#actions



Action	Meaning
reduce(*func*)	Aggregate the elements of the dataset using a function *func* (which takes two arguments and returns one). The function should 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. This is 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.
takeSample(*withReplacement*, *num*, [*seed*])	Return an array with a random sample of *num* elements of the dataset, with or without replacement, optionally pre-specifying a random number generator seed.

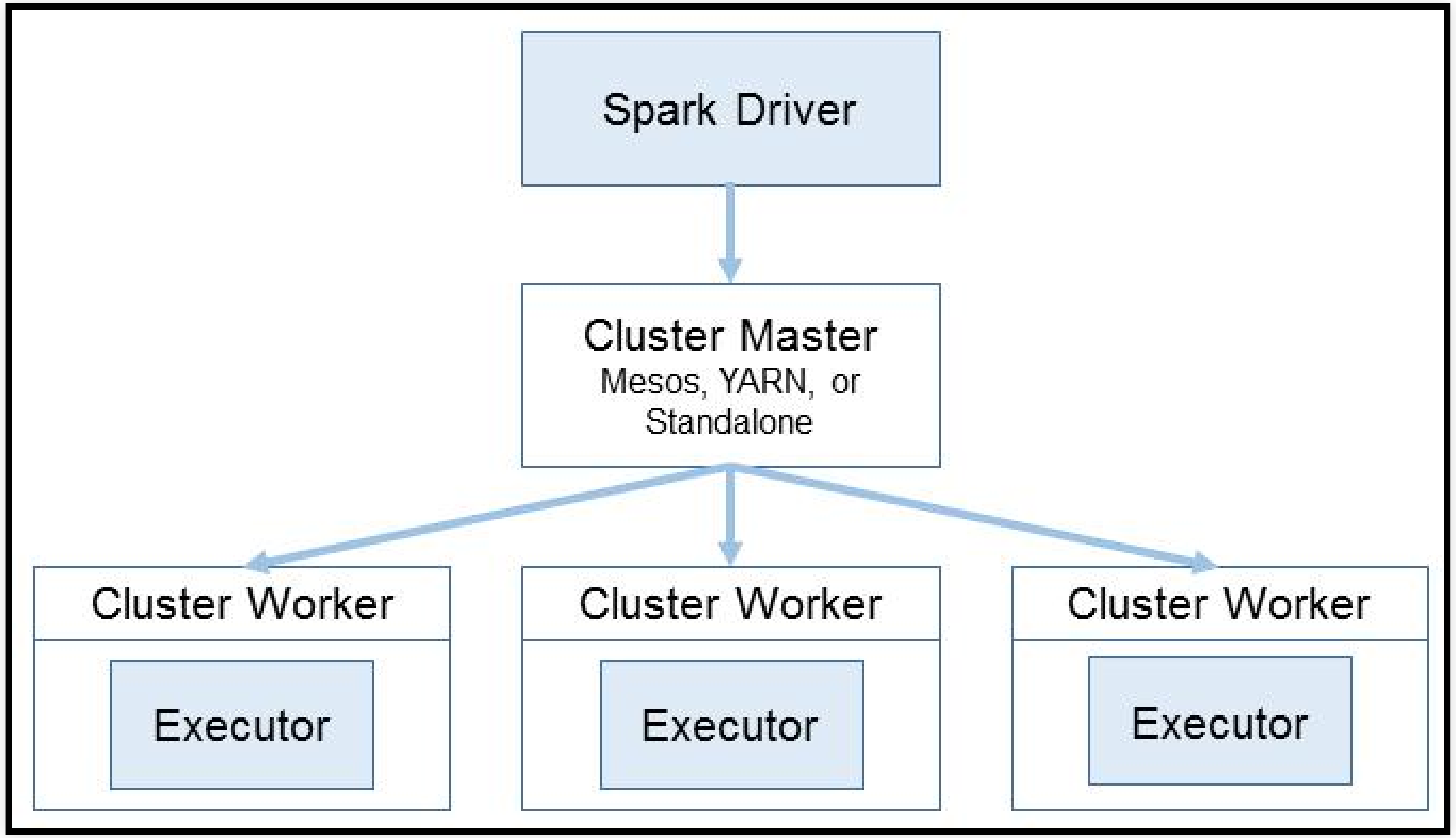
Actions



takeOrdered(*n*, *[ordering]*)	Return the first *n* elements of the RDD using either their natural order or a custom comparator.
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*) (Java and Scala)	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. This is available on RDDs of key-value pairs that implement Hadoop's Writable interface. In Scala, it is also available on types that are implicitly convertible to Writable (Spark includes conversions for basic types like Int, Double, String, etc).
saveAsObjectFile(*path*) (Java and Scala)	Write the elements of the dataset in a simple format using Java serialization, which can then be loaded using `SparkContext.objectFile()`.
countByKey()	Only available on RDDs of type (K, V). Returns a hashmap of (K, Int) pairs with the count of each key.
foreach(*func*)	Run a function *func* on each element of the dataset. This is usually done for side effects such as updating an Accumulator or interacting with external storage systems. **Note**: modifying variables other than Accumulators outside of the `foreach()` may result in undefined behavior. See Understanding closures for more details.

Plan

Spark Cluster Architecture



Follow execution in SparkUI (1)

```
val data = 1 to 1000    1
```

(1) DRIVER: generate a sequence of numbers

Follow execution in SparkUI (2)

```
val data = 1 to 1000    1
val parallelizedData = sc.parallelize(data)   2
```

(2) DRIVER: define a RDD by parallelizing local data

Follow execution in SparkUI (3)

```
val data = 1 to 1000    1
val parallelizedData = sc.parallelize(data)  2
val filteredData = parallelizedData.filter(_%2==0)  3
```

(3) DRIVER: define another RDD that contains the filtered data

Follow execution in SparkUI (4)

```
val data = 1 to 1000    1
val parallelizedData = sc.parallelize(data)  2
val filteredData = parallelizedData.filter(_%2==0)  3
val sortedData = filteredData.sortBy( (x:Int) => - x ) 4
```

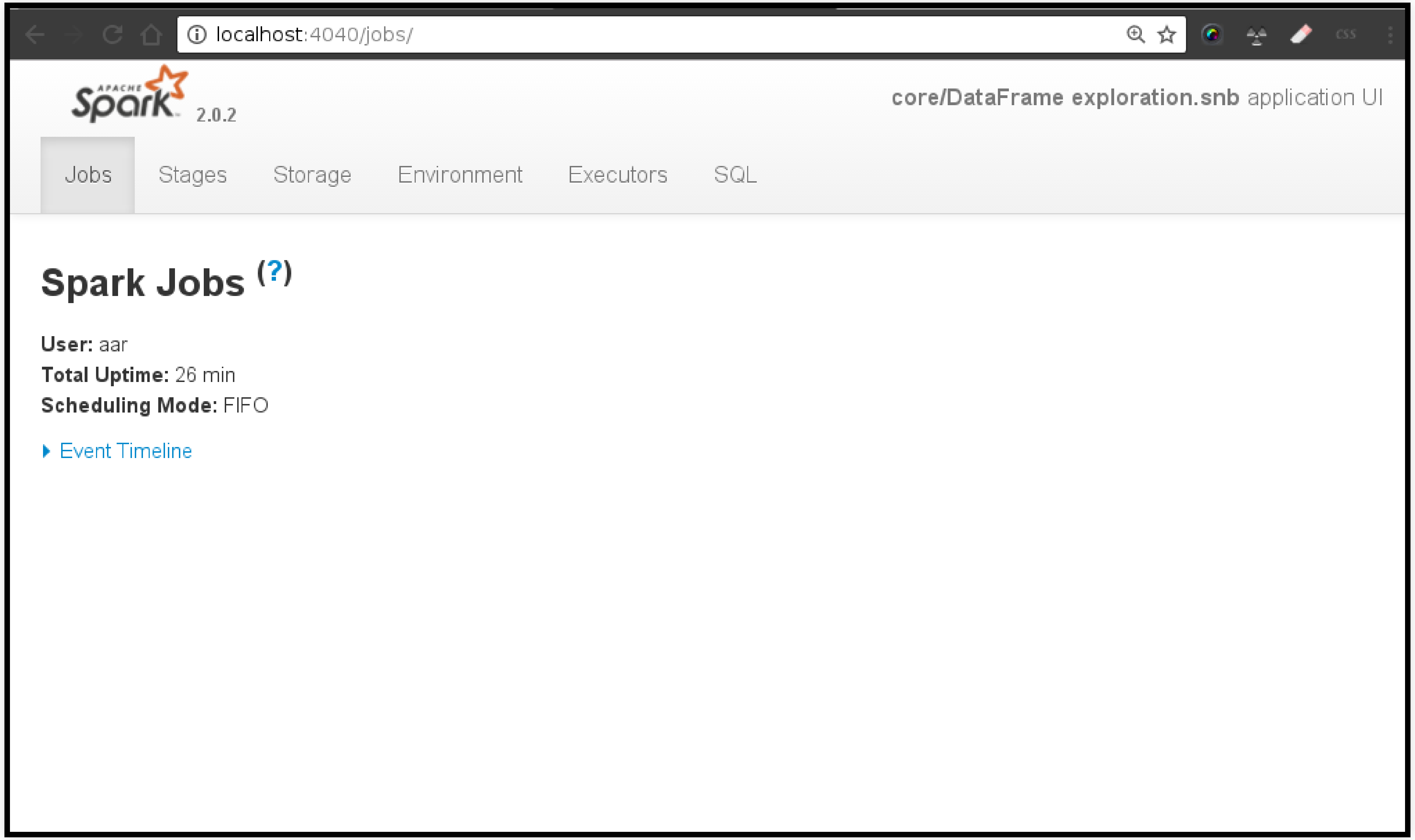
(4) DRIVER: define another RDD that contains sorted data

Follow execution in SparkUI (5)

```
val data = 1 to 1000    1
val parallelizedData = sc.parallelize(data)  2
val filteredData = parallelizedData.filter(_%2==0)  3
val sortedData = filteredData.sortBy( (x:Int) => - x ) 4
val groupedData = sortedData.groupBy( (x:Int)=> x%3) 5
```

(5) DRIVER: define another RDD that contains groupped data

Follow execution in SparkUI (5)



localhost:4040/jobs/

2.0.2

core/DataFrame exploration.snb application UI

Spark Jobs (?)

User: aar
Total Uptime: 26 min
Scheduling Mode: FIFO

Event Timeline

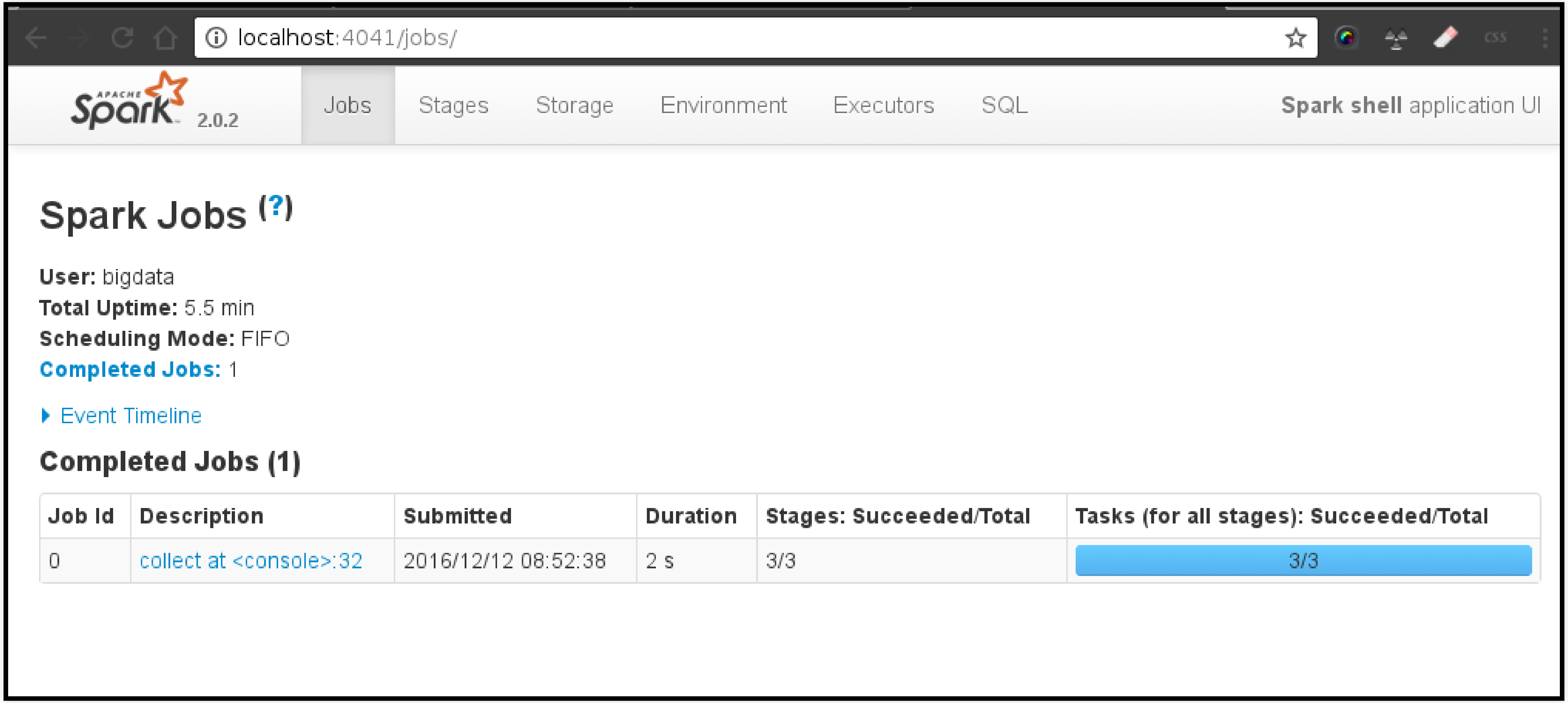
Follow execution in SparkUI (6)

```
val data = 1 to 1000   1
val parallelizedData = sc.parallelize(data)  2
val filteredData = parallelizedData.filter(_%2==0)  3
val sortedData = filteredData.sortBy( (x:Int) => - x ) 4
val groupedData = sortedData.groupBy( (x:Int)=> x%3) 5
groupedData.collect 6
```

(6) DRIVER → MASTER → WORKERs → MASTER → DRIVER

- **An action invoked on a RDD (sortBy)** ⇒ starts the distributed computation:
  - DRIVER: sends a JOB to the MASTER (DAG + data)
  - MASTER: partition the data, calculate the pysical plan (Stages/) and sends tasks to the WORKERS
  - WORKERs: execute tasks and return to the MASTER
  - MASTER sends the response back to the DRIVER

Follow execution in SparkUI (6)



Spark Jobs (?)

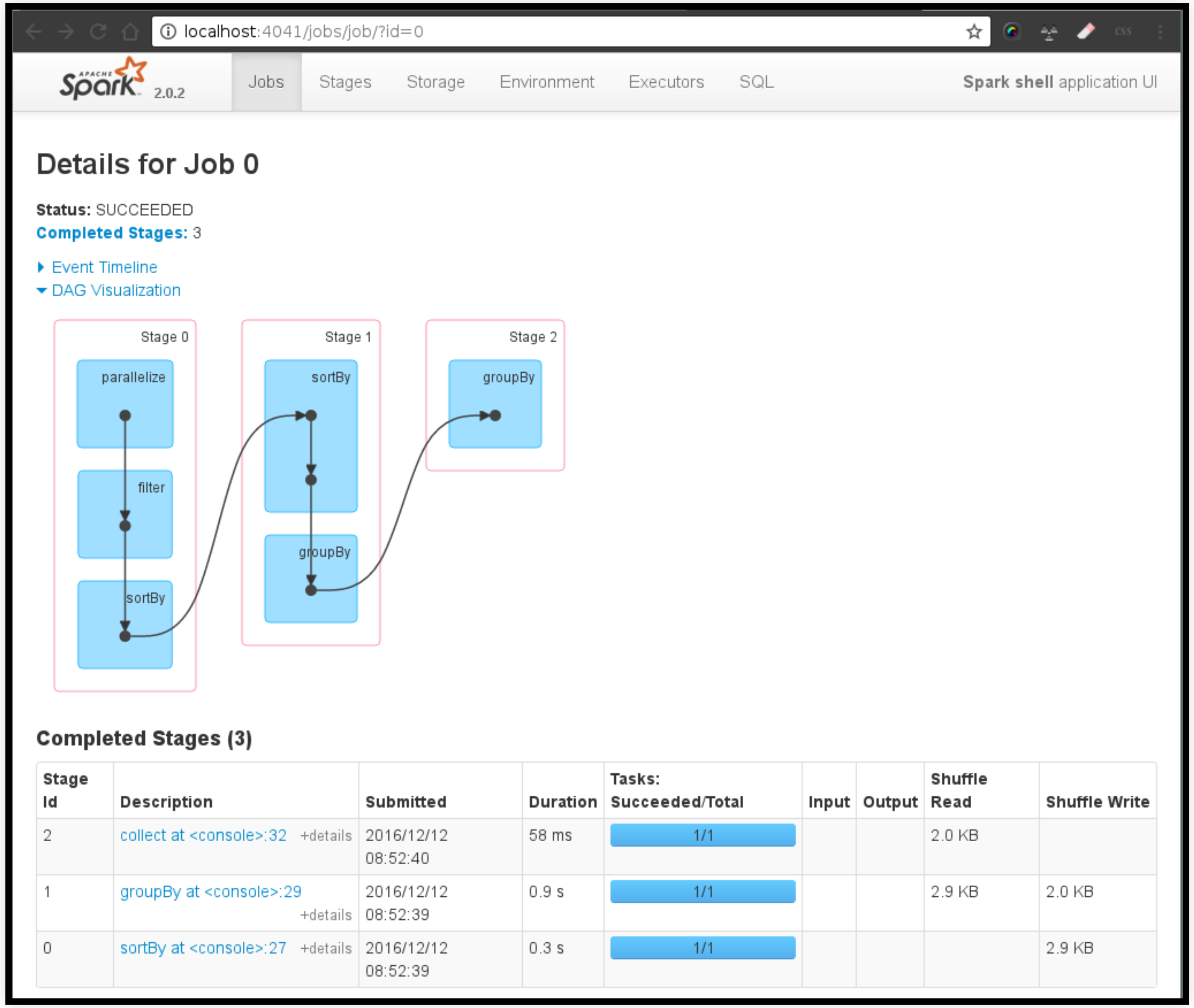
User: bigdata
Total Uptime: 5.5 min
Scheduling Mode: FIFO
Completed Jobs: 1

▸ Event Timeline

Completed Jobs (1)

Job Id	Description	Submitted	Duration	Stages: Succeeded/Total	Tasks (for all stages): Succeeded/Total
0	collect at <console>:32	2016/12/12 08:52:38	2 s	3/3	3/3

Follow execution in SparkUI (6)



Details for Job 0

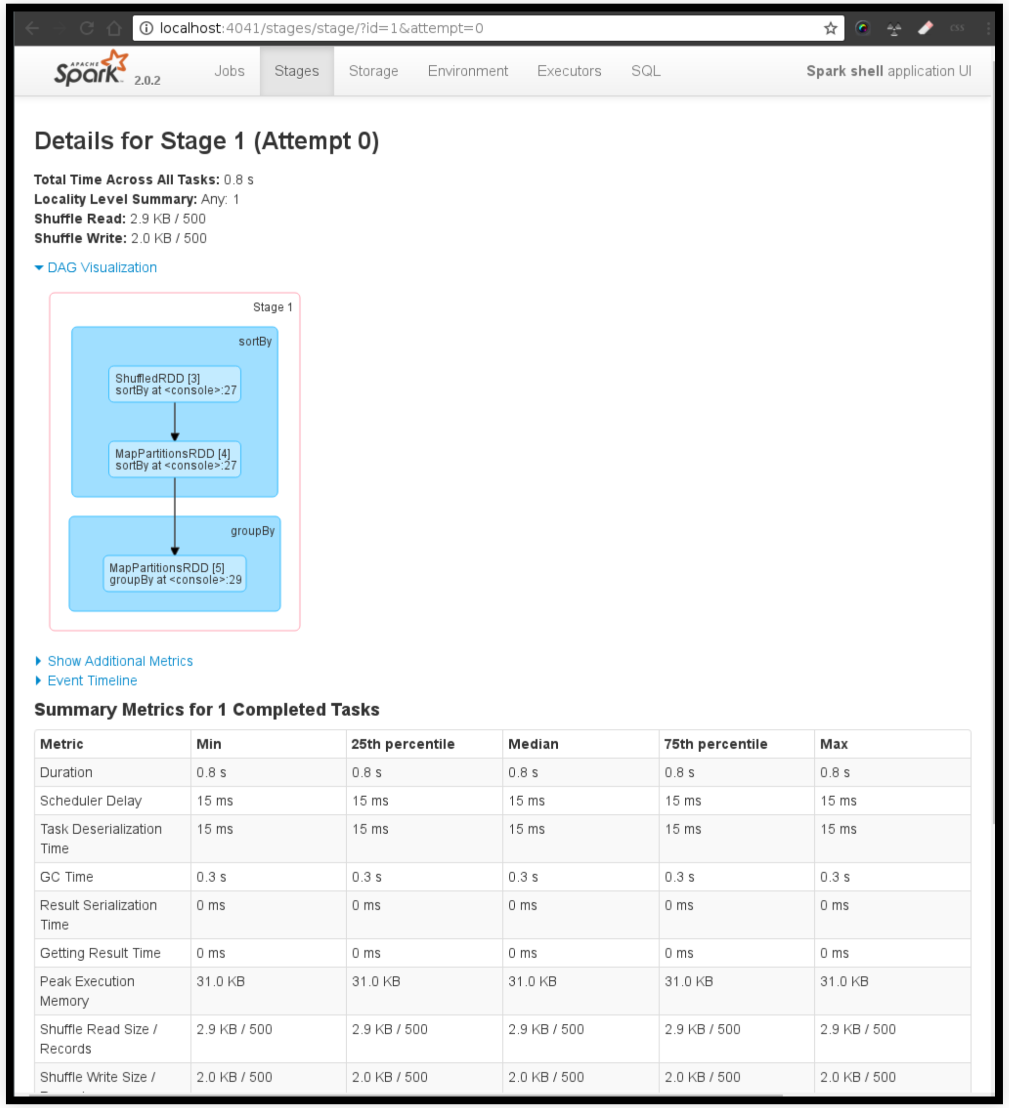
Status: SUCCEEDED
Completed Stages: 3

- Event Timeline
- DAG Visualization

Completed Stages (3)

Stage Id	Description	Submitted	Duration	Tasks: Succeeded/Total	Input	Output	Shuffle Read	Shuffle Write
2	collect at <console>:32 +details	2016/12/12 08:52:40	58 ms	1/1			2.0 KB	
1	groupBy at <console>:29 +details	2016/12/12 08:52:39	0.9 s	1/1			2.9 KB	2.0 KB
0	sortBy at <console>:27 +details	2016/12/12 08:52:39	0.3 s	1/1				2.9 KB

Follow execution in SparkUI (6)


localhost:4041/stages/stage/?id=1&attempt=0
Spark 2.0.2
Jobs
Stages
Storage
Environment
Executors
SQL
Spark shell application UI
Stage 1
sortBy
ShuffledRDD [3] sortBy at <console>:27
MapPartitionsRDD [4] sortBy at <console>:27
groupBy
MapPartitionsRDD [5] groupBy at <console>:29

Details for Stage 1 (Attempt 0)

Total Time Across All Tasks: 0.8 s
Locality Level Summary: Any: 1
Shuffle Read: 2.9 KB / 500
Shuffle Write: 2.0 KB / 500

▾ DAG Visualization

▸ Show Additional Metrics
▸ Event Timeline

Summary Metrics for 1 Completed Tasks

Metric	Min	25th percentile	Median	75th percentile	Max
Duration	0.8 s	0.8 s	0.8 s	0.8 s	0.8 s
Scheduler Delay	15 ms	15 ms	15 ms	15 ms	15 ms
Task Deserialization Time	15 ms	15 ms	15 ms	15 ms	15 ms
GC Time	0.3 s	0.3 s	0.3 s	0.3 s	0.3 s
Result Serialization Time	0 ms	0 ms	0 ms	0 ms	0 ms
Getting Result Time	0 ms	0 ms	0 ms	0 ms	0 ms
Peak Execution Memory	31.0 KB	31.0 KB	31.0 KB	31.0 KB	31.0 KB
Shuffle Read Size / Records	2.9 KB / 500	2.9 KB / 500	2.9 KB / 500	2.9 KB / 500	2.9 KB / 500
Shuffle Write Size /	2.0 KB / 500	2.0 KB / 500	2.0 KB / 500	2.0 KB / 500	2.0 KB / 500

Plan

SparkSQL

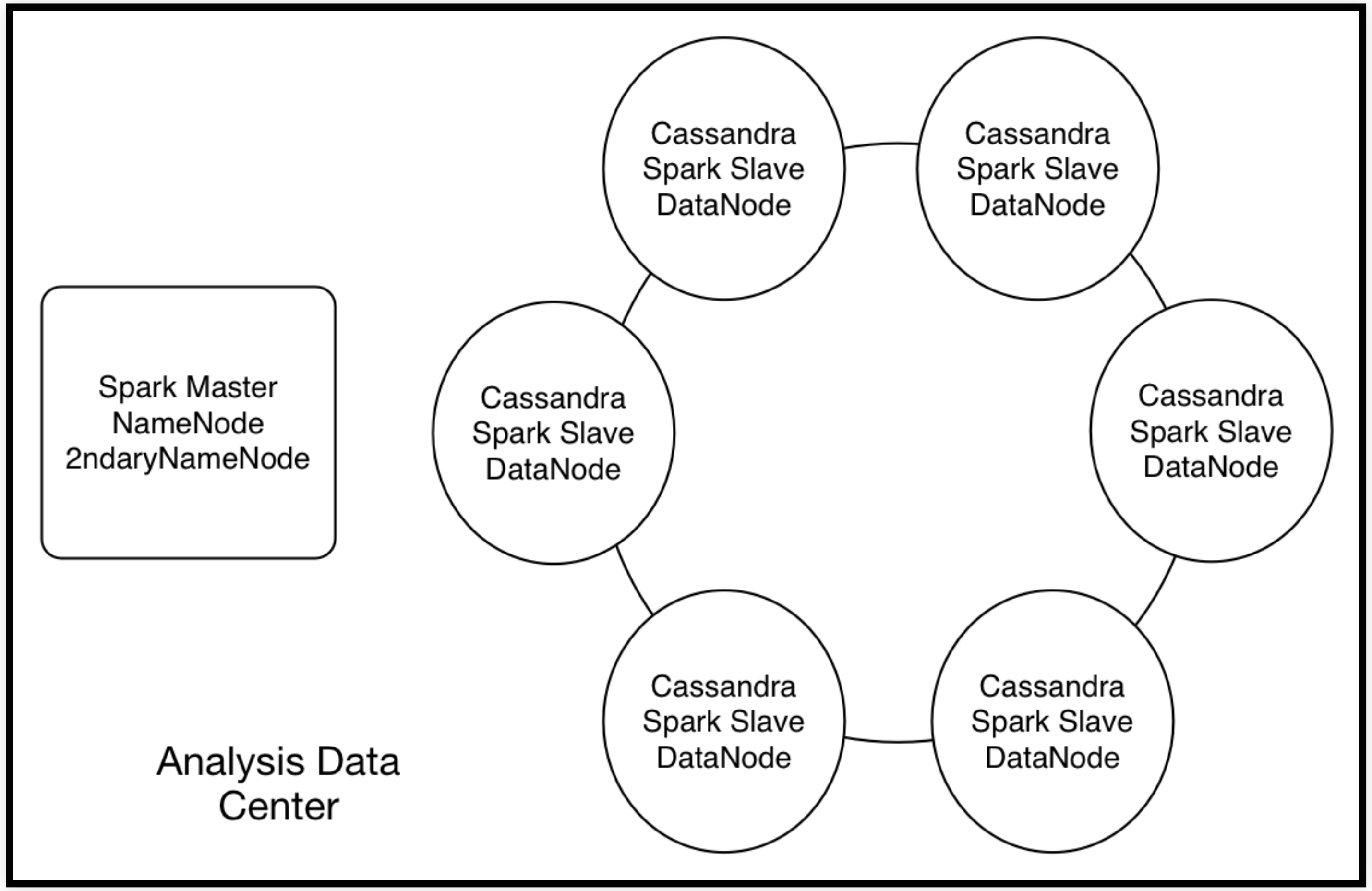
- storing general objects in relational like tables
- provide a SQL like interface for querying

SparkSQL example

```
val sqlContext = new org.apache.spark.sql.SQLContext(sc) 1

case class Person(name: String, age: Int) 2

val people = sc.textFile("persons.txt")
            .map(_.split(","))
            .map(p => Person(p(0), p(1).trim.toInt)
            .toDF 3

people.createOrReplaceTempView("people") 4

val teenagers = sql("""
        SELECT * from people
        WHERE age < 15
        """) 5

teenagers.show 6
```

http://spark.apache.org/docs/latest/sql-programming-guide.html

Cassandra and Spark

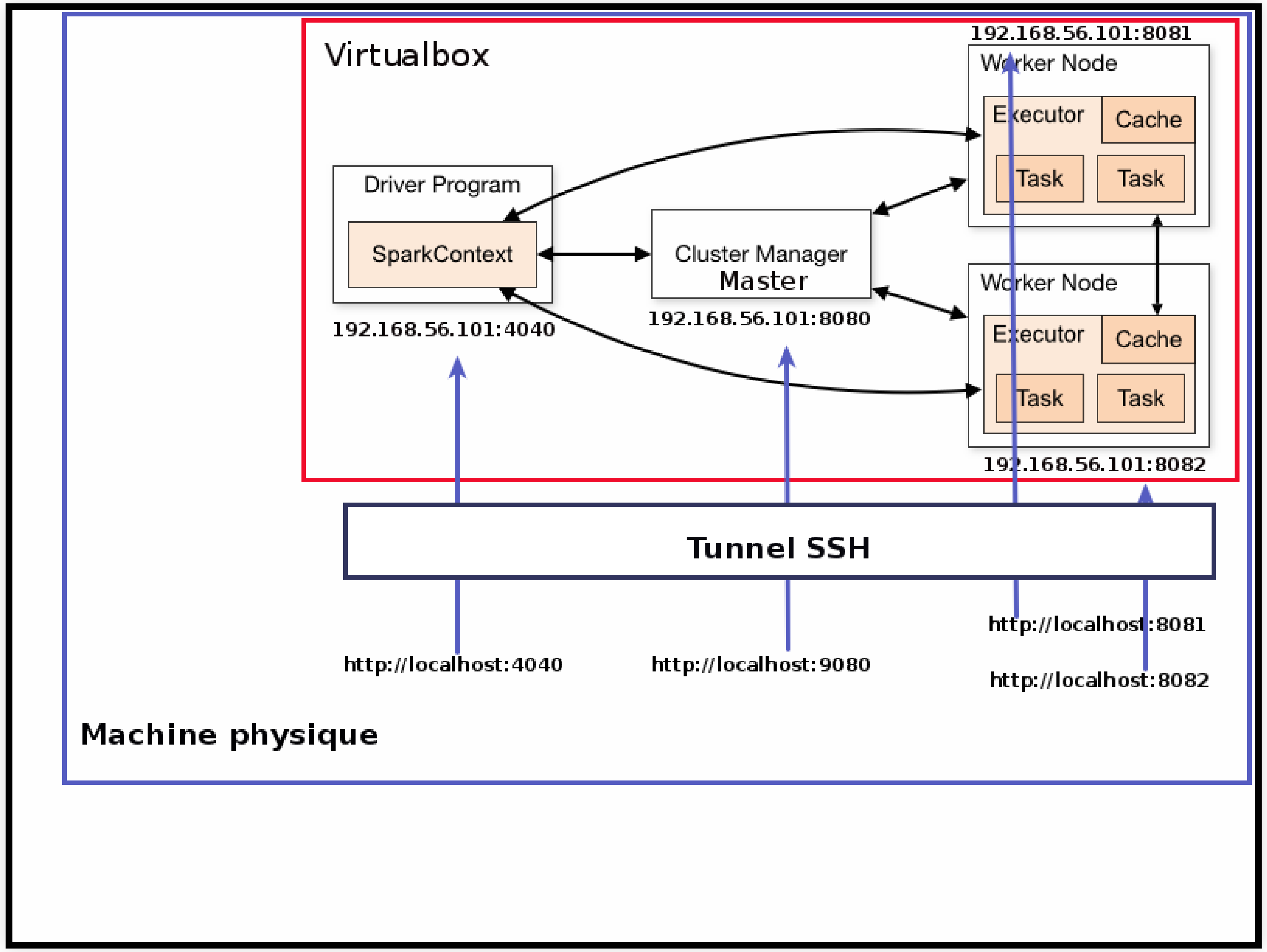

Cassandra Spark Slave DataNode
Cassandra Spark Slave DataNode
Spark Master NameNode 2ndaryNameNode
Cassandra Spark Slave DataNode
Cassandra Spark Slave DataNode
Cassandra Spark Slave DataNode
Cassandra Spark Slave DataNode
Analysis Data Center

Cassandra and Spark example

```
CassandraConnector(conf).withSessionDo { session =>
  session.execute(s"""
      CREATE KEYSPACE IF NOT EXISTS demo
      WITH REPLICATION = {'class': 'SimpleStrategy', 'replication_factor': 1 }""")
  session.execute(s"CREATE TABLE IF NOT EXISTS demo.wordcount (word TEXT PRIMARY KEY, count COUNTER)")
}

sc.textFile(words)
  .flatMap(_.split("\\s+"))
  .map(word => (word.toLowerCase, 1))
  .reduceByKey(_ + _)
  .saveToCassandra("demo", "wordcount")

// print out the data saved from Spark to Cassandra
sc.cassandraTable("demo", "wordcount")
  .collect.foreach(println)
```

See Datastax Spark Driver documentation on github

TP Spark: RDDs on sparkShell



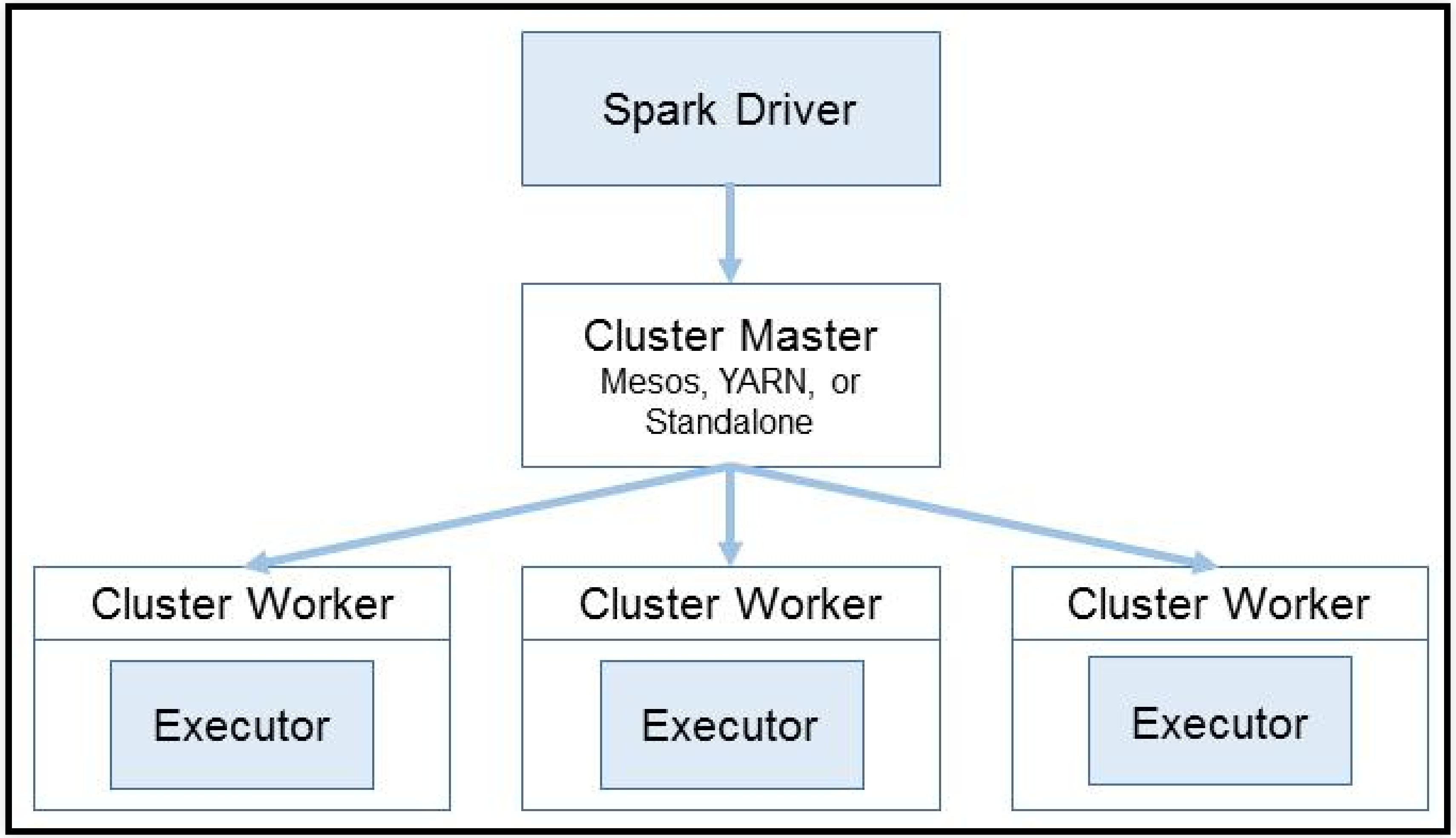
TP Spark RDD,SparkSQL,Cassandra

Spark Dataframes & Datasets

Plan

1. **RDD execution review**
2. Best practices
3. Dataframes & Dataset API
4. Spark notebooks
5. TP2 Data exploration with the Dataframe API

Spark Cluster Architecture



Follow execution in SparkUI (1)

```
val data = 1 to 1000    1
```

(1) DRIVER: generate a sequence of numbers

Follow execution in SparkUI (2)

```
val data = 1 to 1000    1
val parallelizedData = sc.parallelize(data)   2
```

(2) DRIVER: define a RDD by parallelizing local data

Follow execution in SparkUI (3)

```
val data = 1 to 1000    1
val parallelizedData = sc.parallelize(data)   2
val filteredData = parallelizedData.filter(_%2==0)   3
```

(3) DRIVER: define another RDD that contains the filtered data

Follow execution in SparkUI (4)

```
val data = 1 to 1000    1
val parallelizedData = sc.parallelize(data)  2
val filteredData = parallelizedData.filter(_%2==0)  3
val sortedData = filteredData.sortBy( (x:Int) => - x ) 4
```

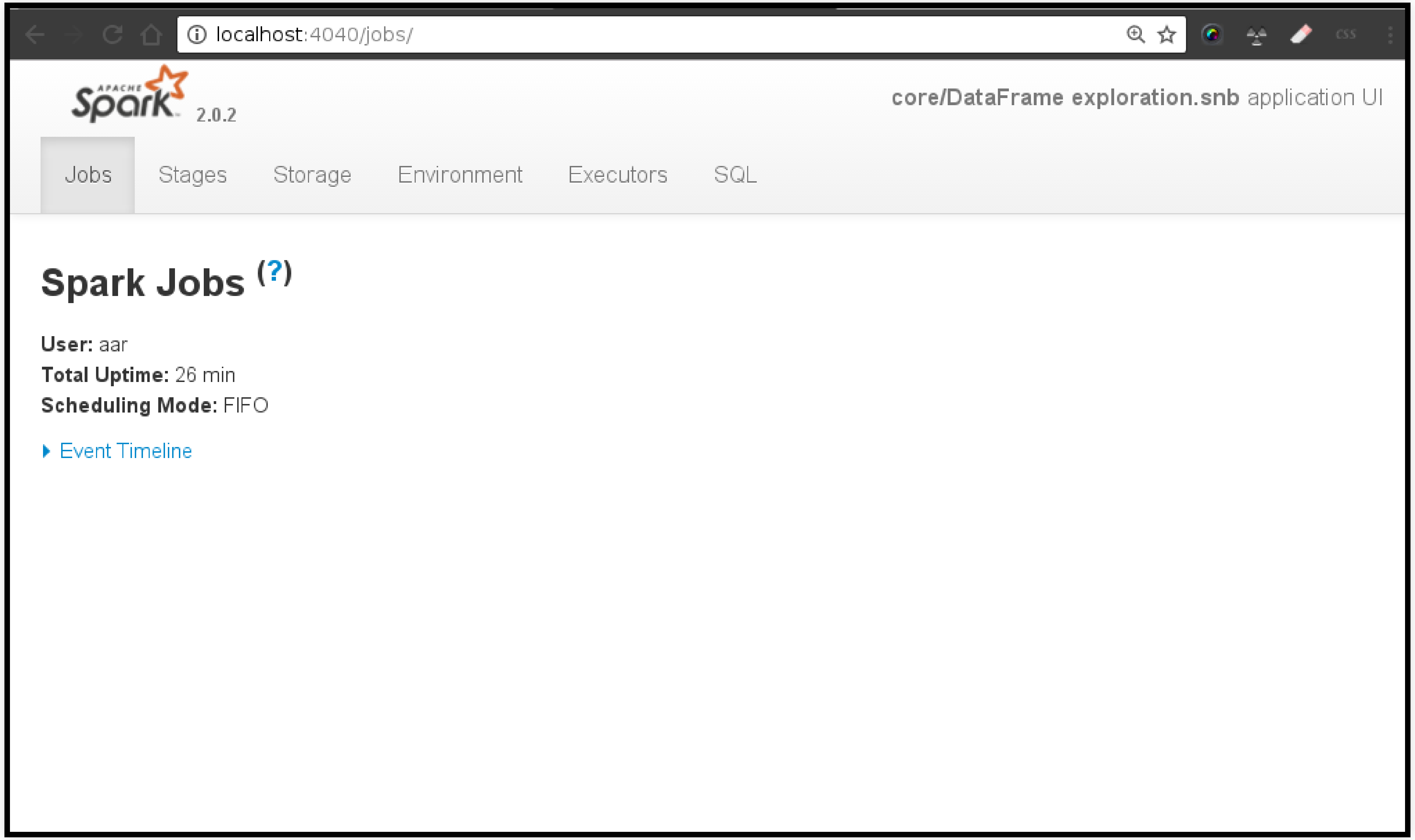
(4) DRIVER: define another RDD that contains sorted data

Follow execution in SparkUI (5)

```
val data = 1 to 1000    1
val parallelizedData = sc.parallelize(data)  2
val filteredData = parallelizedData.filter(_%2==0)  3
val sortedData = filteredData.sortBy( (x:Int) => - x ) 4
val groupedData = sortedData.groupBy( (x:Int)=> x%3) 5
```

(5) DRIVER: define another RDD that contains groupped data

Follow execution in SparkUI (5)



Spark Jobs (?)

User: aar
Total Uptime: 26 min
Scheduling Mode: FIFO

Event Timeline

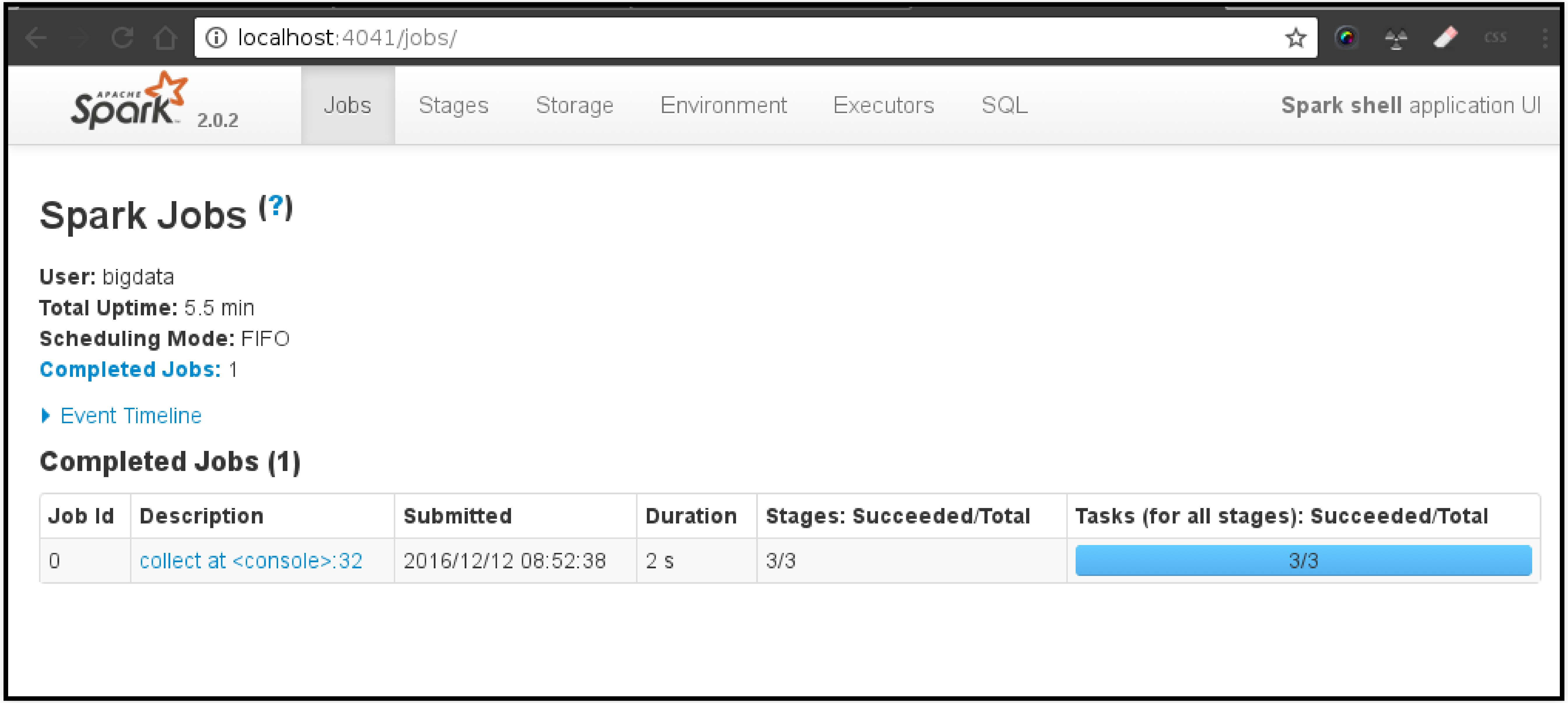
Follow execution in SparkUI (6)

```
val data = 1 to 1000    1
val parallelizedData = sc.parallelize(data)  2
val filteredData = parallelizedData.filter(_%2==0)  3
val sortedData = filteredData.sortBy( (x:Int) => - x ) 4
val groupedData = sortedData.groupBy( (x:Int)=> x%3) 5
groupedData.collect 6
```

(6) DRIVER → MASTER → WORKERs → MASTER → DRIVER

- **An action invoked on a RDD (sortBy)** ⇒ starts the distributed computation:
  - DRIVER: sends a JOB to the MASTER (DAG + data)
  - MASTER: partition the data, calculate the pysical plan (Stages/) and sends tasks to the WORKERS
  - WORKERs: execute tasks and return to the MASTER
  - MASTER sends the response back to the DRIVER

Follow execution in SparkUI (6)



localhost:4041/jobs/

Spark 2.0.2 | Jobs | Stages | Storage | Environment | Executors | SQL | **Spark shell** application UI

Spark Jobs (?)

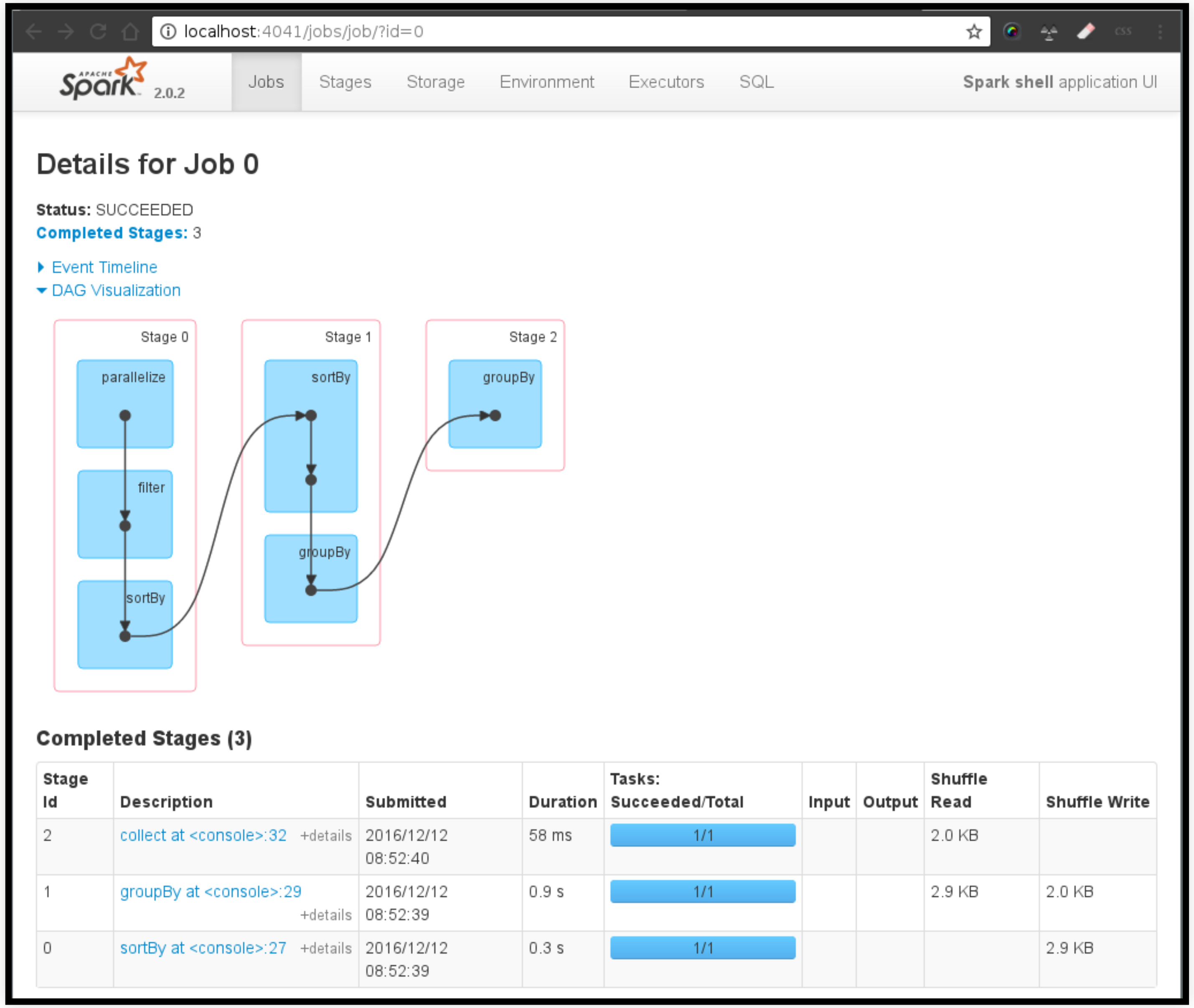
User: bigdata
Total Uptime: 5.5 min
Scheduling Mode: FIFO
Completed Jobs: 1

▸ Event Timeline

Completed Jobs (1)

Job Id	Description	Submitted	Duration	Stages: Succeeded/Total	Tasks (for all stages): Succeeded/Total
0	collect at <console>:32	2016/12/12 08:52:38	2 s	3/3	3/3

Follow execution in SparkUI (6)



Details for Job 0

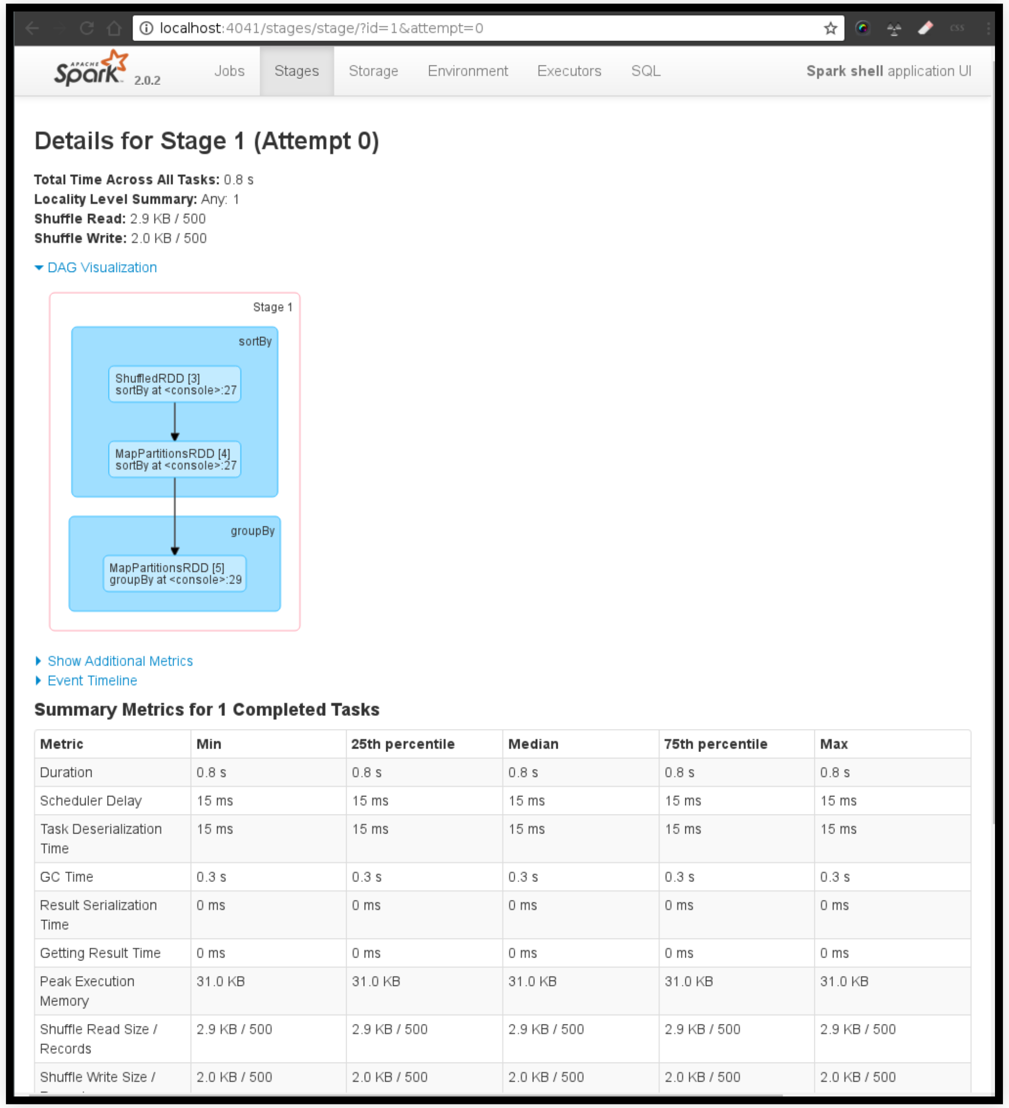
Status: SUCCEEDED
Completed Stages: 3

▸ Event Timeline
▾ DAG Visualization

Completed Stages (3)

Stage Id	Description	Submitted	Duration	Tasks: Succeeded/Total	Input	Output	Shuffle Read	Shuffle Write
2	collect at <console>:32 +details	2016/12/12 08:52:40	58 ms	1/1			2.0 KB	
1	groupBy at <console>:29 +details	2016/12/12 08:52:39	0.9 s	1/1			2.9 KB	2.0 KB
0	sortBy at <console>:27 +details	2016/12/12 08:52:39	0.3 s	1/1				2.9 KB

Follow execution in SparkUI (6)


localhost:4041/stages/stage/?id=1&attempt=0
Spark 2.0.2
Jobs
Stages
Storage
Environment
Executors
SQL
Spark shell application UI
Stage 1
sortBy
ShuffledRDD [3] sortBy at <console>:27
MapPartitionsRDD [4] sortBy at <console>:27
groupBy
MapPartitionsRDD [5] groupBy at <console>:29
Show Additional Metrics
Event Timeline

Details for Stage 1 (Attempt 0)

Total Time Across All Tasks: 0.8 s
Locality Level Summary: Any: 1
Shuffle Read: 2.9 KB / 500
Shuffle Write: 2.0 KB / 500

DAG Visualization

Summary Metrics for 1 Completed Tasks

Metric	Min	25th percentile	Median	75th percentile	Max
Duration	0.8 s	0.8 s	0.8 s	0.8 s	0.8 s
Scheduler Delay	15 ms	15 ms	15 ms	15 ms	15 ms
Task Deserialization Time	15 ms	15 ms	15 ms	15 ms	15 ms
GC Time	0.3 s	0.3 s	0.3 s	0.3 s	0.3 s
Result Serialization Time	0 ms	0 ms	0 ms	0 ms	0 ms
Getting Result Time	0 ms	0 ms	0 ms	0 ms	0 ms
Peak Execution Memory	31.0 KB	31.0 KB	31.0 KB	31.0 KB	31.0 KB
Shuffle Read Size / Records	2.9 KB / 500	2.9 KB / 500	2.9 KB / 500	2.9 KB / 500	2.9 KB / 500
Shuffle Write Size /	2.0 KB / 500	2.0 KB / 500	2.0 KB / 500	2.0 KB / 500	2.0 KB / 500

RDD review

- **construct a RDD** (parallelize/ read from distributed datasources)
- **transformations** ⇒ DAG (lazy)
  - map, filter, flatMap, mapPartitions, mapPartitionsWithIndex, sample, union, intersection, distinct, groupByKey, reduceByKey, sortByKey, join, cogroup, repatition, cartesian, glom, …
- **actions** ⇒ execute DAG + retrieve result on the DRIVER
  - reduce, collect, count, first, take, foreach, saveAs…, min, max, …

Plan

1. RDD execution review
2. **Best practices**
3. Dataframes & Dataset API
4. Spark notebooks
5. TP2 Data exploration with the Dataframe API

Use ReduceByKey over GroupByKey

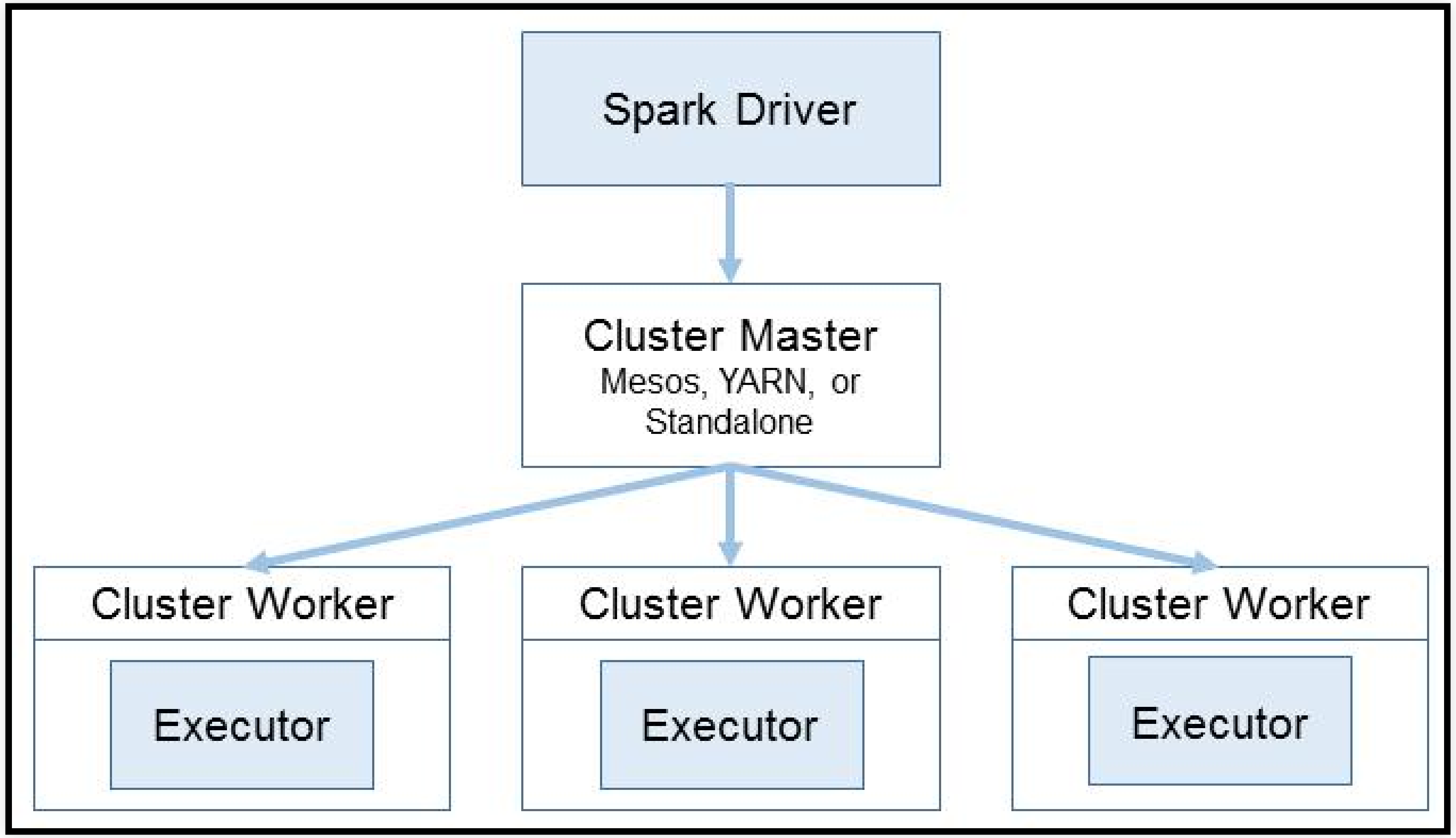
```
val words = Array("one", "two", "two", "three", "three", "three")
val wordPairsRDD = sc.parallelize(words).map(word => (word, 1))

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

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

- ***groupByKey*** - all the key-value pairs are shuffled around
- ***reduceByKey*** - combine output with a common key on each partition before shuffling the data.

Task not serializable Exception (1)

```
var myClass = new NonSerializableClass()

var rdd = sc.parallelize(data)

// Wrong: Don't do this!!
rdd.foreach( ... myClass ...)
```

Spark Cluster Architecture



Spark execution modes

- ***Local***: run locally with as many worker threads as logical cores

```
./bin/spark-shell --master local[*]
```

- ***Standalone***

```
./bin/spark-shell --master spark://HOST1:PORT1
./bin/spark-shell --master spark://HOST1:PORT1,HOST2:PORT2
```

- ***Cluster mode*** (Mesos, Yarn, Kubernetes)

```
./bin/spark-shell --master mesos://HOST:PORT
./bin/spark-shell --master yarn
./bin/spark-shell --master k8s://HOST:PORT
```

Data serialization

- **convenience** (*work with any Java type*) VS **performance**
- *Java serialization* [default]: works with any class you create that implements java.io.Serializable
  - flexible but often quite slow, and leads to large serialized formats for many classes
- *Kryo* serialization: significantly faster and more compact (10x)
  - does not support all Serializable types, requires you to register the classes you'll use in the program in advance for best performance

```
val conf = new SparkConf().setMaster(...).setAppName(...)
conf.set("spark.serializer", "org.apache.spark.serializer.KryoSerializer")
conf.registerKryoClasses(Array(classOf[MyClass1], classOf[MyClass2]))
val sc = new SparkContext(conf)
```

Task not serializable Exception (2)

```
var myClass = new NonSerializableClass()

var rdd = sc.parallelize(data)

// Wrong: Don't do this!!
rdd.foreach( ... myClass ...)
```

- **use only serializable classes in closures**

Task not serializable Exception (3)

```
object AwsClient{
    val s3 = new AmazonS3Client(new BasicAWSCredentials(AWS_ID, AWS_KEY))
}


sampleDF.foreach( r=> {
            ...
            AwsClient.s3.putObject(... ,... ,... )
})
```

- **or you can just wrap it in an object to make it available at the worker side**

Counting

```
var counter = 0
var rdd = sc.parallelize(data)

// Wrong: Don't do this!!
rdd.foreach(x => counter += x)

println("Counter value: " + counter)
```

Closure

- **Closure** = *variables and methods which must be visible for the executor (context)*

1. Spark computes the **task's closure** before execution
2. The closure is *serialized and sent to each executor*
   - the counter in the memory of the driver node but this is no longer visible to the executors
   - the executors only see the copy from the serialized closure
3. Result ⇒ counter=0 (or something else if *local[*]*)

Shared variables: Accumulators and Brodcast variables

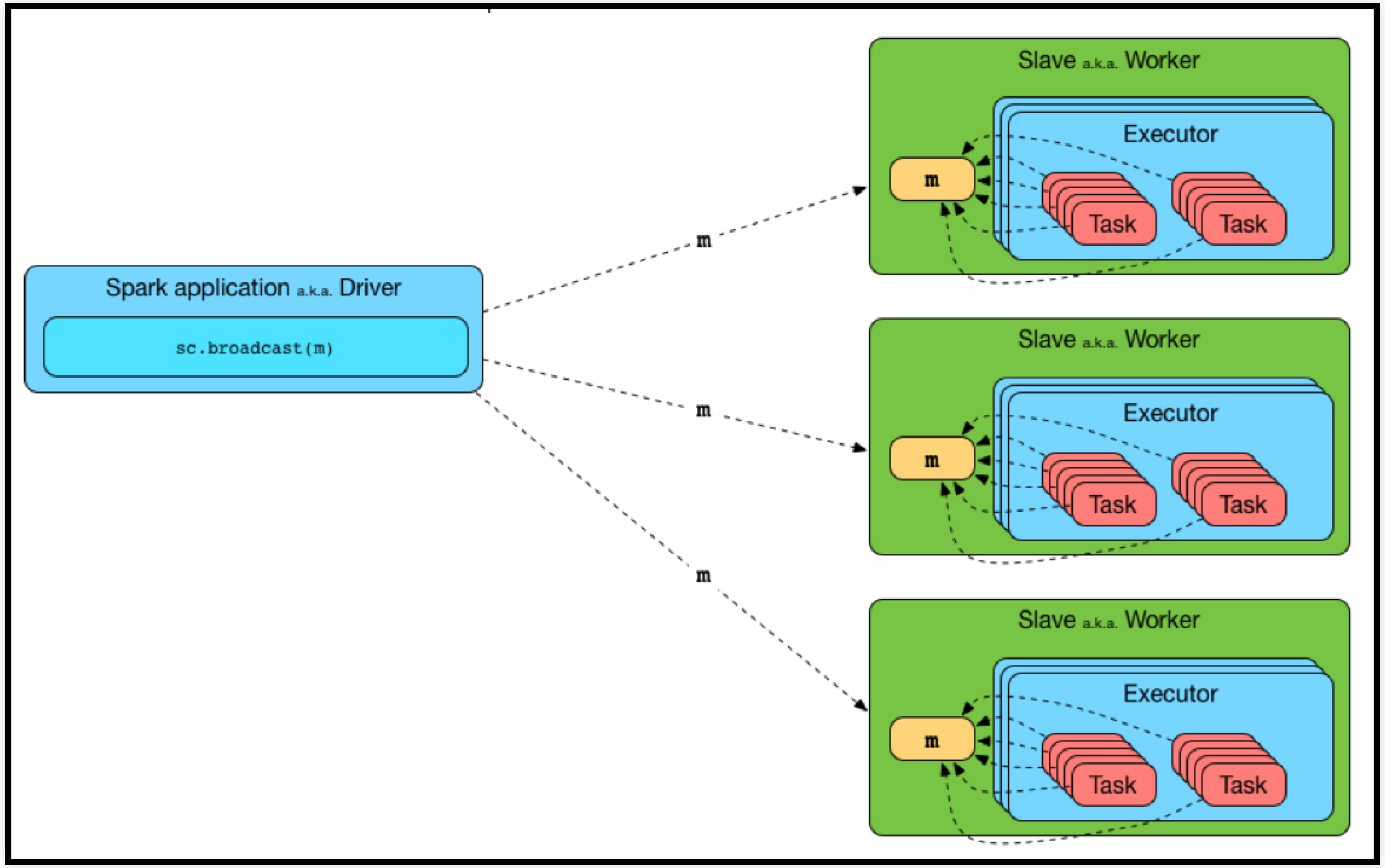
- **closures - should not be used to mutate some global state**
- **accumulators** ⇒ mechanism for safely updating a variable

```
scala> val accum = sc.longAccumulator("My Accumulator")
accum: org.apache.spark.util.LongAccumulator = LongAccumulator(id: 0, name: Some(My Accumulator), value: 0
scala> sc.parallelize(Array(1, 2, 3, 4)).foreach(x => accum.add(x))
scala> accum.value
res2: Long = 10
```

- **broadcast variables** ⇒ give every node a copy of a large input dataset (efficient manner)

```
scala> val broadcastVar = sc.broadcast(Array(1, 2, 3))
broadcastVar: org.apache.spark.broadcast.Broadcast[Array[Int]] = Broadcast(0)

scala> broadcastVar.value
res0: Array[Int] = Array(1, 2, 3)
```

Brodcast variables:



Test on cluster mode !

- most of the problems do not appear in local mode (serialization, classloading, network issue)
- something that works well in a small dataset hangs forever in production
- ⇒ **test early in cluster mode**

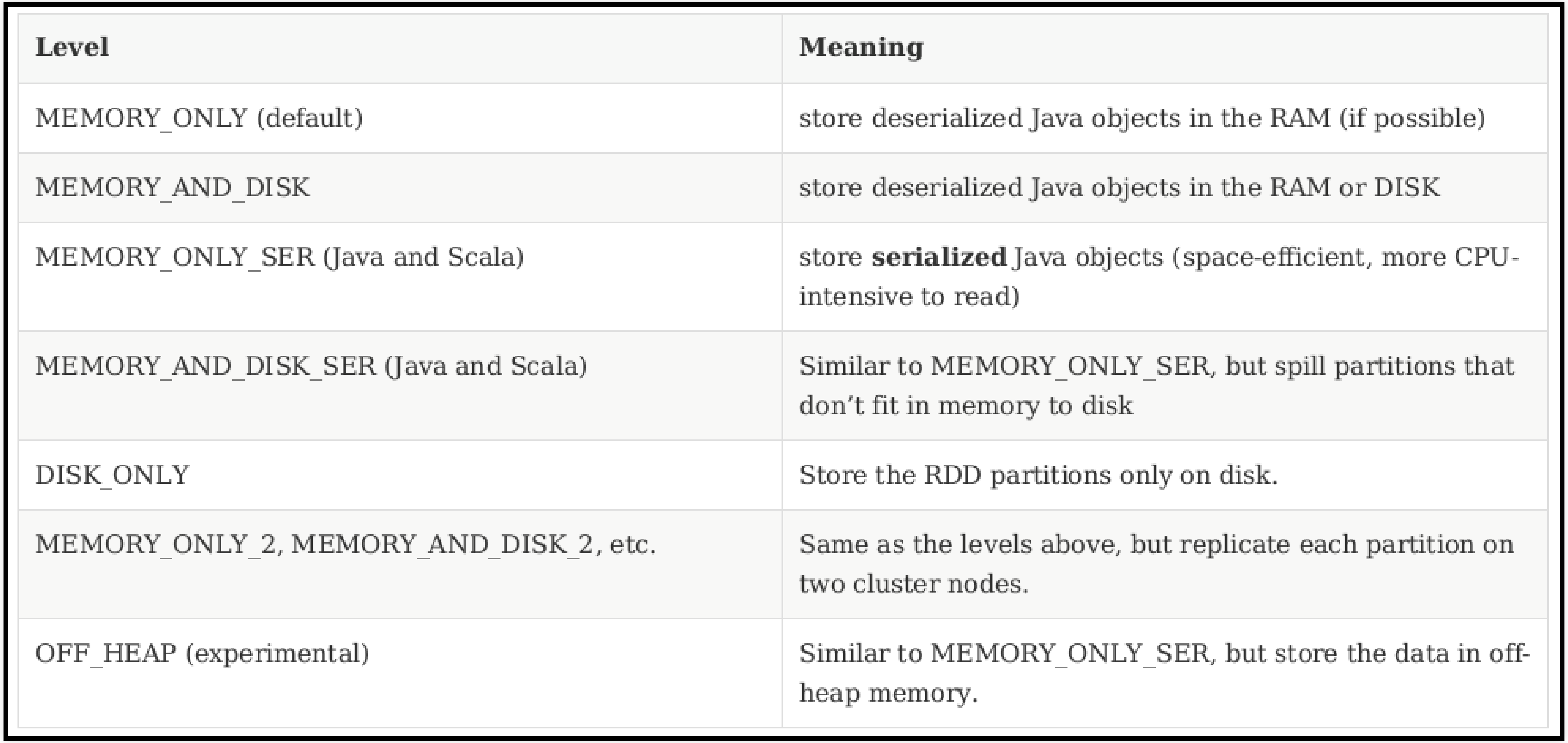
Tuning Apache Spark

- collect
- caching
- parallelism level / partitioning
- shuffles

Avoid collect

- collect ⇒ often OutOfMemoryError trying to load results back to driver
  - **not only *collect()***: *averageByKey()*, *collectByKey()*, …
- alternatives: count() before collect or sample

Caching levels



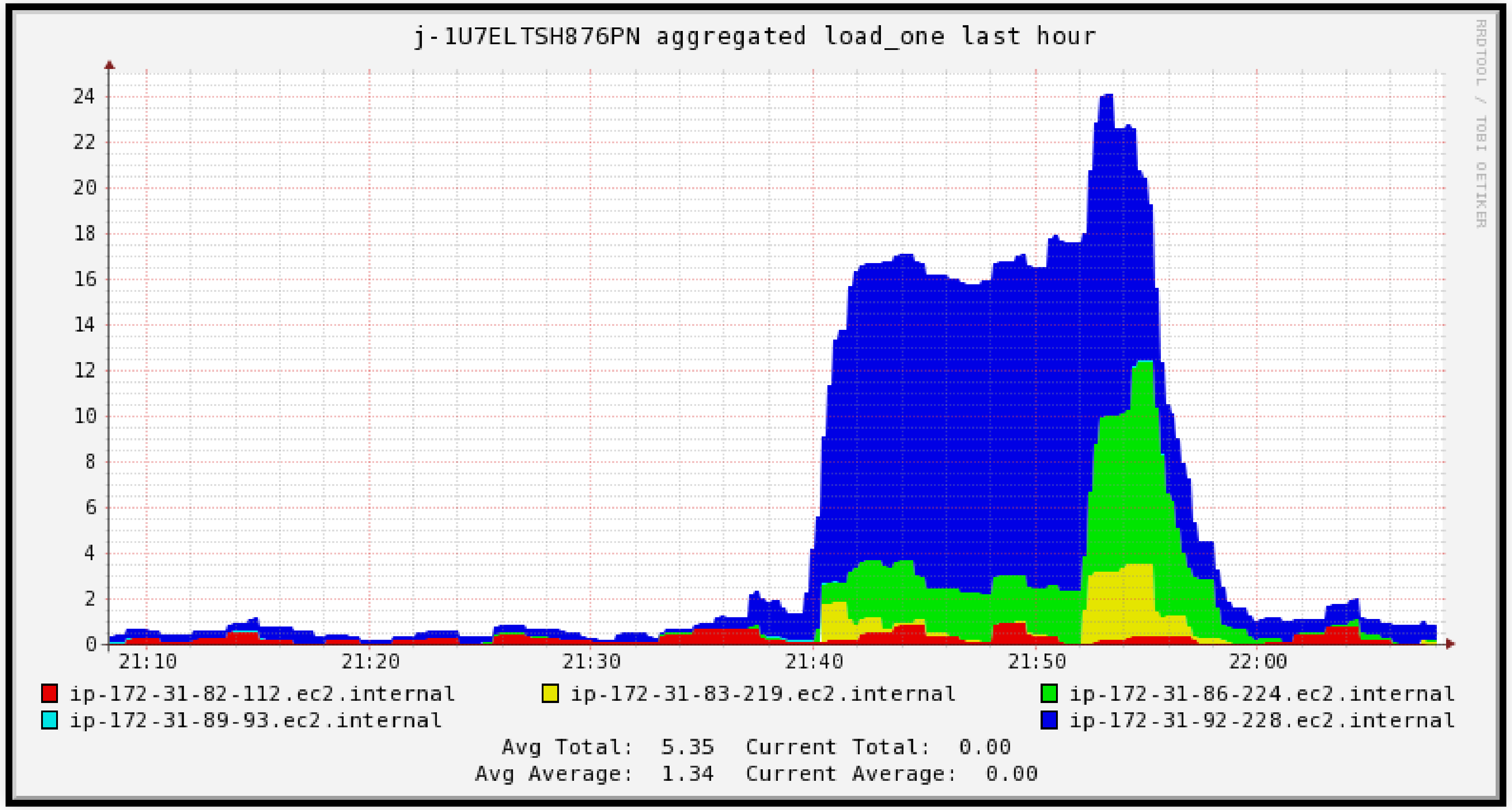
Level	Meaning
MEMORY_ONLY (default)	store deserialized Java objects in the RAM (if possible)
MEMORY_AND_DISK	store deserialized Java objects in the RAM or DISK
MEMORY_ONLY_SER (Java and Scala)	store **serialized** Java objects (space-efficient, more CPU-intensive to read)
MEMORY_AND_DISK_SER (Java and Scala)	Similar to MEMORY_ONLY_SER, but spill partitions that don't fit in memory to disk
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.
OFF_HEAP (experimental)	Similar to MEMORY_ONLY_SER, but store the data in off-heap memory.

Decide caching level

1. If your RDDs fit in RAM ⇒ **MEMORY_ONLY**
2. If not ⇒ try **MEMORY_ONLY_SER + a fast serialization library**
3. ***DISK***? ⇒ unless the DAG is expensive to compute or filters a large amount of the data !
4. ***Replicated storage levels***? ⇒ fast fault recovery (e.g. using Spark to serve requests from a web application).
    1. all the storage levels provide full fault tolerance by recomputing lost data !
    2. the replicated ones let you continue running tasks on the RDD without waiting to recompute a lost partition.

Parallelism level / partitioning (1)

- Goal: fully utilize ressources while minimizing waits and recompute lost partitions



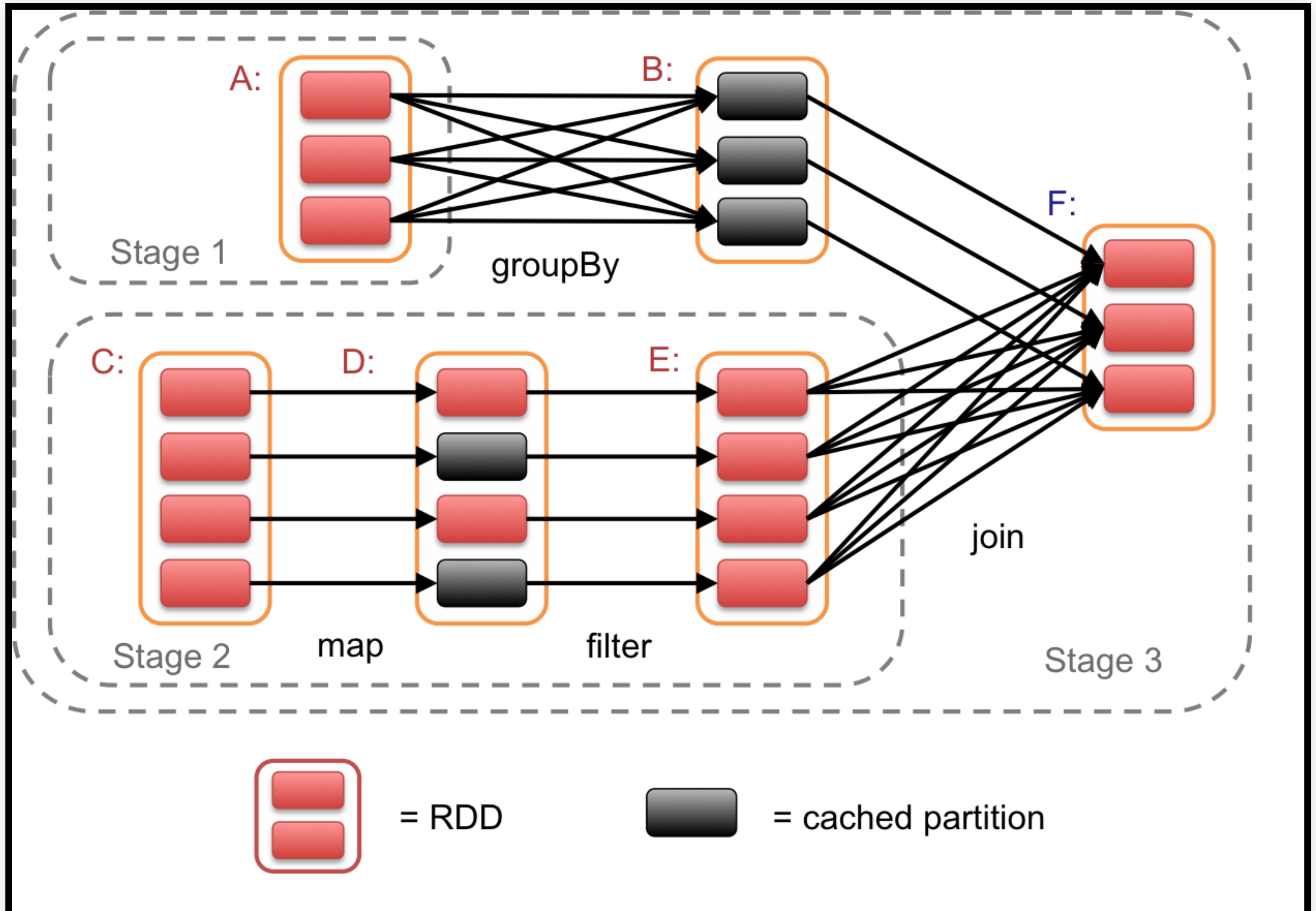
Parallelism level

- Spark automatically sets the number of “map” tasks to run on each file according to its size
  - You can control it through optional parameters ex: SparkContext.textFile("file",numpartitions)
  - For distributed “reduce” operations, such as groupByKey and reduceByKey, it uses the largest parent RDD’s number of partitions.
- *spark.default.parallelism* ⇒ recommend 2-3 tasks per CPU core in your cluster
- ***repartion(num_partitions)*** and ***coalesce(num_partitions)*** ⇒ change number of partitions

Tuning parallelism level

- a tasks should take at least 200 ms to execute (monitoring the task execution latency from the Spark UI!)
  - considerable longer tasks ⇒ increase the level of parallelism (until performance stops improving)
- OutOfMemoryError? often one of the reduce tasks in groupByKey is too large (shuffles (sortByKey, groupByKey, reduceByKey, join, etc) build a hash table within each task to perform the grouping, which can often be large)⇒ increase the level of parallelism ⇒ smaller tasks

Shuffle



Source: Matei Zaharia slide deck on Spark at Strata conference Feb 2013

Number of shuffles

- Shuffle ⇒ high CPU, network, IO cost
  - In general⇒ avoiding shuffle will make your program run faster
  - Exception: data arrives in a few large unsplittable files
    - large numbers of records in each partition
    - not enough partitions to take advantage of all the available cores
    - **repartition with a high number of partitions** (⇒shuffle) ⇒ leverage more of the cluster’s CPU.

Plan

1. RDD execution review
2. Best practices
3. **Dataframes & Dataset API**
4. Spark notebooks
5. TP2 Data exploration with the Dataframe API

RDD API

- Most data is structured (JSON, CSV, Avro, Parquet, Hive ...)
  - **Programming RDDs inevitably ends up with a lot of tuples (_1, _2, ...)**
- Functional transformations (e.g. map/reduce) are not as intuitive

```
val myRDD = sc.parallelize(persons)
myRDD.map(x=>(x.dept,(x.age,1)))
     .reduceByKey((a,b)=>(a._1+b._1,a._2+b._2))
     .map(p=>(p._1,p._2._1/p._2._2))
     .collect
```

RDD vs SQL

```
val myRDD = sc.parallelize(persons)
myRDD.map(x=>(x.dept,(x.age,1)))
     .reduceByKey((a,b)=>(a._1+b._1,a._2+b._2))
     .map(p=>(p._1,p._2._1/p._2._2))
     .collect
```

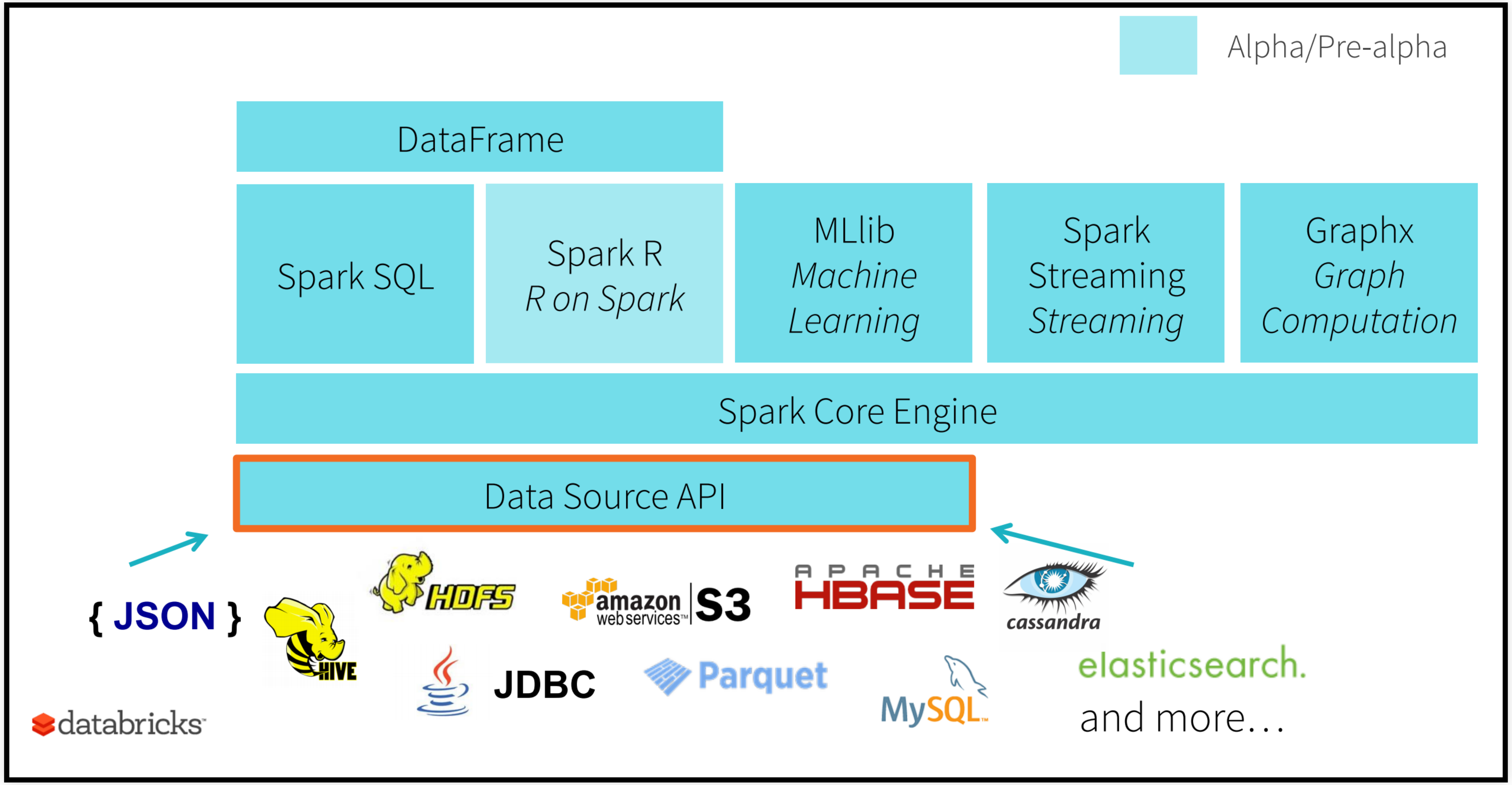
```
--- SQL LIKE
SELECT avg(age) FROM pdata
GROUP BY dept;
```

RDD vs DataFrames

```
val myRDD = sc.parallelize(persons)
myRDD.map(x=>(x.dept,(x.age,1)))
     .reduceByKey((a,b)=>(a._1+b._1,a._2+b._2))
     .map(p=>(p._1,p._2._1/p._2._2))
     .collect
```

```
data.groupBy("dept").avg("age")
```

Introducing DataFrames and SparkSQL



Spark Dataframes

- **RDD with schema**
    - Distributed collection of data grouped into named columns
- Domain-specific functions designed for common tasks
    - Metadata
    - Relational data processing: project, filter, aggregation, join

Spark Dataframes Examples

```
// Create from RDD
val dfFromRDD = myRDD.df

// Create from JSON
val df = sparkSession.read.json("examples/src/main/resources/people.json")

// Show the content of the DataFrame
df.show()
// age  name
// 30   Andy
// 19   Justin

// Print the schema in a tree format
df.printSchema()
// root
// |-- age: long (nullable = true)
// |-- name: string (nullable = true)
```

Spark Dataframes : Column selection

```
// Select only the "name" column
df.select("name").show() // implicit conversion from string to Column
// name
// Michael
// Andy
// Justin

//Equivalent column selection
df.select(df("name")).show() // apply function on dataframe

df.select(col("name")).show() //using the col function

df.select(df.col("name")).show() //using the col method

df.select($"name").show // easy scala shorcut
```

Spark Dataframes: simple column computations

```
// Select everybody, but increment the age by 1
df.select(df("name"), df("age") + 1).show()
// name    (age + 1)
// Michael null
// Andy    31
// Justin  20
```

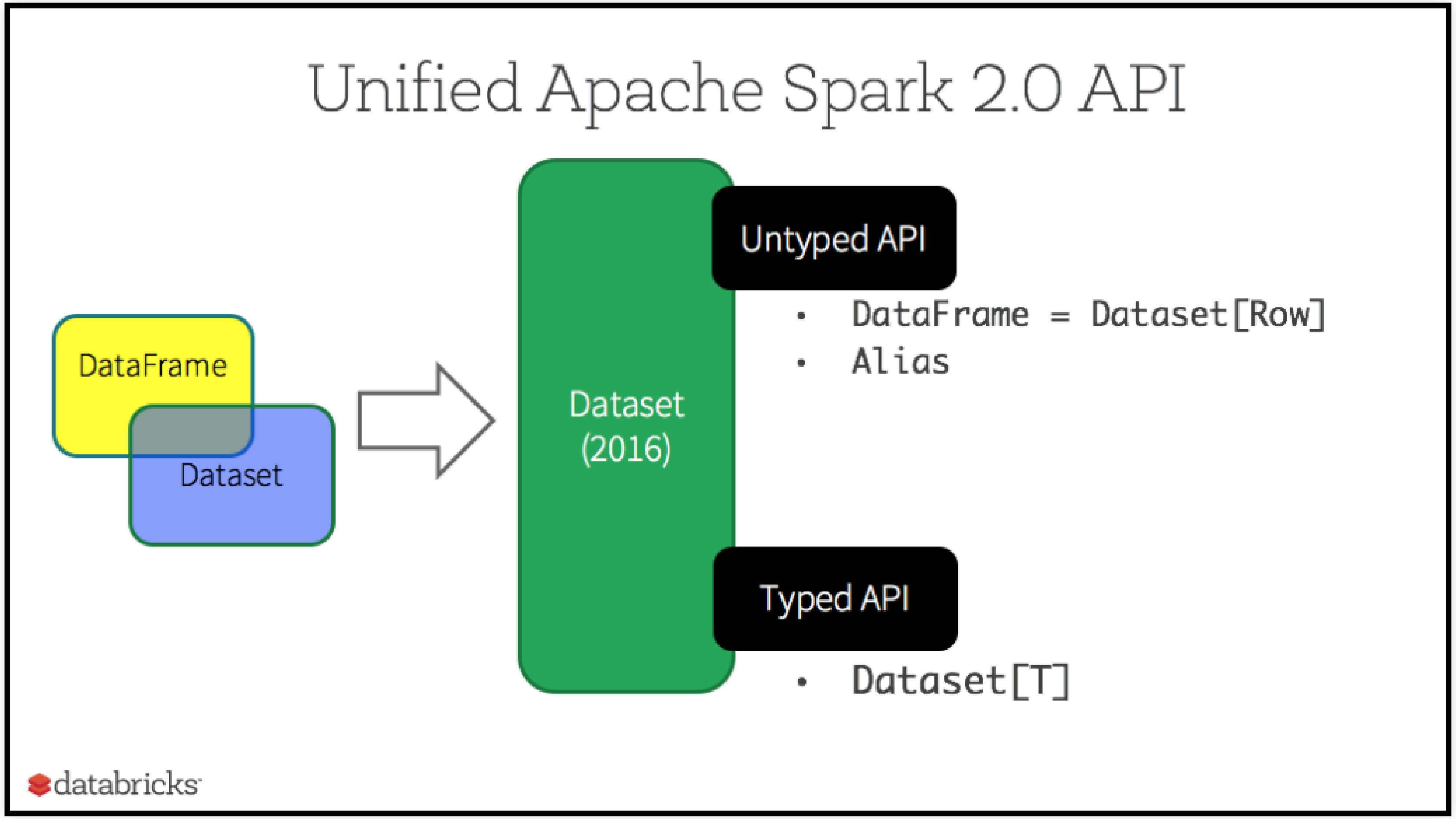
Spark Dataframes Filtering

```
// Select people older than 21
df.filter(df("age") > 21).show()
// age name
// 30  Andy

// Count people by age
df.groupBy("age").count().show()
// age  count
// null 1
// 19   1
// 30   1
```

Complex example

```
val accidents = usagers
    .filter($"grav" == 2 ) // on filtre les victimes
    .groupBy("Num_Acc") // on groupe sur l'id de l'accident
    .count() // on compte le nombre des victimes
    .sort(desc("count"))// on trie par le nombre des victimes
    .limit(3)// on garde uniquement les 3 accidents les plus meurtirers
    .withColumnRenamed("count","nb_victimes")
    .join(caracteristiques, usagers("Num_acc")== caracteristiques("Num_Acc"))// on fait le jointure avec l
    .join(lieux, usagers("Num_acc")== lieux("Num_Acc"))// on fait le jointure avec les lieux
    .join(vehicules, usagers("Num_Acc")== vehicules("Num_Acc")) // on fait la jointure avec les vehicules
    .groupBy(caracteristiques("an"),caracteristiques("mois"),
            caracteristiques("jour"),caracteristiques("adr"),
            caracteristiques("dep"), caracteristiques("atm"),
            $"nb_victimes") // on groupe sur les colonnes qui nous interessent
    .count().withColumnRenamed("count","nb_vehicules") // on compte le nombre des vehicules
```

Dataframe vs Datasets



- DataFrame = Dataset[Row]
- Alias

- Dataset[T]

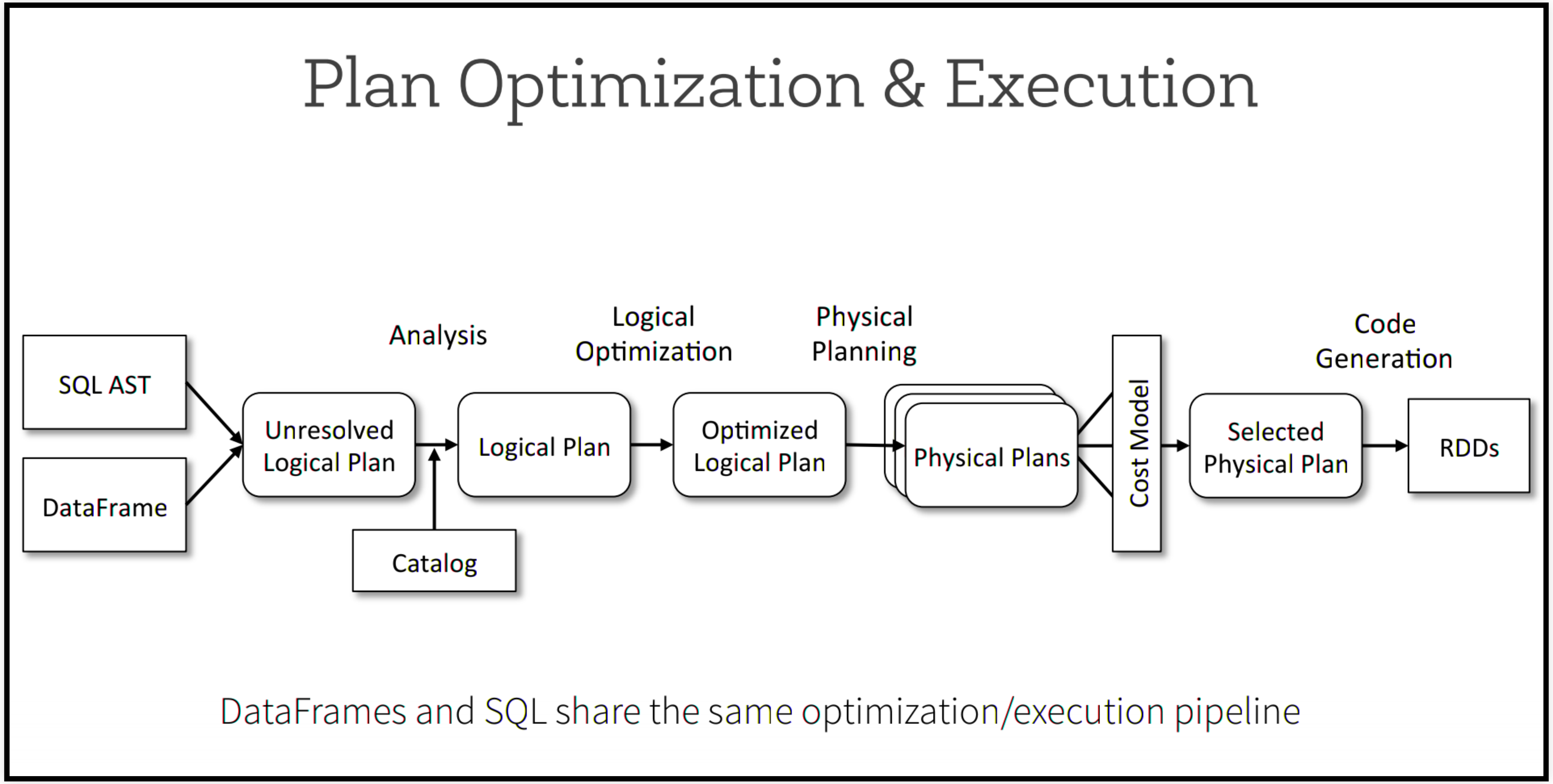
Spark Datasets

- type safe generalisation of Dataframes
- all the methods available to Dataframes are the same
  - type Dataframe = Dataset[Row]
- use native types instead of Row

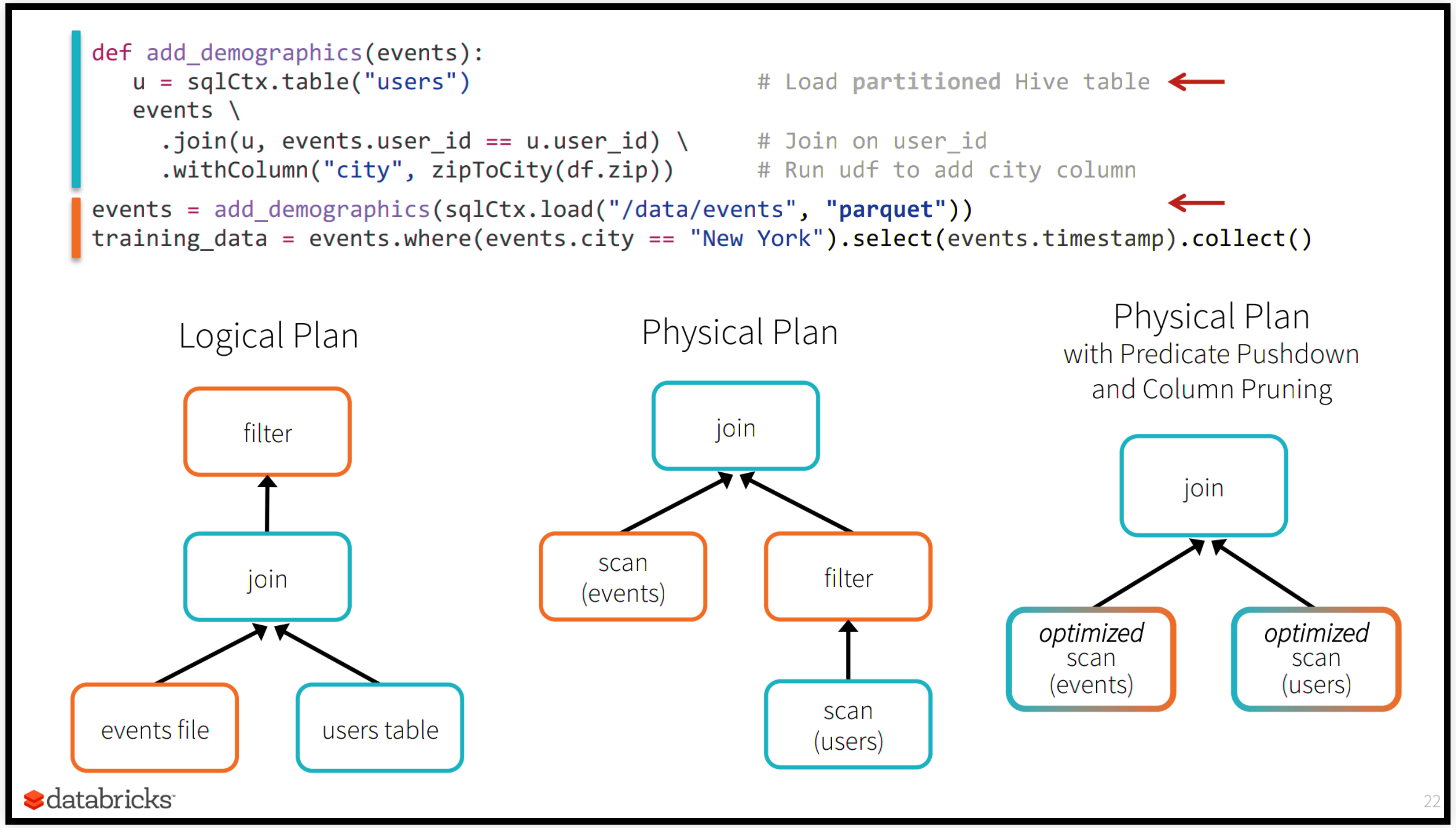
Spark Dataset example

```scala
case class University(name: String, numStudents: Long, yearFounded: Long)

val schools = sqlContext.read.json("/schools.json").as[University]

schools.map(s => s"${s.name} is ${2015 – s.yearFounded} years old")
```

SparkSQL/Dataframes execution model



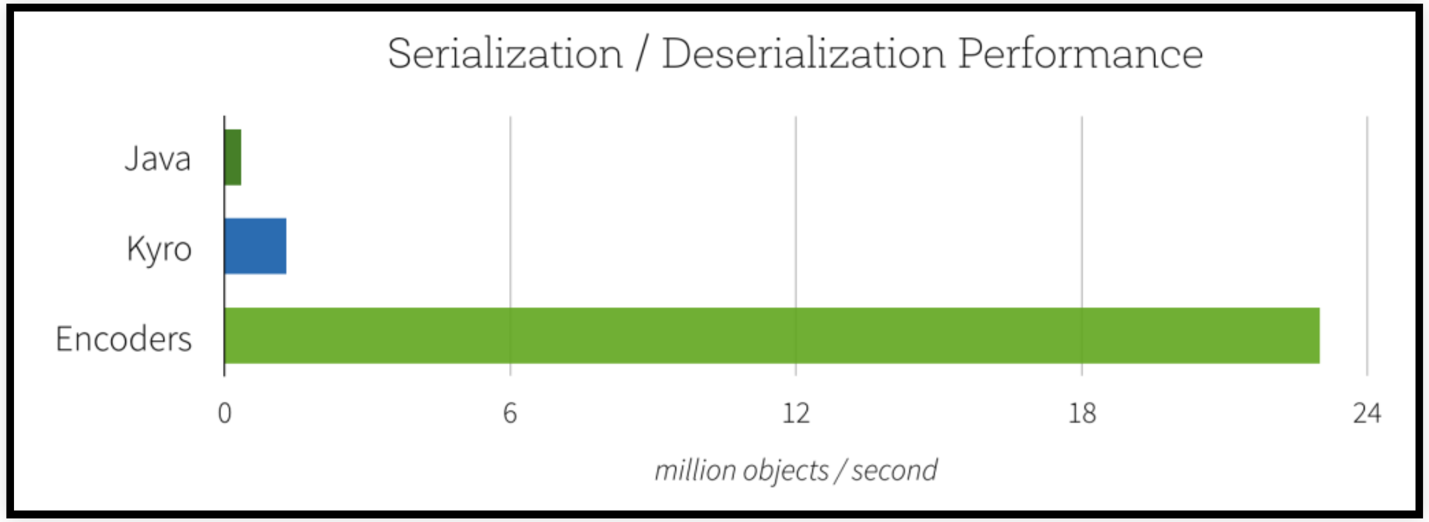
SparkSQL/Dataframe optimisations

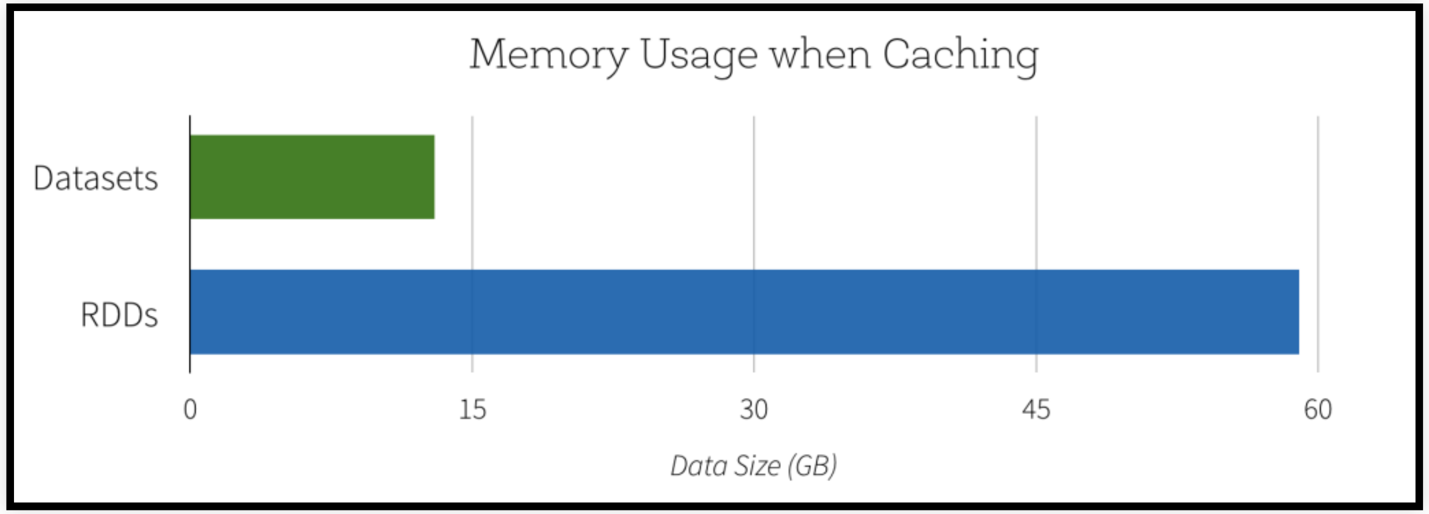


```
def add_demographics(events):
   u = sqlCtx.table("users")                     # Load partitioned Hive table
   events \
     .join(u, events.user_id == u.user_id) \     # Join on user_id
     .withColumn("city", zipToCity(df.zip))      # Run udf to add city column
events = add_demographics(sqlCtx.load("/data/events", "parquet"))
training_data = events.where(events.city == "New York").select(events.timestamp).collect()
```

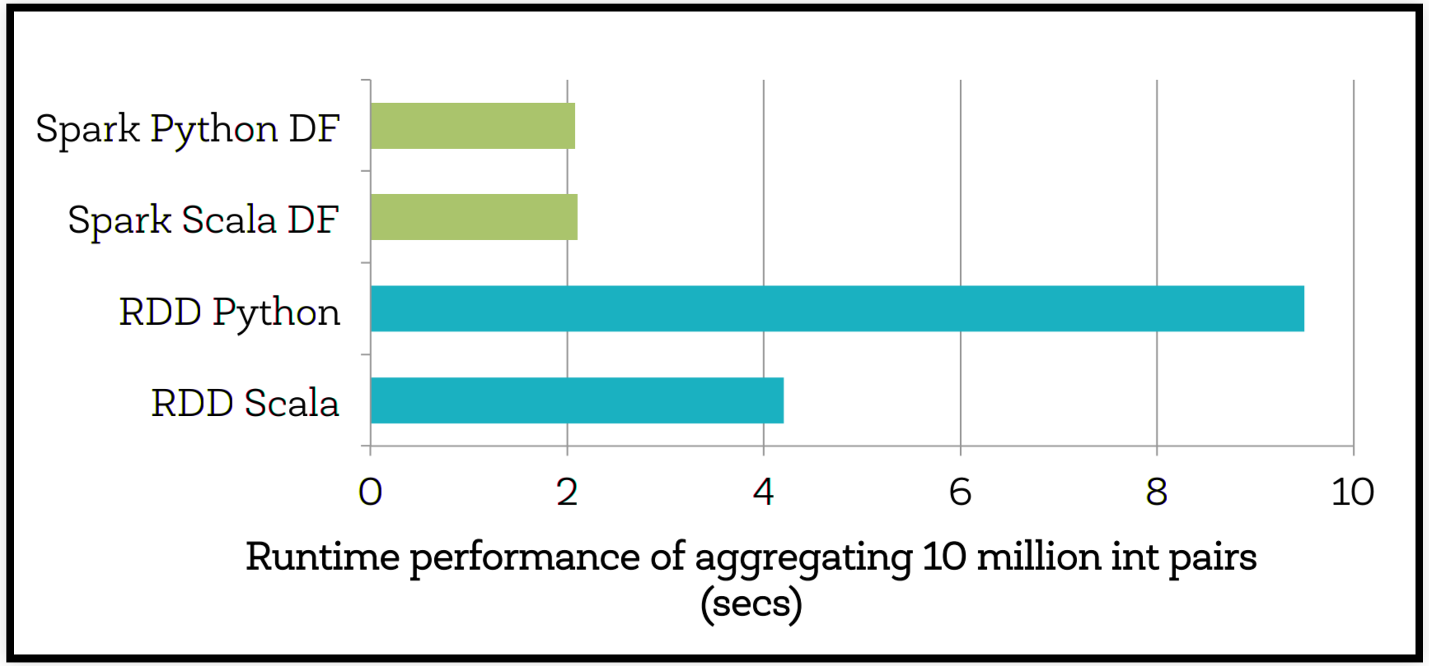
Source: http://www.slideshare.net/databricks/spark-sqlsse2015public

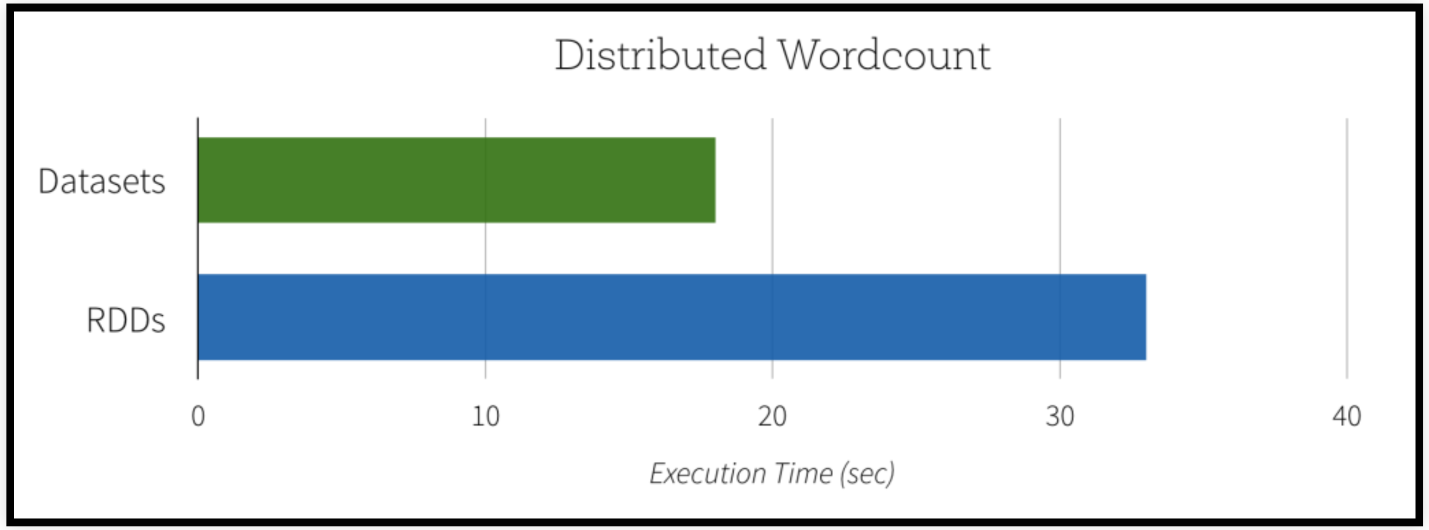
SparkSQL/Dataframe optimisations



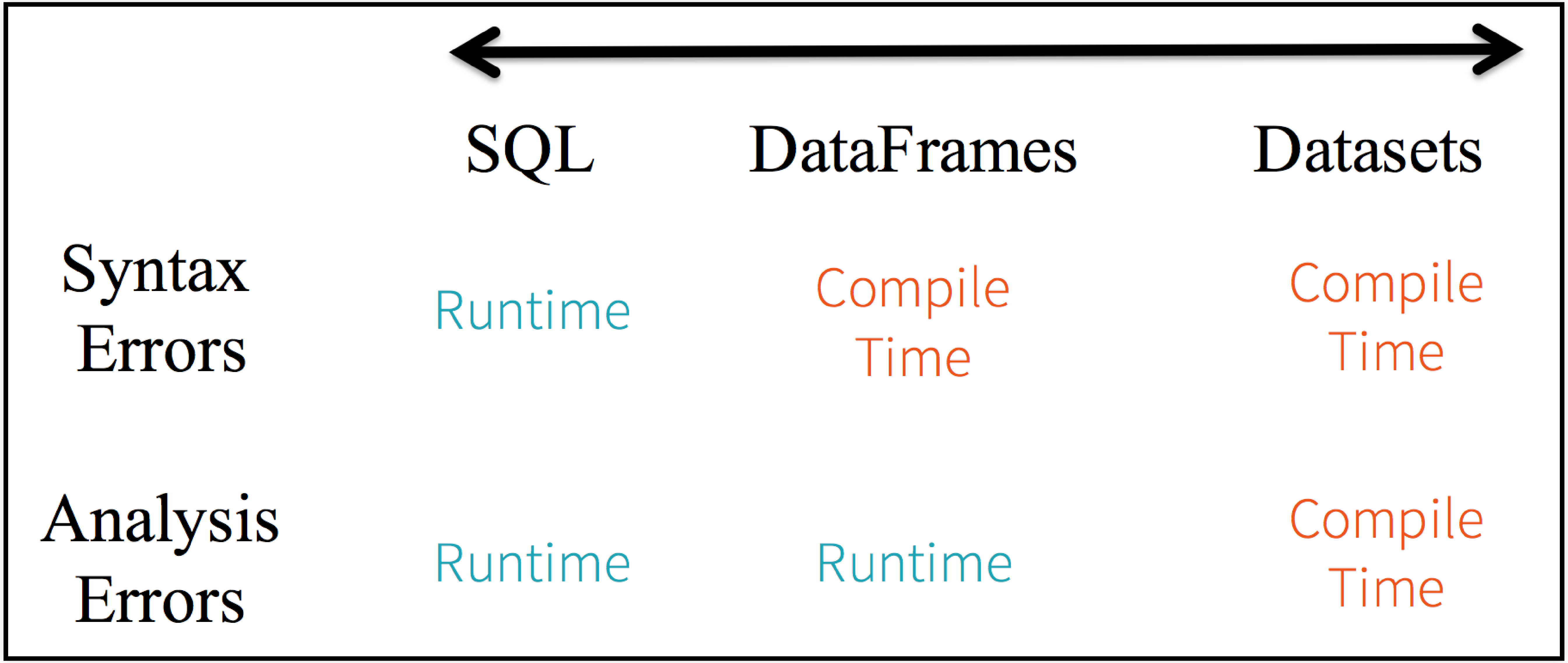


SparkSQL/Dataframe performance




Distributed Wordcount
Datasets
RDDs
0
10
20
30
40
Execution Time (sec)

SparkSQL vs Dataframe vs Datasets



	SQL	DataFrames	Datasets
Syntax Errors	Runtime	Compile Time	Compile Time
Analysis Errors	Runtime	Runtime	Compile Time

Frameless (BETA)

- new Spark API, called **TypedDataset**
  - Typesafe columns referencing (no more runtime errors when accessing non-existing columns)
  - Customizable, typesafe encoders (if a type does not have an encoder, it should not compile)
  - Enhanced type signature for built-in functions (if you apply an arithmetic operation on a non-numeric column, you get a compilation error)
  - Typesafe casting and projections

Frameless (1)

```scala
import spark.implicits._

// Our example case class Foo acting here as a schema
case class Foo(i: Long, j: String)

// Assuming spark is loaded and SparkSession is bind to spark
val initialDs = spark.createDataset( Foo(1, "Q") :: Foo(10, "W") :: Foo(100, "E") :: Nil )

// Assuming you are on Linux or Mac OS
initialDs.write.parquet("/tmp/foo")

val ds = spark.read.parquet("/tmp/foo").as[Foo]
// ds: org.apache.spark.sql.Dataset[Foo] = [i: bigint, j: string]
```

Frameless (2)

```
ds.show()
// +---+---+
// |  i|  j|
// +---+---+
// |  1|  Q|
// | 10|  W|
// |100|  E|
// +---+---+
//
//The value ds holds the content of the initialDs read from a parquet file. Let's try to only use field i

// Using a standard Spark TypedColumn in select()
val filteredDs = ds.filter($"i" === 10).select($"i".as[Long])
// filteredDs: org.apache.spark.sql.Dataset[Long] = [i: bigint]

filteredDs.show()
// +---+
// |  i|
```

Frameless (3)

```
// Using a standard Spark TypedColumn in select()
val filteredDs = ds.filter($"i" === 10).select($"i".as[Long])
// filteredDs: org.apache.spark.sql.Dataset[Long] = [i: bigint]

filteredDs.show()
// +---+
// |  i|
// +---+
// | 10|
// +---+
//

filteredDs.explain()
// == Physical Plan ==
// *Project [i#1771L]
// +- *Filter (isnotnull(i#1771L) && (i#1771L = 10))
//    +- *FileScan parquet [i#1771L] Batched: true, Format: Parquet, Location: InMemoryFileIndex[file:/tmp
```

Frameless (4)

```
//Unfortunately, this syntax is not bulletproof: it fails at run-time if we try to access a non existing c

scala> ds.filter($"i" === 10).select($"x".as[Long])
org.apache.spark.sql.AnalysisException: cannot resolve '`x`' given input columns: [i, j];;
'Project ['x]
+- Filter (i#1771L = cast(10 as bigint))
   +- Relation[i#1771L,j#1772] parquet
```

Frameless (5)

```
//There are two things to improve:
// * we would want to avoid the as[Long] casting that we are required to type for type-safety.
// * we want a solution where reference to a non existing column name fails at compilation time

// Proposition

ds.filter(_.i == 10).map(_.i).show()
// +------+
// |value|
// +------+
// |   10|
// +------+
//----

// The two closures in filter and map are functions that operate on Foo and the compiler will helps us cap
   //
   //scala> ds.filter(_.i == 10).map(_.x).show()
   //<console>:20: error: value x is not a member of Foo
```

Frameless (6)

```
// Unfortunately, this syntax does not allow Spark to optimize the code.

ds.filter(_.i == 10).map(_.i).explain()
// == Physical Plan ==
// *SerializeFromObject [input[0, bigint, false] AS value#1805L]
// +- *MapElements <function1>, obj#1804: bigint
//    +- *Filter <function1>.apply
//       +- *DeserializeToObject newInstance(class $line14.$read$$iw$$iw$$iw$$iw$Foo), obj#1803: $line14.$
//          +- *FileScan parquet [i#1771L,j#1772] Batched: true, Format: Parquet, Location: InMemoryFileIn
```

Frameless (7)

```
import frameless.TypedDataset
import frameless.syntax._

val fds = TypedDataset.create(ds)
// fds: frameless.TypedDataset[Foo] = [i: bigint, j: string]

fds.filter(fds('i) === 10).select(fds('i)).show().run()
// +---+
// | _1|
// +---+
// | 10|
// +---+
//
```

Frameless (8)

```
//And the optimized Physical Plan:

fds.filter(fds('i) === 10).select(fds('i)).explain()
// == Physical Plan ==
// *Project [i#1771L AS _1#1876L]
// +- *Filter (isnotnull(i#1771L) && (i#1771L = 10))
//    +- *FileScan parquet [i#1771L] Batched: true, Format: Parquet, Location: InMemoryFileIndex[file:/tmp

//And the compiler is our friend.
scala> fds.filter(fds('i) === 10).select(fds('x))
<console>:24: error: No column Symbol with shapeless.tag.Tagged[String("x")] of type A in Foo
       fds.filter(fds('i) === 10).select(fds('x))
                                            ^
```

Conclusions

- To understand Spark you need to understand RDDs
- Use Dataframes/Datasets/SparkSQL for the optimisations
- Use RDD for custom optimisations
- Watch for data shuffles (use the Spark UI)
- Use the documentation: http://spark.apache.org/docs/latest/programming-guide.html

TP2 - Spark Dataframes et Datasets

- work in spark-notebooks http://andreiarion.github.io/TP6_TPSpark_Dataframes.html

Spark notebooks

- Visual tools for data exploration (@scale)
  - syntax highlight, external tool integration

Spark notebooks

- Apache Zeppelin
  - custom interpretors (Scala,Python,R,SQL,Markdown...)
  - Spark 2.0.2 not yet supported
- Spark-Notebook
  - bleeding edge, most up-to-date, reactive
  - few bugs, scala only ex. tp !
- Databricks cloud
  - nothing to install, easy to use, AWS integration, stable, supported
  - cost! (DBU + subscription)

TP2 - Spark Dataframes et Datasets

- work in spark-notebooks http://andreiarion.github.io/TP6_TPSpark_Dataframes.html

Ressources:

- Learning Spark book
- Mastering Apache Spark
- Spark tuning
- Frameless vs Datasets
- Avoiding spark pitfalls at scale
- SMACK