Apache Spark

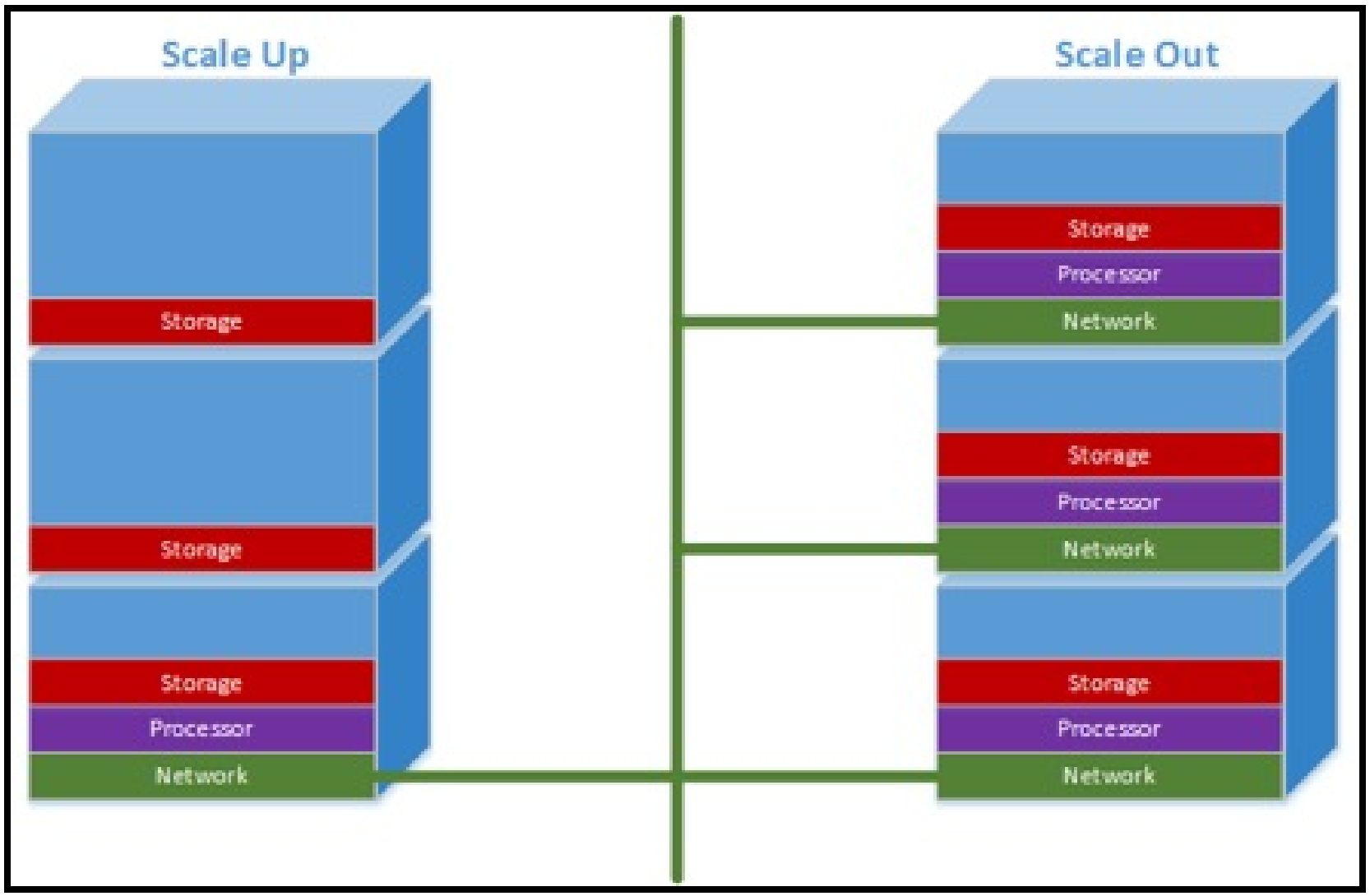
Plan

1. **Context: Distributed programming**
2. Spark data model
3. Spark programming
4. Spark execution model
5. Spark cluster
6. Spark ecosystem
7. Spark vs MapReduce
8. (Spark on AWS ES Cluster)

Plan

- Distributed programming models
  - Data flow models (MapReduce, Apache Spark)

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

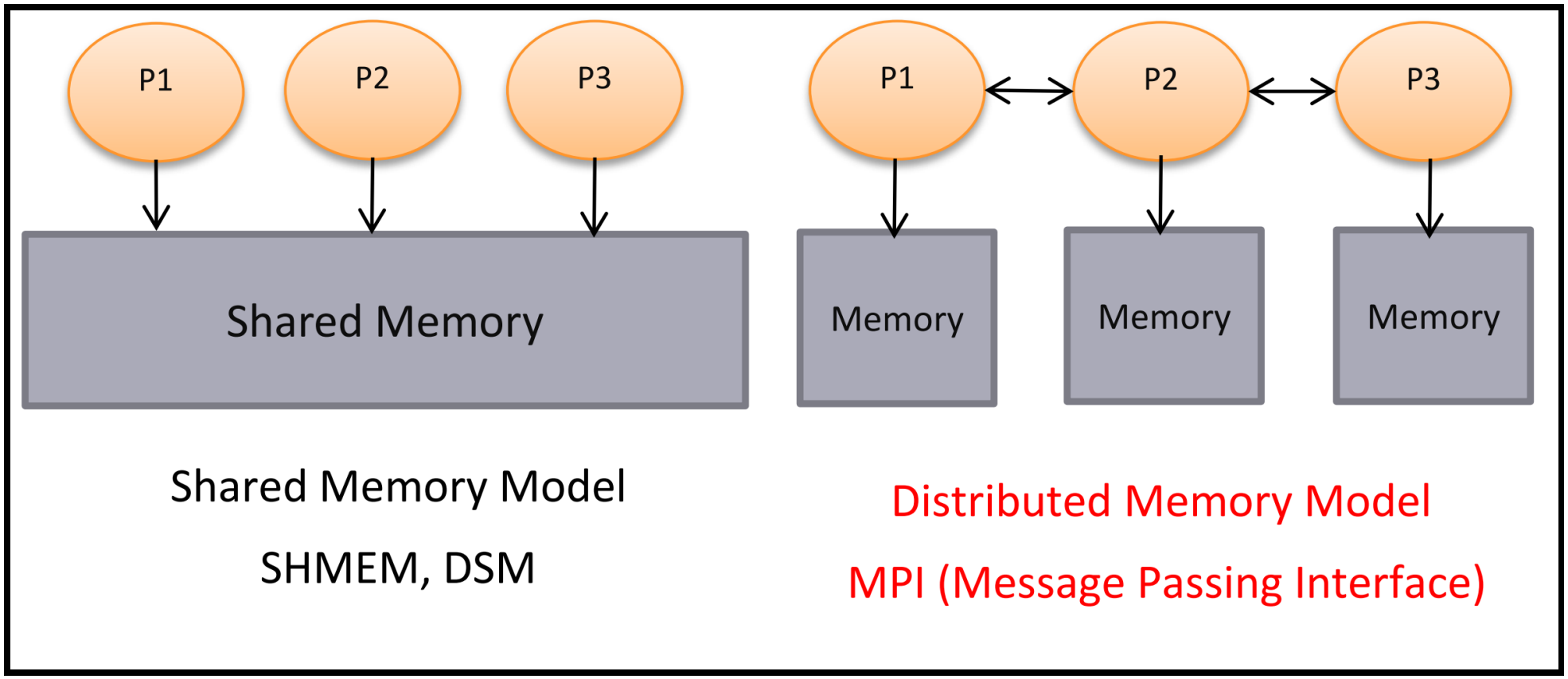
- Databases: used as a shared ressource
  - data persistence
  - standardized access to data (API / query language)
  - guaranteed properties (CAP) via sharding and replication
- Distributed programming:
  - express general computation
  - pipelines of tasks

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 / 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

TOP 10 Sites for November 2015

For more information about the sites and systems in the list, click on the links or view the complete list.

RANK	SITE	SYSTEM	CORES	RMAX (TFLOP/S)	RPEAK (TFLOP/S)	POWER (KW)
1	National Super Computer Center in Guangzhou China	**Tianhe-2 (MilkyWay-2)** - TH-IVB-FEP Cluster, Intel Xeon E5-2692 12C 2.200GHz, TH Express-2, Intel Xeon Phi 31S1P NUDT	3,120,000	33,862.7	54,902.4	17,808
2	DOE/SC/Oak Ridge National Laboratory United States	**Titan** - Cray XK7 , Opteron 6274 16C 2.200GHz, Cray Gemini interconnect, NVIDIA K20x Cray Inc.	560,640	17,590.0	27,112.5	8,209
3	DOE/NNSA/LLNL United States	**Sequoia** - BlueGene/Q, Power BQC 16C 1.60 GHz, Custom IBM	1,572,864	17,173.2	20,132.7	7,890
4	RIKEN Advanced Institute for Computational Science (AICS) Japan	K computer, SPARC64 VIIIfx 2.0GHz, Tofu interconnect Fujitsu	705,024	10,510.0	11,280.4	12,660
5	DOE/SC/Argonne National Laboratory United States	**Mira** - BlueGene/Q, Power BQC 16C 1.60GHz, Custom IBM	786,432	8,586.6	10,066.3	3,945

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);
```

”Traditional” distributed programming

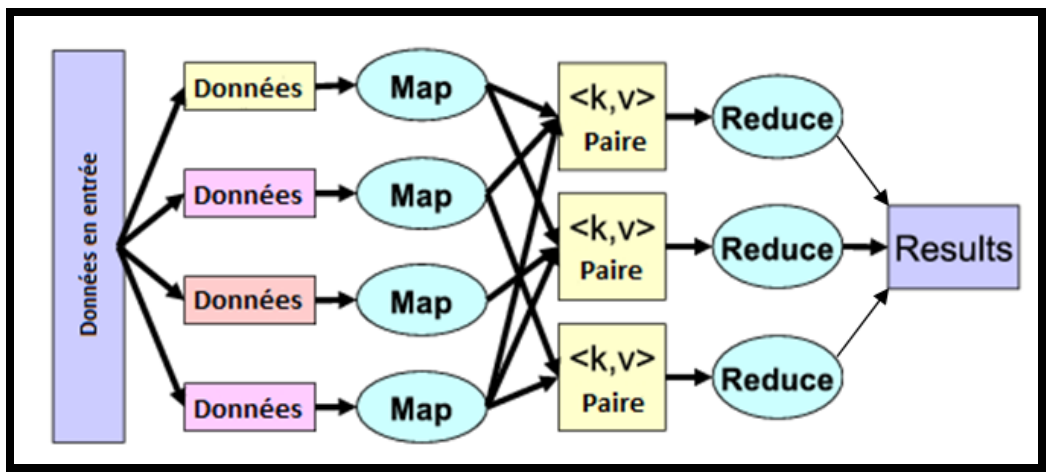
- Low level
  - manual data decomposition
  - manual load balancing
- Very difficult to do at scale:
  - How to efficiently split data/code across nodes?
    - Must consider network & data locality
  - How to deal with failures? (inevitable at scale)

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

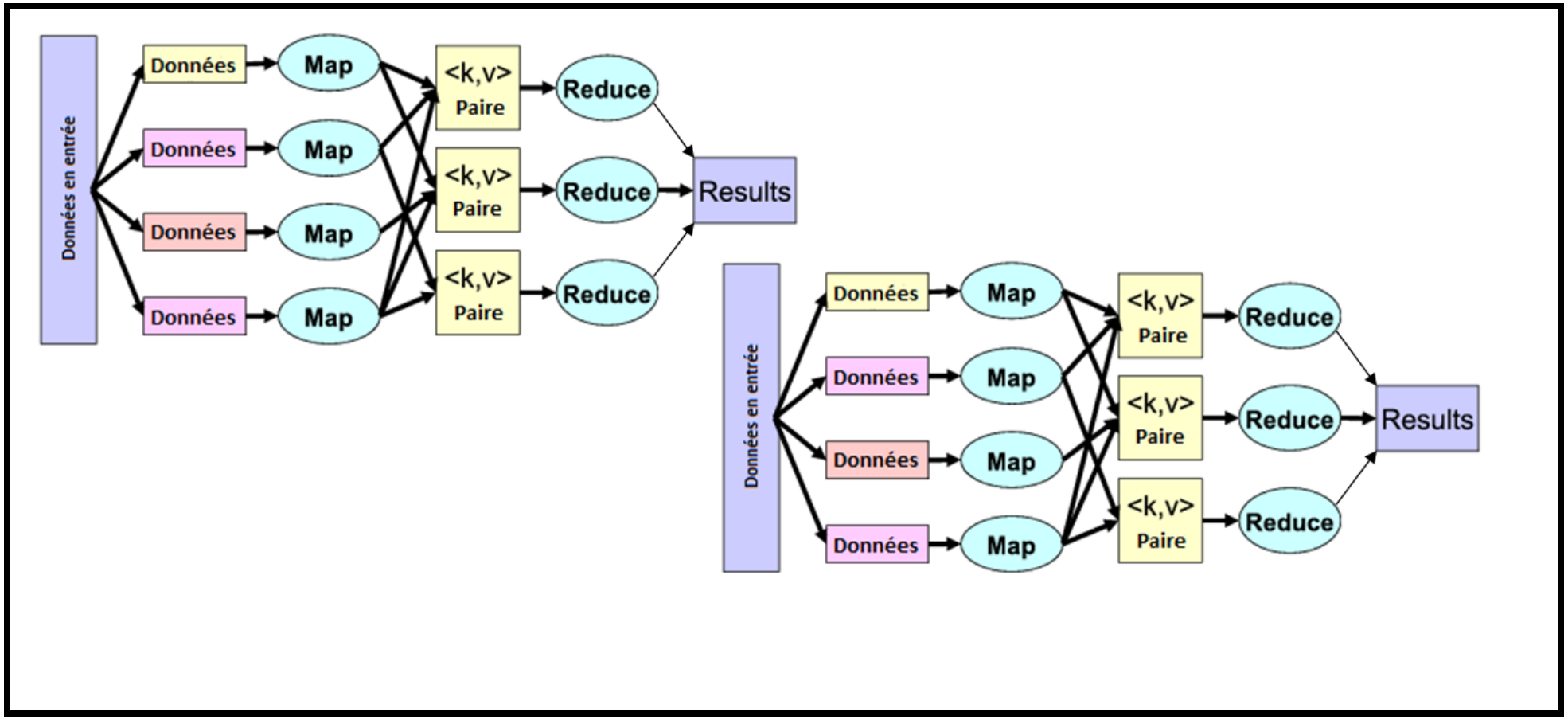


© http://commons.wikimedia.org/wiki/File:Mapreduce.png

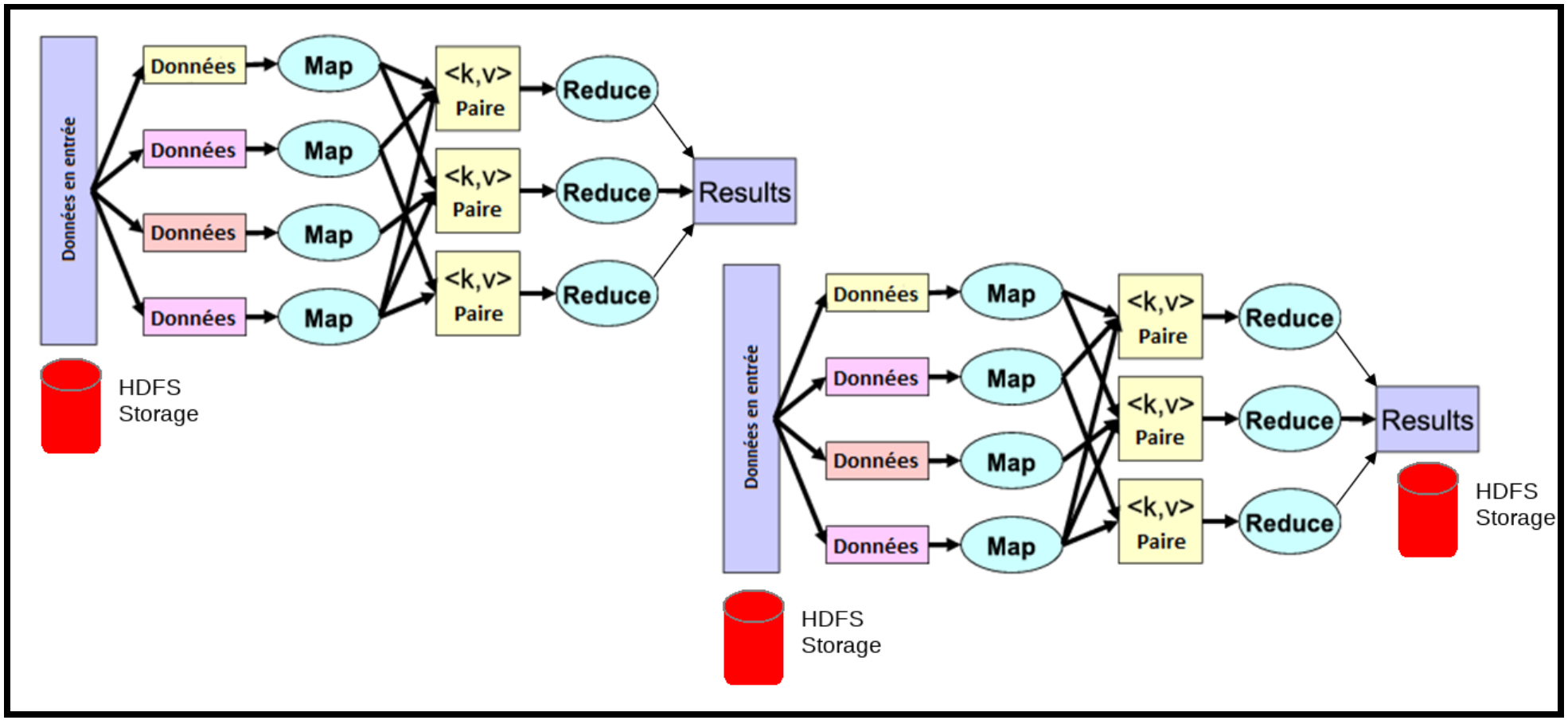
Map Reduce ⇒ **strong points:**

- Simpler 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



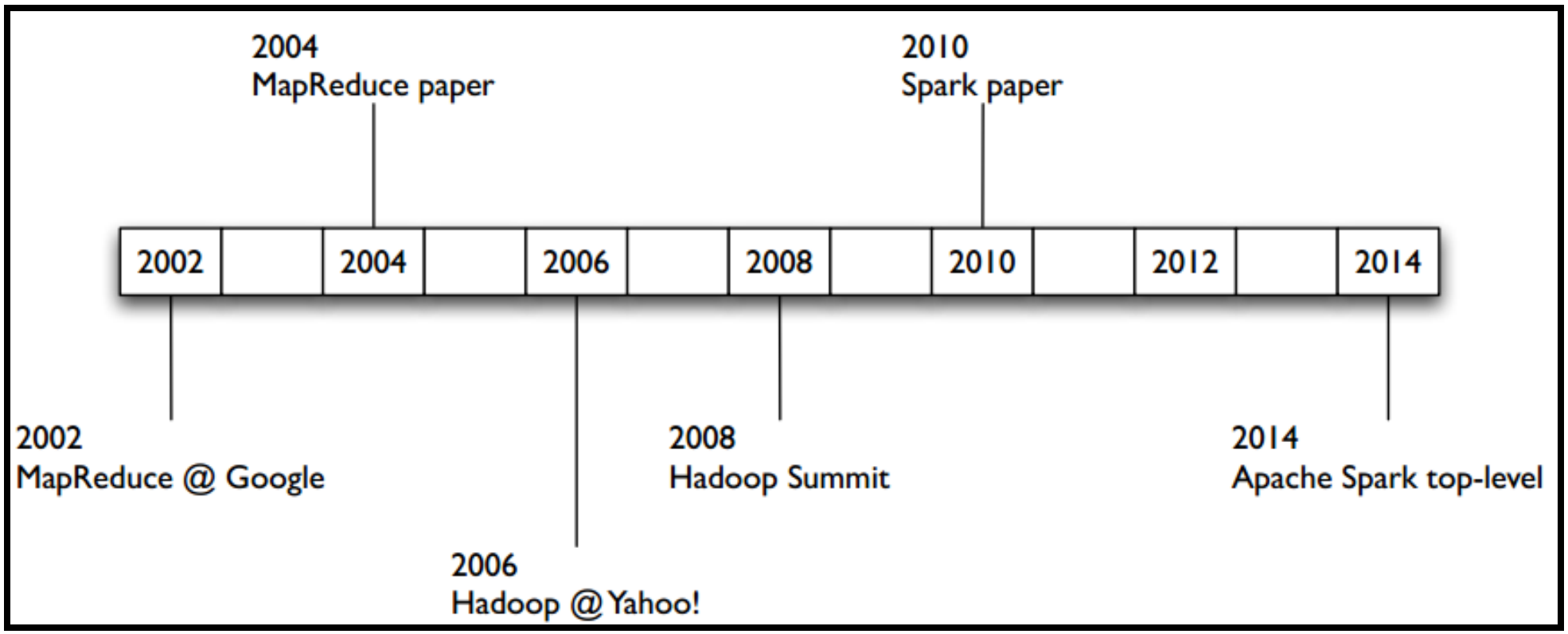
Map Reduce iterations



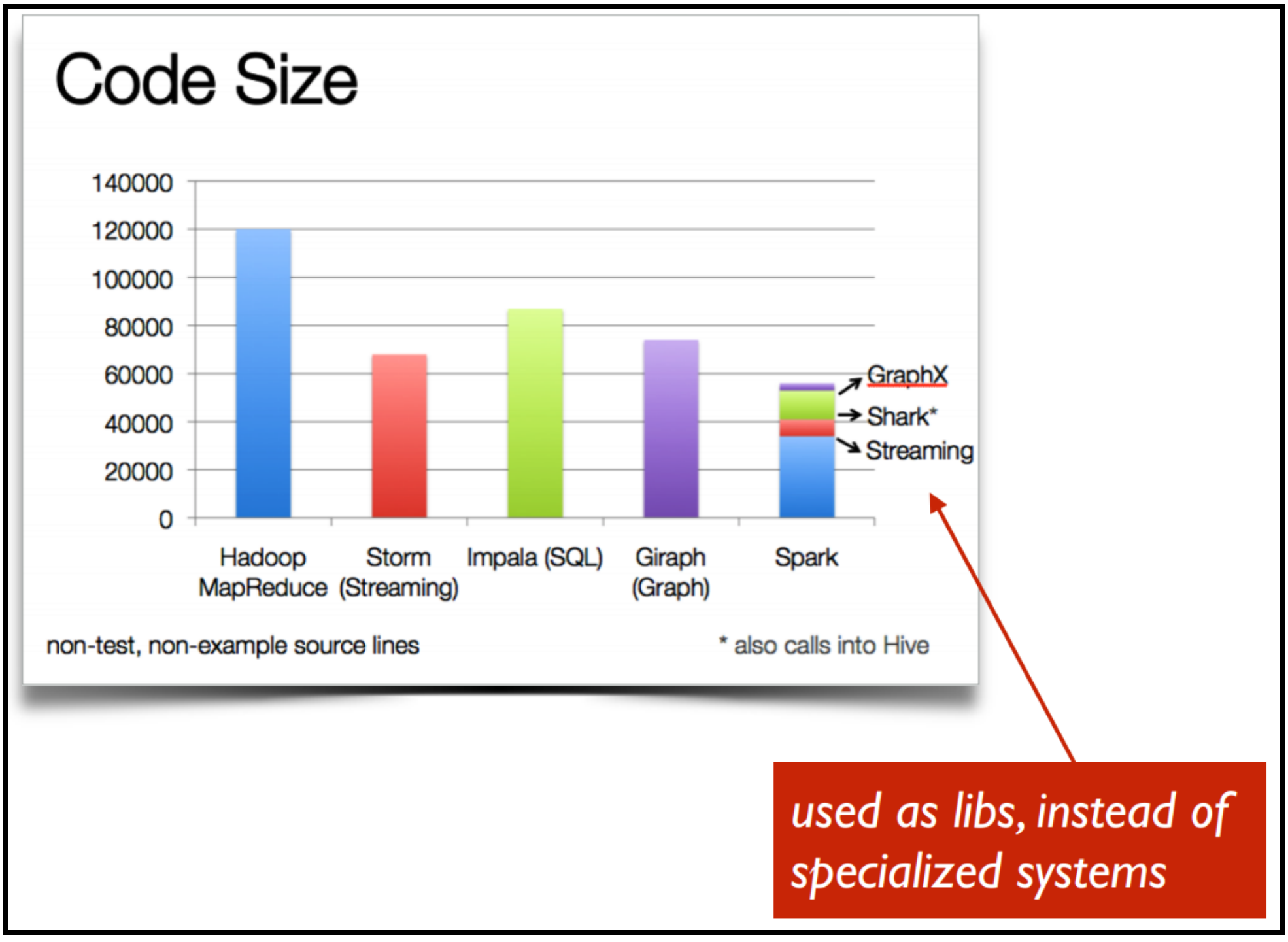
Map Reduce ⇒ **weak points:**

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

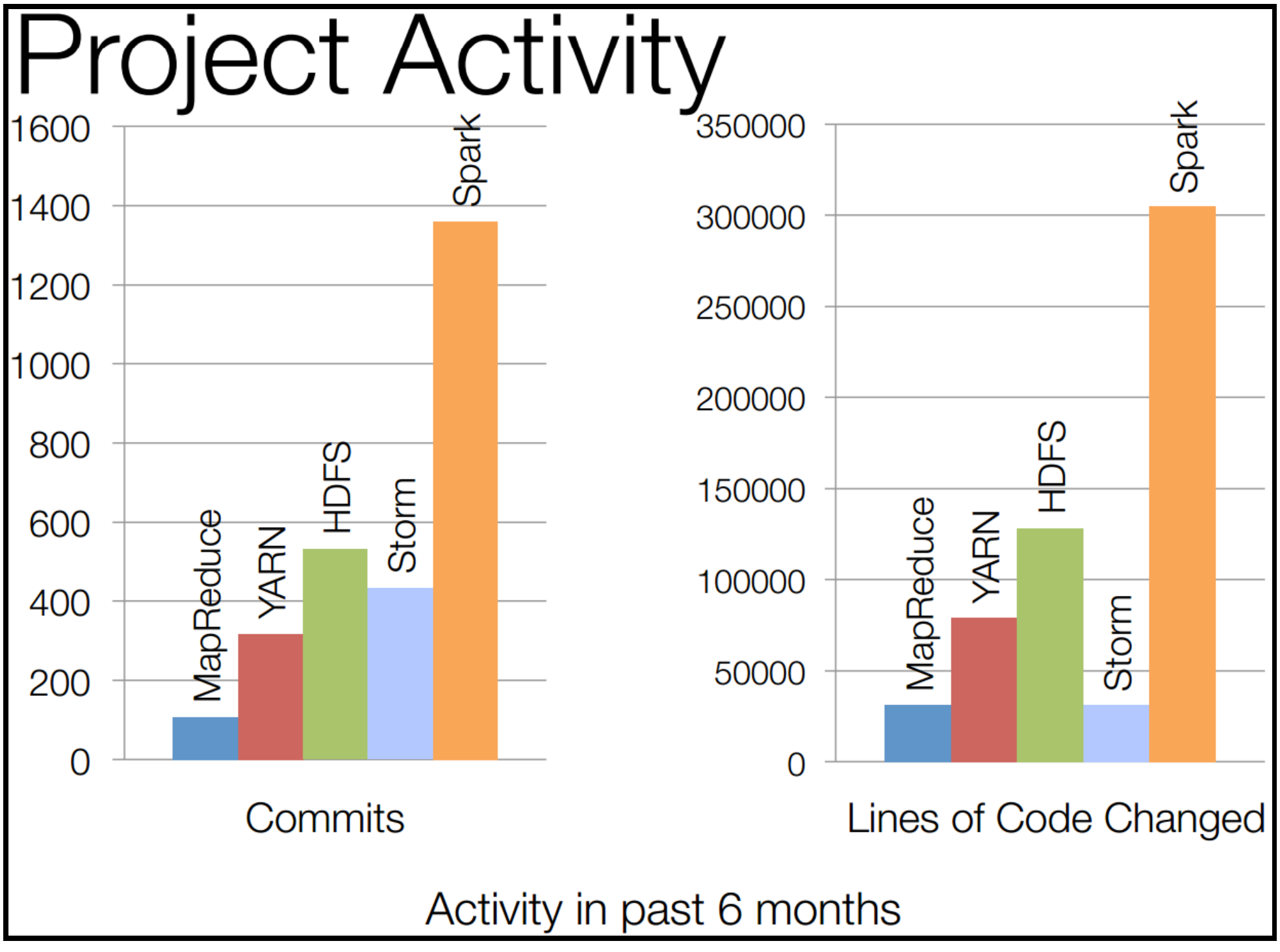
Apache Spark



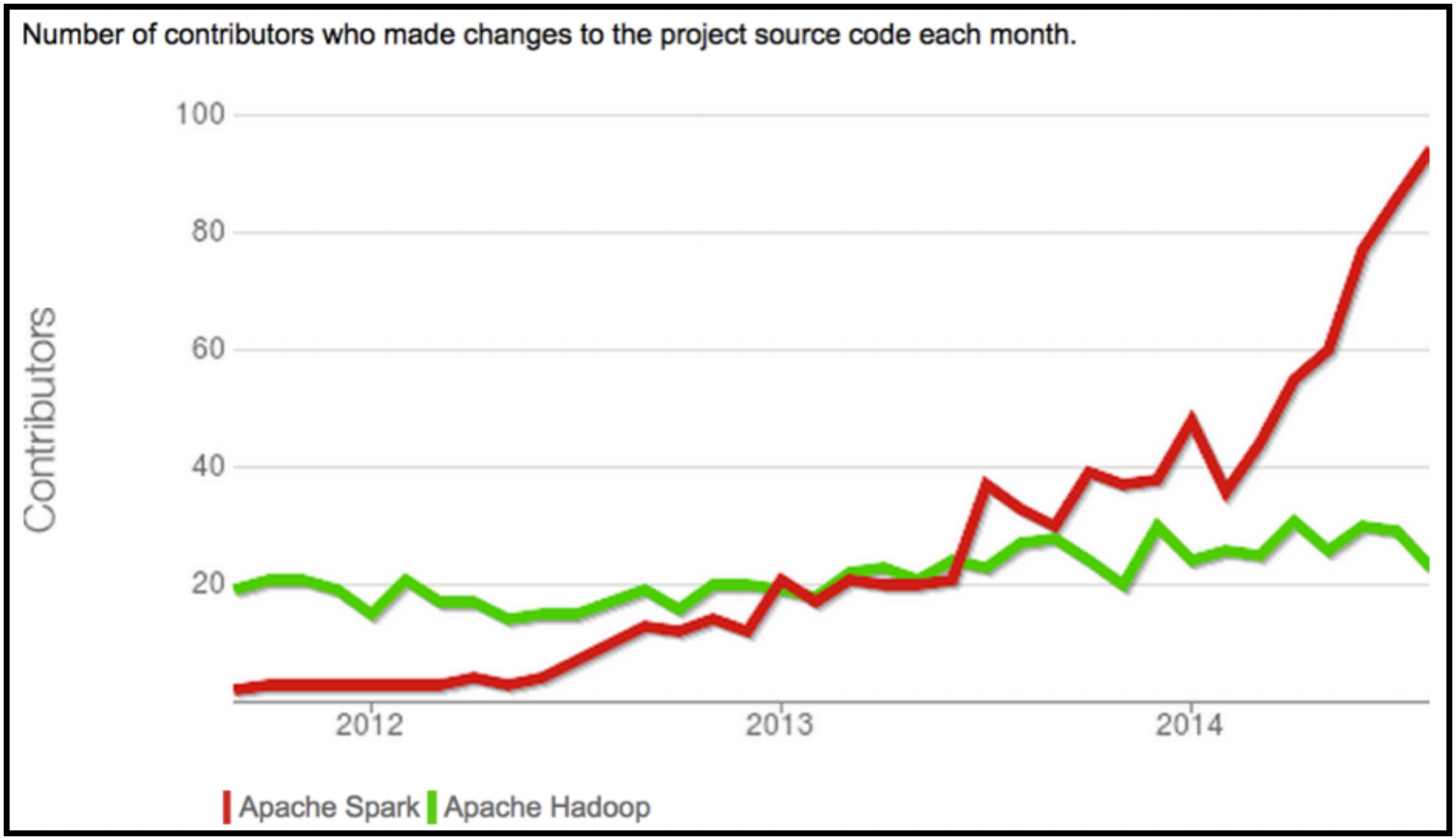
Spark code size



Spark project activity



Spark contributors



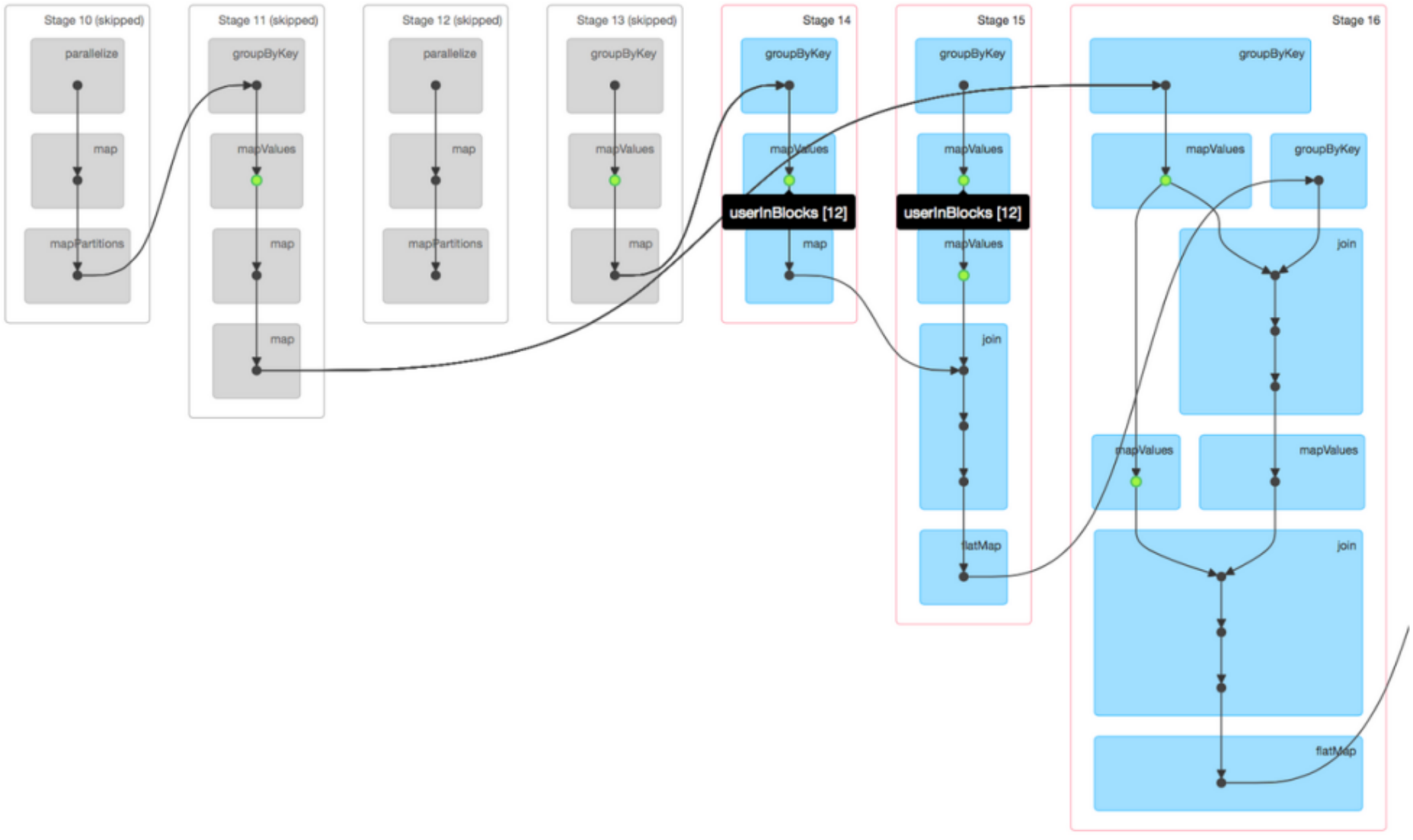
Apache Spark: general dataflow model

- express general DAG computations in a general 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
- Easier to use:
  - generalized patterns ⇒ unified engine for many use cases
  - functional programming / ease of use ⇒ reduction in cost to

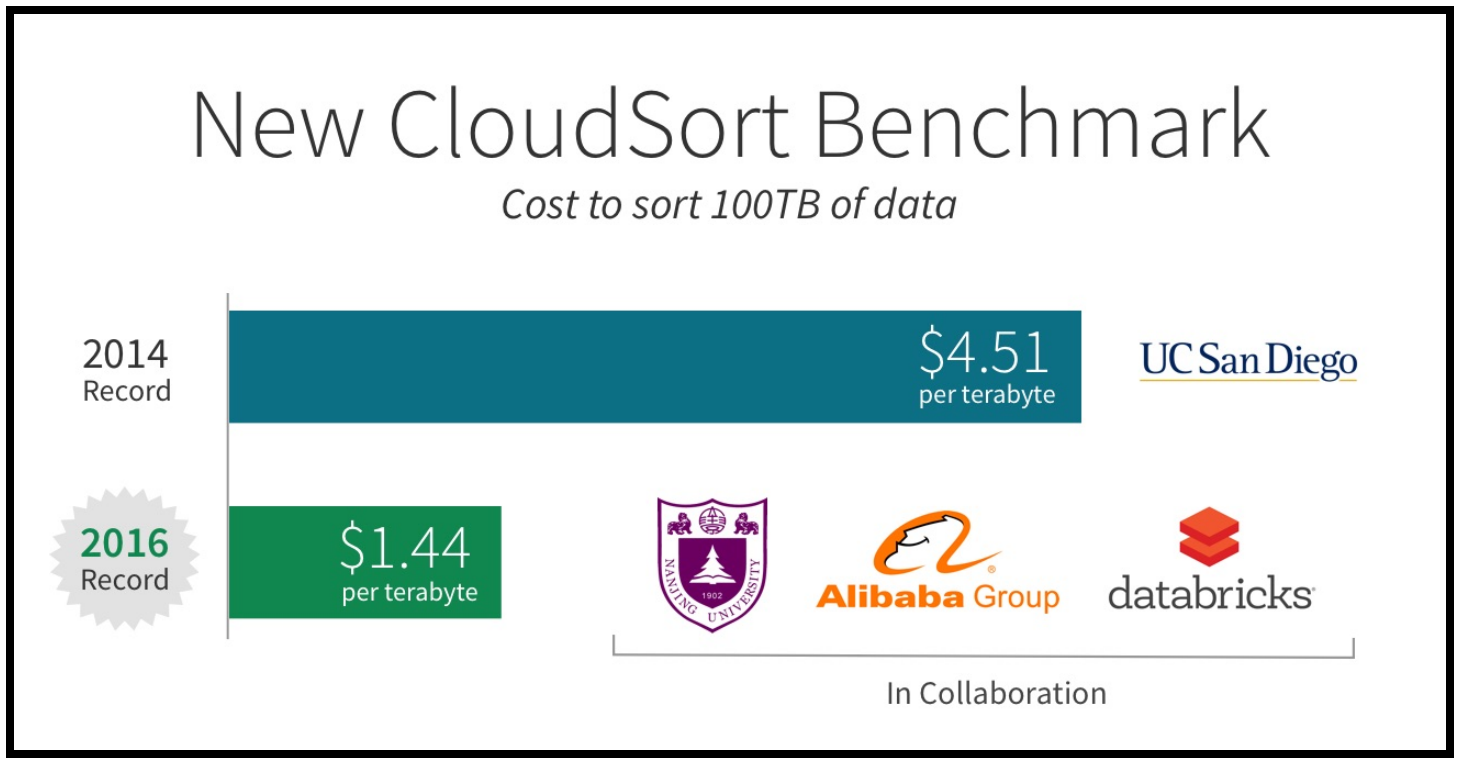
maintain large apps

Performance: 3X faster using 10X fewer machines for distributed sort

	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
6. Spark vs MapReduce
7. (Spark on AWS ES Cluster)

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

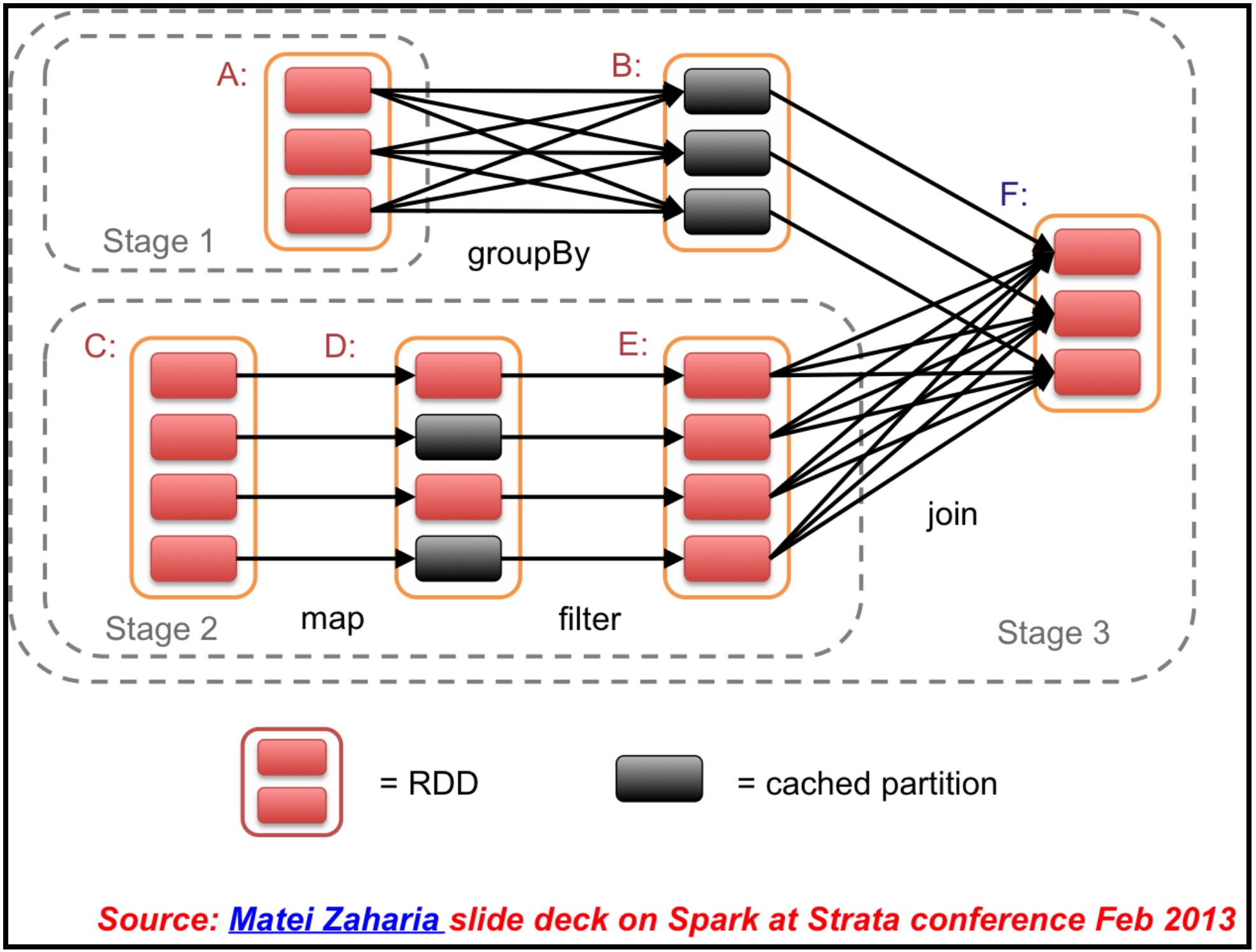
Resilient Distributed Datasets (RDDs)

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

Building RDDs

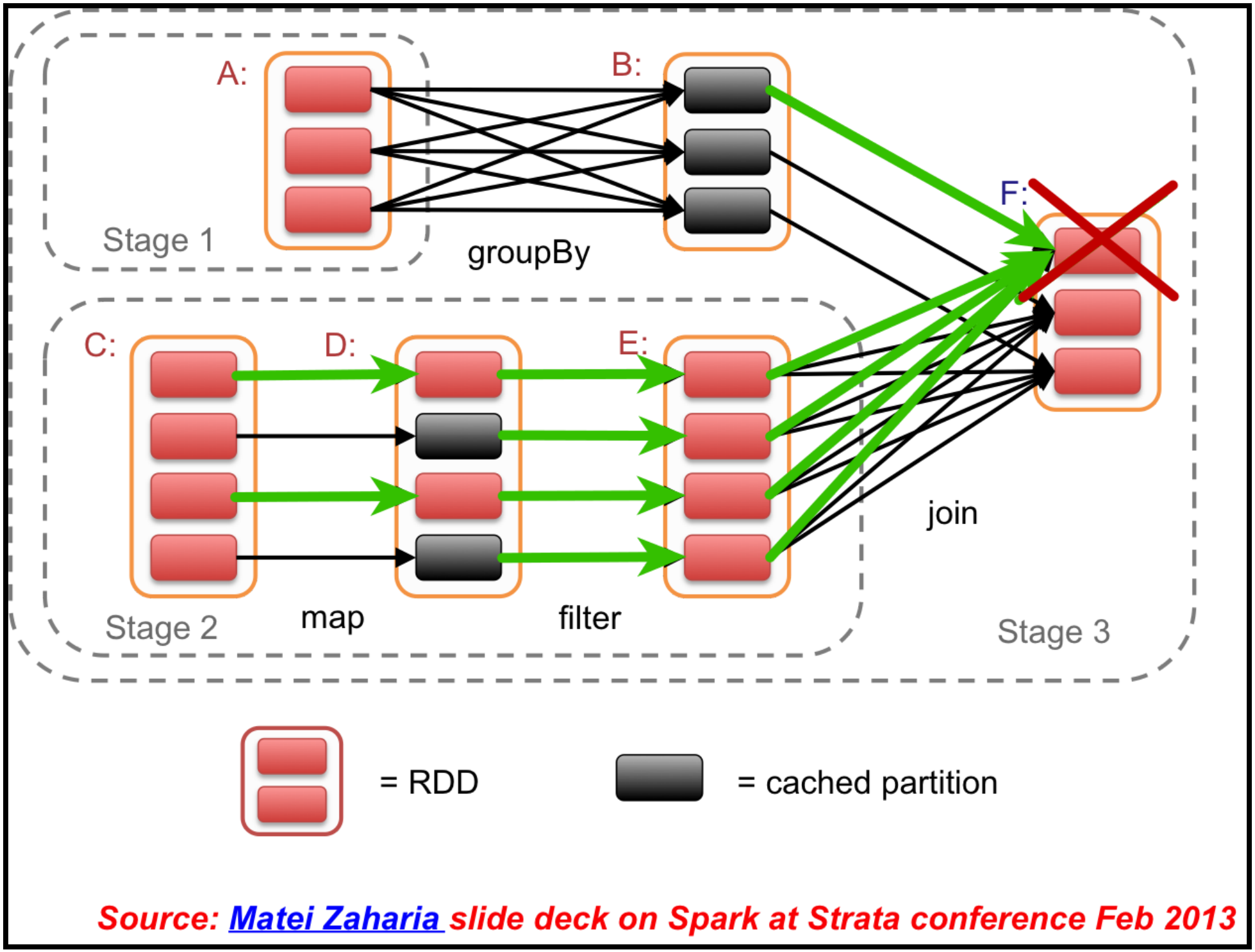
- Built via :
  - paralelization of data (**sc.parallelize**)
  - reading from parallel data sources (hdfs, s3, Cassandra)
  - parallel transformations of other RDDs(map, filter, …)
- Automatically rebuilt on failure (lineage)

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

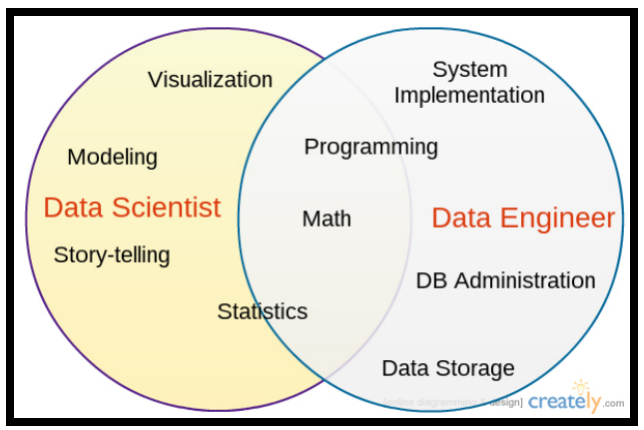
Plan

1. Distributed programming
2. Spark data model
3. **Spark programming**
    1. Data scientist vs Data engineer
    2. Scala crash course
    3. Spark Hello World
4. Spark execution model
5. Spark cluster
6. Spark ecosystem
7. Spark vs MapReduce
8. (Spark on AWS ES Cluster)

Spark

1. A ***general pourpose computation engine***:
    1. distribute data
    2. distribute computation
2. Lots of extensions on the side: machine learning, graph procesing, SparkSQL
3. Can be used directly by data scientists and data engineers

Data scientist - Data engineer (Roles)



Data scientist:

- ask the right questions about data
- perform data analysis
- create statisticals or mathematical models
- presents results

Data engineer:

- presents results

- enable data scientists to do their job effectively
- manage the data workflow (collection, storage, processing)
- serves the data via un API to a data scientist to easily querying it

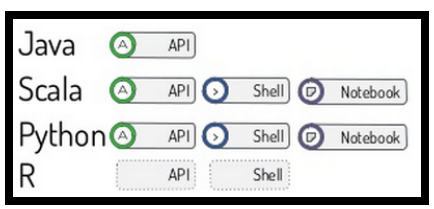
101.datascience

Data scientist - Data engineer (Languages)

1. Data scientist:
    1. R
    2. Python
    3. Matlab…
2. Data engineer:
    1. Java/C

Why Scala

- Spark itself is written in Scala:
  - lambdas
  - extensions (paralel processing, dsl)
- Documentation/tutorials
- Scala/Python only tools
  - Spark-shell
    - inline test of data processing algorithms
  - Spark notebooks: shell on steroids ⇒ data analytics dashboards



Why Scala

- ***Use the same language from analysis to large scale deployement***

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

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.
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

Beyond RDDs

SparkSQL

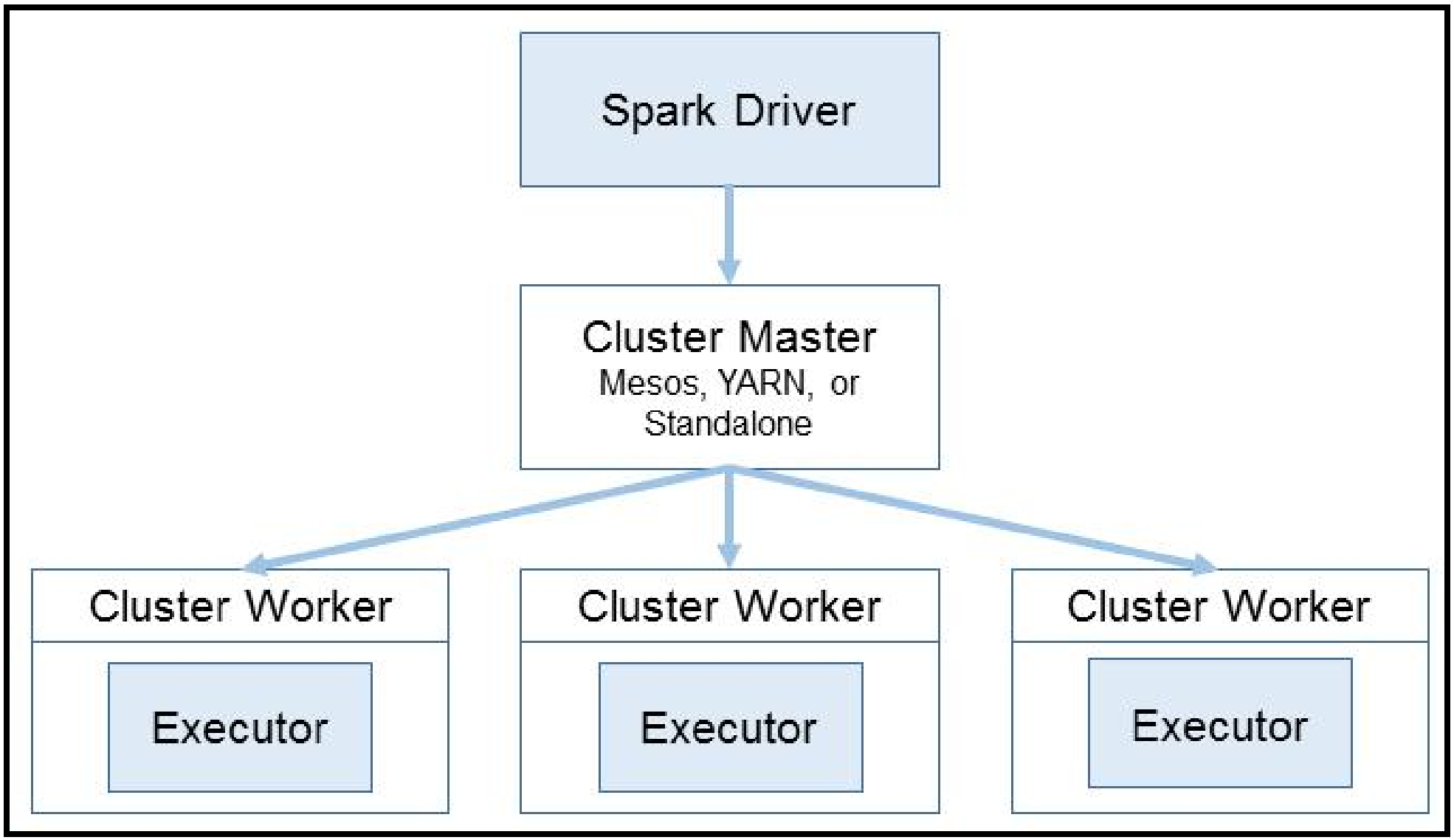
- storing general objects in relational like tables
- provide a SQL like interface for querying

SparkSQL example (<2.0)

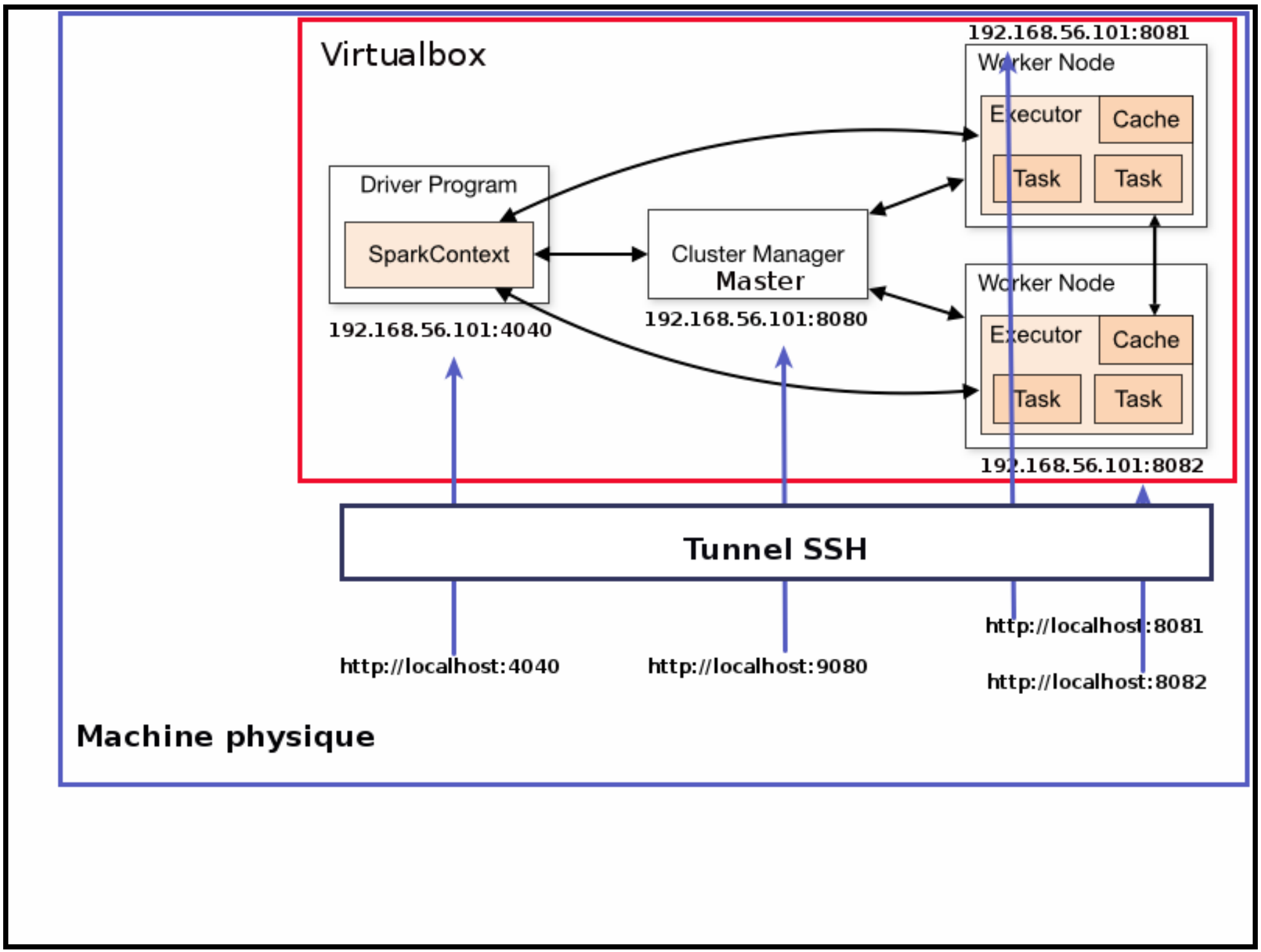
```
val sqlContext = new org.apache.spark.sql.SQLContext(sc)
import sqlContext._


// Define the schema using a case class.
case class Person(name: String, age: Int)


// Create an RDD of Person objects and register it as a table.
val people = sc.textFile("examples/src/main/resources/
people.txt").map(_.split(",")).map(p => Person(p(0), p(1).trim.toInt))
people.registerAsTable("people")


// SQL statements can be run by using the sql methods provided by sqlContext.
val teenagers = sql("SELECT name FROM people WHERE age >= 13 AND age <= 19")
```

Spark 2.0 ⇒ http://spark.apache.org/docs/latest/sql-programming-guide.html

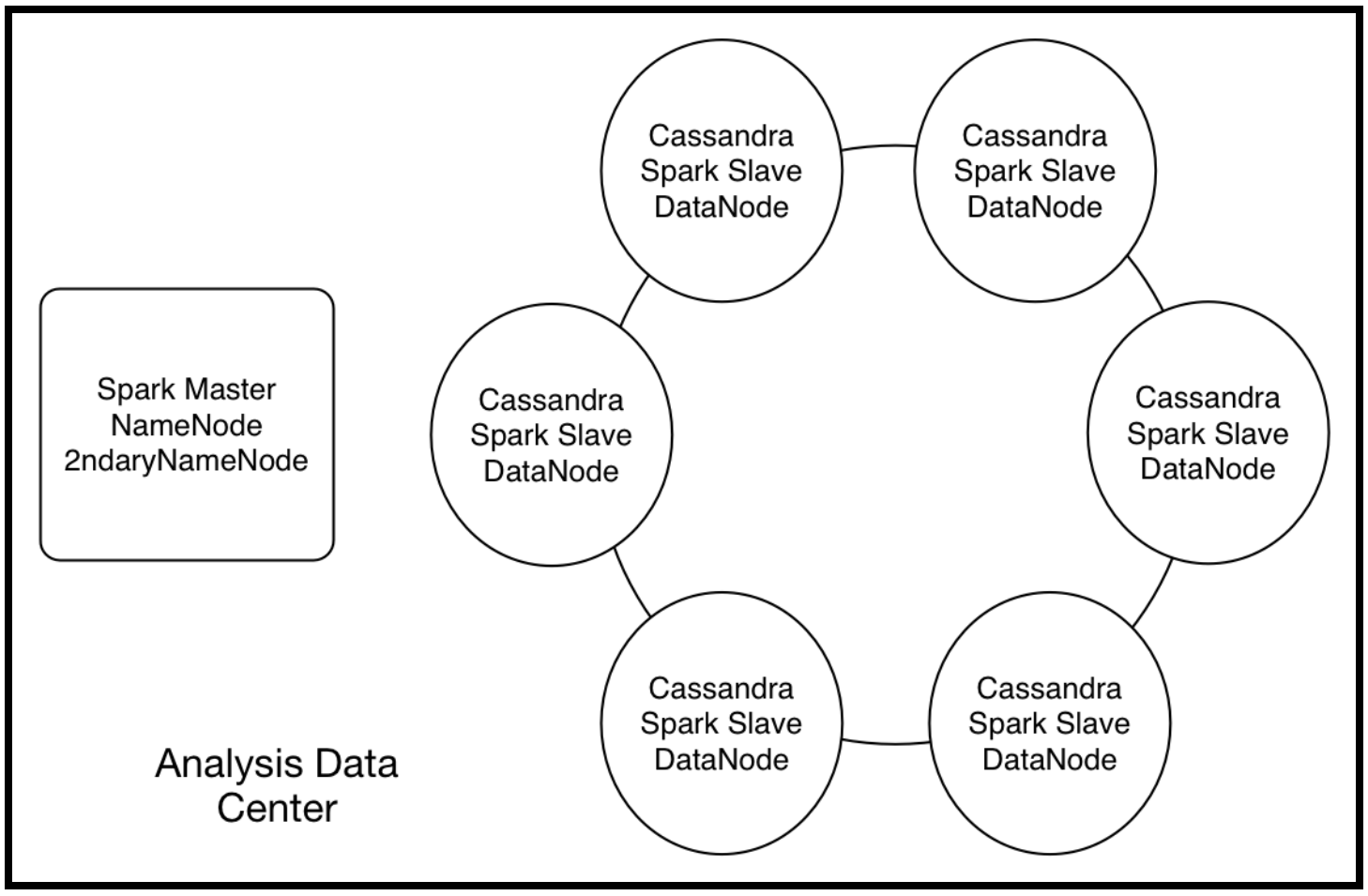
Spark Cluster Architecture



Spark Cluster Architecture



Cassandra and Spark



Planet Cassandra article

Cassandra and Spark example

Datastax Spark Driver on github

```
package com.datastax.spark.connector.demo

import org.apache.spark.SparkContext._
import com.datastax.spark.connector.cql.CassandraConnector
import com.datastax.spark.connector._

object WordCountDemo extends DemoApp {

  CassandraConnector(conf).withSessionDo { session =>
    session.execute(s"CREATE KEYSPACE IF NOT EXISTS demo WITH REPLICATION = 
    session.execute(s"CREATE TABLE IF NOT EXISTS demo.wordcount (word TEXT PR
    session.execute(s"TRUNCATE demo.wordcount")
  }

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

- (!) see TP for update version

TP Spark1

TP Spark RDD,SparkSQL,Cassandra