University of Bologna
Dipartimento di Informatica – Scienza e Ingegneria (DISI)
Engineering Bologna Campus

Class of **Computer Networks M** or
Infrastructures for Cloud Computing and Big Data

Global Data Batching

Luca Foschini

Academic year 2016/2017

Data processing in today large clusters

Excellent **data parallelism**

- It is easy to find what to parallelize
  Example: web data crawled by Google that need to be indexed – documents can be analyzed independently
- It is common to use thousands of nodes for one program that processes large amounts of data

Communication overhead not very significant w.r.t. the **overall execution time**

- Tasks access the disk frequently and sometimes run complex algorithms – **access to data & computation time** dominates the execution time
- **Data access rate** can become the bottleneck

MapReduce: motivations

Programmers can **focus only on the application logic and parallel tasks** without the hassle of dealing with scheduling, fault-tolerance, and synchronization?

MapReduce is a programming framework that provides

- **High-level API** to specify parallel tasks
- **Runtime system** that takes care of
  - Automatic parallelization & scheduling
  - Load balancing
  - Fault tolerance
  - I/O scheduling
  - Monitoring & status updates
- Everything runs on top of **GFS** (a distributed file system)

User benefits

- Huge speedups in programming/prototyping

  "it makes it possible to write a simple program and run it efficiently on a thousand machines in a half hour"
- Programmers can exploit **quite easily very large amounts of resources**

  Including users with no experience in distributed / parallel systems
- Based on abstract **black box** approach

Traditional MapReduce definitions

Statements that go back to **functional languages** (e.g., LISP, Scheme) as a **sequence of two steps for parallel exploration and results** (Map and Reduce)

- Also in other programming languages: Map/Reduce in Python, Map in Perl

- **Map** (distribution phase)
  - **Input**: a *list* and a *function*
  - **Execution**: the function **is applied to** each list item
  - **Result**: a *new list* with the results of the function
- **Reduce** (result harvesting phase)
  - **Input**: a *list* and a *function*
  - **Execution**: the function **combines/aggregates** the list items
  - **Result**: one new item

What is MapReduce… in a nutshell

The terms are borrowed from Functional Languages (e.g., Lisp)

Sum of squares:

- **(map square '(1 2 3 4))**
  - Output: (1 4 9 16)

  [processes each record sequentially and independently]

- **(reduce + '(1 4 9 16))**
  - (+ 16 (+ 9 (+ 4 1)))
  - Output: 30

  [processes set of all records in batches]

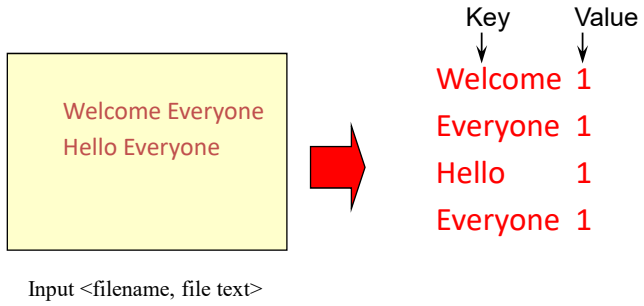
- Let's consider a sample application: Wordcount

  You are given a huge dataset (e.g., Wikipedia dump or all of Shakespeare's works) and asked to list the count for each of the words in each of the searched documents

Map

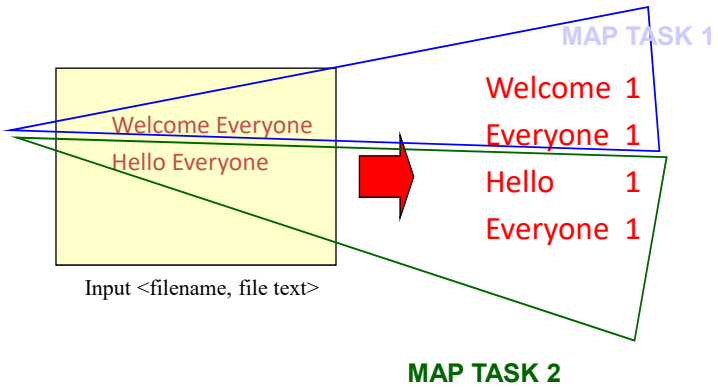
Extensively apply the function

- **Process all single records to generate intermediate key/value pairs**



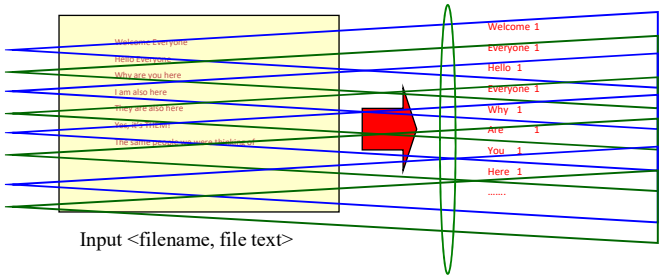
Map

- **In parallel** Process individual records to generate intermediate key/value pairs



Map

- **In parallel** process **a large number** of individual records to generate intermediate key/value pairs



MAP TASKS

Reduce

Collect the whole information

- Reduce processes and **merges all intermediate values** associated **per key**

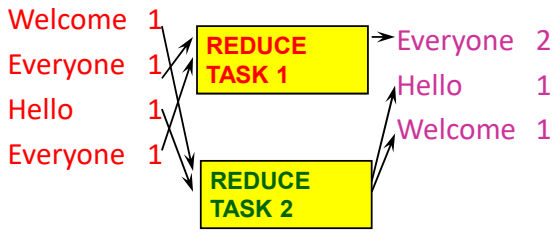
Welcome	1
Everyone	1
Hello	1
Everyone	1

Key	Value
Everyone	2
Hello	1
Welcome	1

Reduce

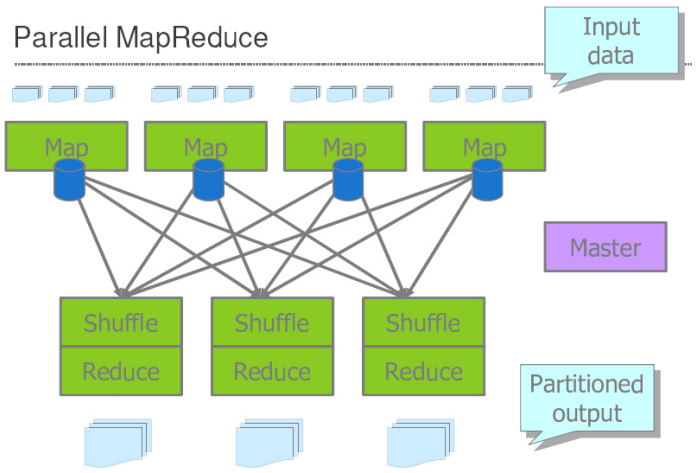
Each key assigned to one Reduce

- **In parallel** processes and merges all intermediate values by partitioning keys



- Popular splitting: *Hash partitioning, such as* key is assigned to
  - reduce # = hash(key)%number of reduce tasks

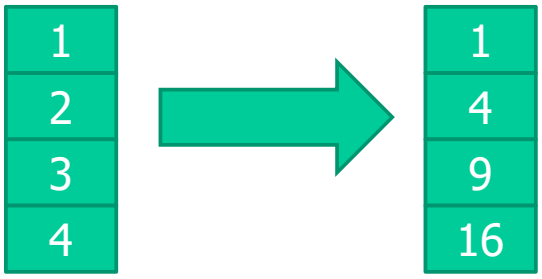
MapReduce: a deployment view



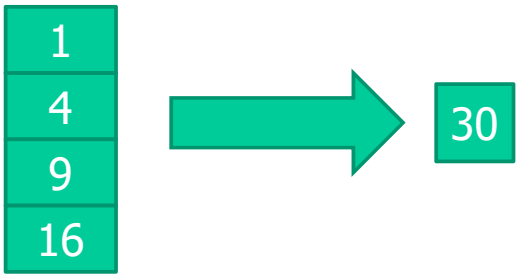
- Read many chunks of distributed data (no data dependencies)
- **Map**: extract something from each chunk of data
- Shuffle and sort
- **Reduce**: aggregate, summarize, filter or transform sorted data
- Programmers can specify **Map and Reduce functions**

Traditional MapReduce examples (again)

Map (square, [1, 2, 3, 4])



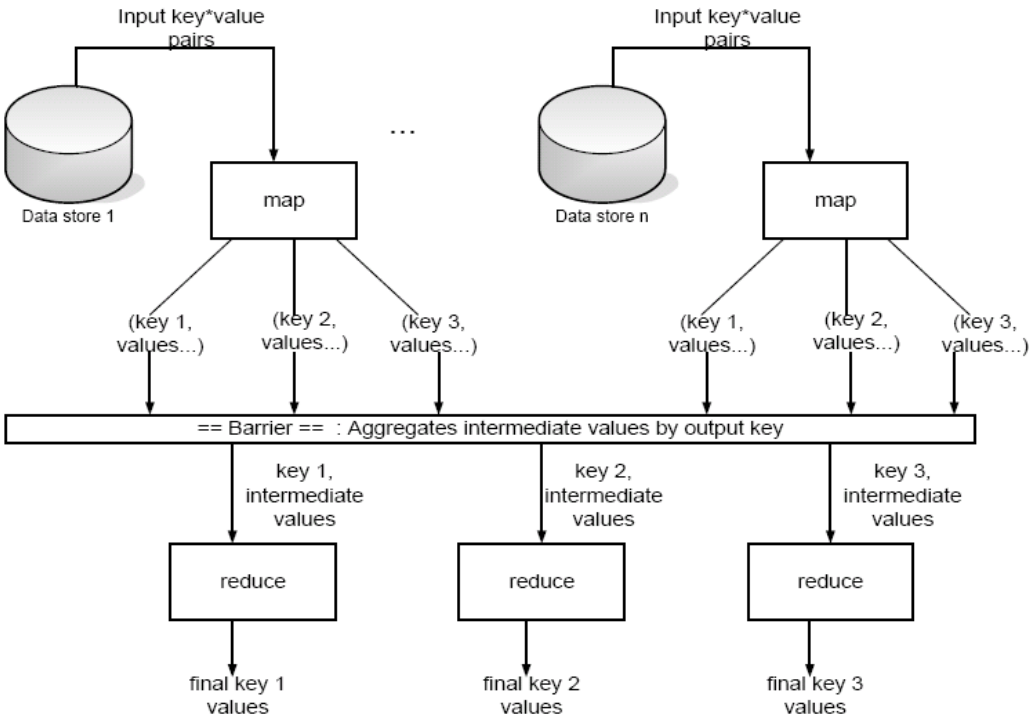
Reduce (add, [1, 4, 9, 16])



Google MapReduce definition

- **map (String key, String val)** is run on each item in set
  - **Input example**: a set of files, with keys being file names and values being file contents
  - Keys & values can have different types: the programmer has to convert between Strings and appropriate types inside map()
  - **Emits**, i.e., outputs, (new-key, new-val) pairs
  - Size of output set can be different from size of input set
- The runtime system aggregates the output of map by key
- **reduce (String key, Iterator vals)** is run for each *unique* key emitted by map()
  - It is possible to have more values for one key
  - Emits final output pairs (possibly smaller set than the intermediate sorted set)

Map & aggregation must finish before reduce can start



Running a MapReduce program

- The final user fills in ***specification object***
  - *Input/output file names*
  - *Optional tuning parameters* (e.g., size to split input/output into)
- The final user invokes ***MapReduce function*** and passes it ***the specification object***
- The **runtime system calls map() and reduce()**

While the final user just has to specify the operations

Word count example

```
map(String input_key, String input_value):
// input_key: document name
// input_value: document contents
   for each word w in input_value:
     EmitIntermediate(w, "1");

reduce(String output_key,
                  Iterator intermediate_values):
// output_key: a word
// output_values: a list of counts
   int result = 0;
   for each v in intermediate_values: result += ParseInt(v);
   Emit(AsString(result));
```

Word count illustrated

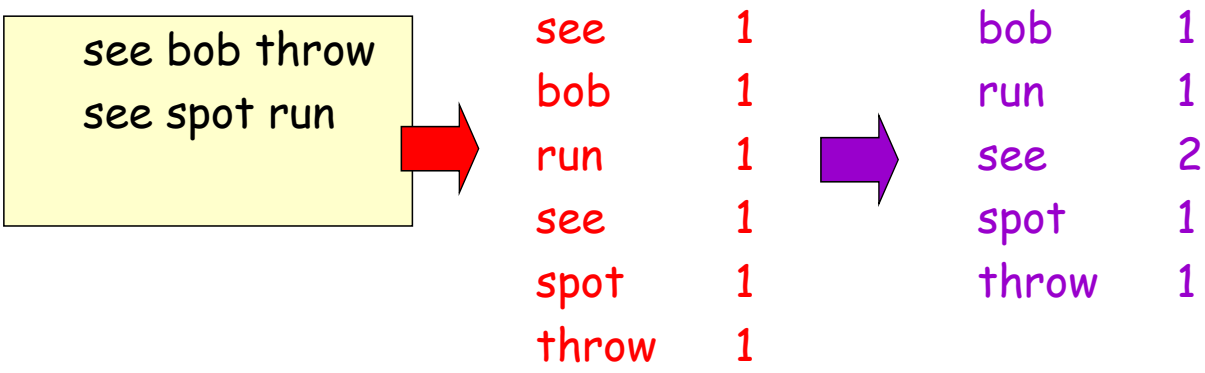
map(key=url, val=contents):

For each word *w* in contents, emit (w, “1”)

reduce(key=word, values=uniq_counts):

Sum all “1”s in values list

Emit result “(word, sum)”



Many other applications

- **Distributed *grep***
  - map() emits a line if it matches a supplied pattern
  - reduce() is an identity function; just emit same line
- **Distributed *sort***
  - map() extracts *sorting key* from record (file) and outputs (key, record) pairs
  - reduce() is an identity function; just emit same pairs
  - The actual sort is done automatically by runtime system
- **Reverse web-link graph**
  - map() emits (*target*, *source*) pairs for each link to a *target* URL found in a file *source*
  - reduce() emits pairs (*target*, list(*source*))

Other applications

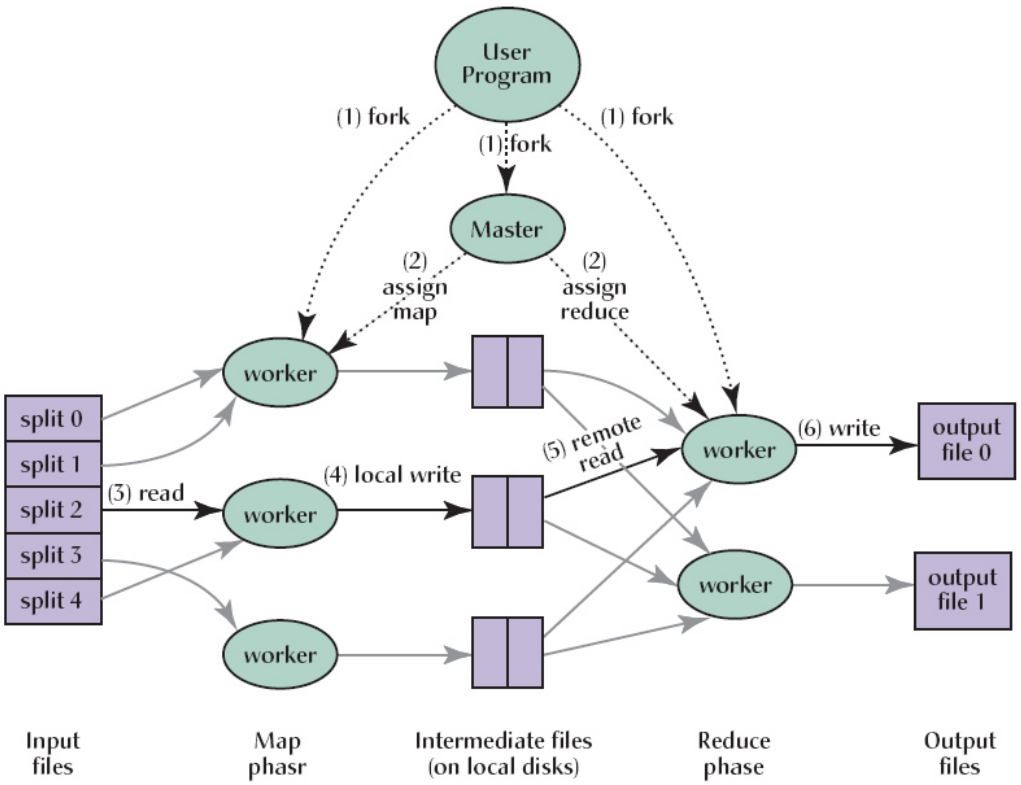
- Machine learning issues
- Google news clustering problems
- Extracting data + reporting popular queries (Zeitgeist)
- Extract properties of web pages for experiments/products
- Processing satellite imagery data
- Graph computations
- Language model for machine translation
- Rewrite of Google Indexing Code in MapReduce

  Size of one phase 3800 => 700 lines, over 5x drop

Implementation overview (at Google)

- **Environment**
  - **Large clusters of PCs connected with Gigabit links**
    - 4-8 GB RAM per machine, dual x86 processors
    - Network bandwidth often significantly less than 1 GB/s
    - Machine failures are common due to # machines
  - **GFS**: distributed file system manages data
    - Storage is provided by cheap IDE disks attached to machine
- **Job scheduling system: jobs made up of tasks, scheduler assigns tasks to machines**
- **Implementation is a C++ library linked into user programs**

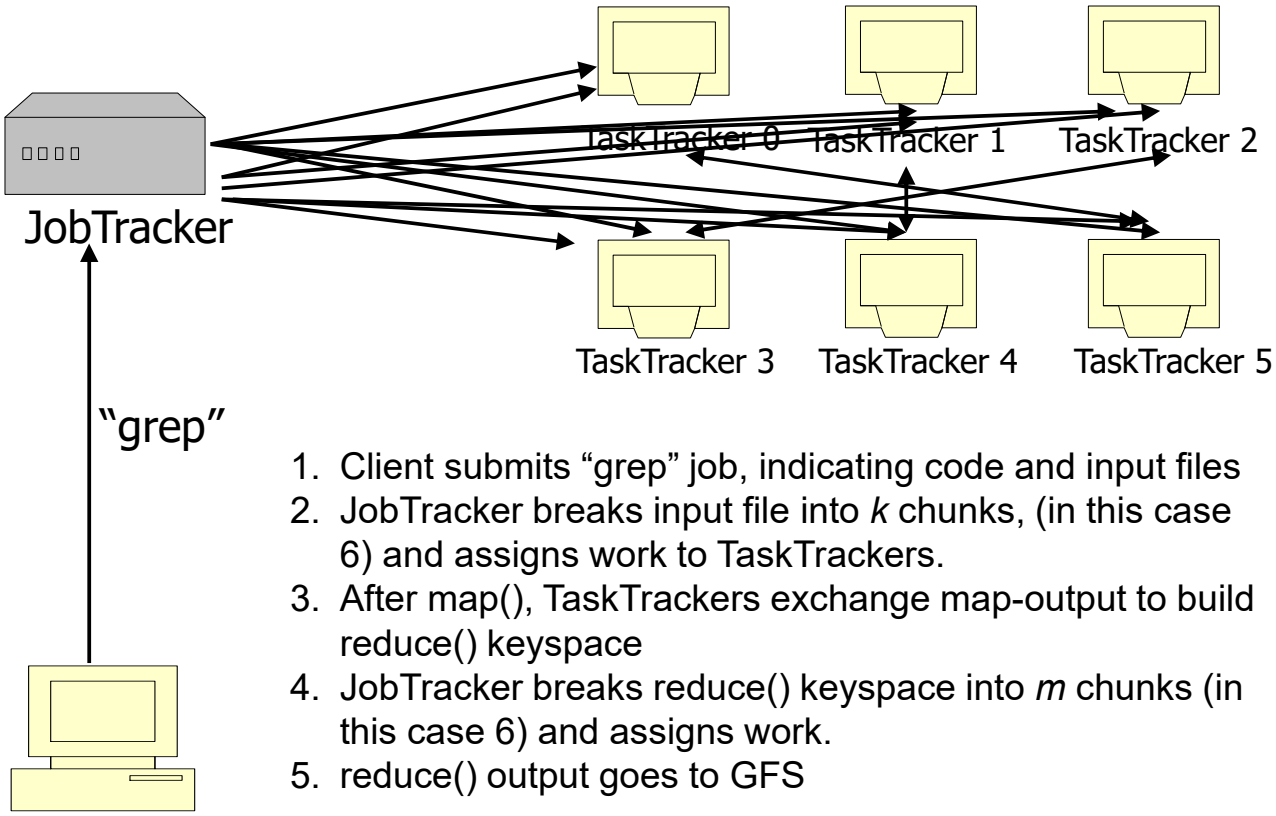
Architecture example



Scheduling and execution

- **One master, many workers**
  - Input data split into M map tasks (typically 64 MB in size)
  - Reduce phase partitioned into R reduce tasks
  - Tasks are assigned to workers dynamically
  - Often: M=200,000; R=4000; workers=2000
- **Master assigns each map task to a free worker**
  - Considers locality of data to worker when assigning a task
  - Worker reads task input (often from local disk)
  - Intermediate key/value pairs written to local disk, divided into R regions, and the locations of the regions are passed to the master
- **Master assigns each reduce task to a free worker**
  - Worker reads intermediate k/v pairs from map workers
  - Worker applies user's reduce operation to produce the output (stored in GFS)

Scheduling and execution example (2)



1. Client submits "grep" job, indicating code and input files
2. JobTracker breaks input file into *k* chunks, (in this case 6) and assigns work to TaskTrackers.
3. After map(), TaskTrackers exchange map-output to build reduce() keyspace
4. JobTracker breaks reduce() keyspace into *m* chunks (in this case 6) and assigns work.
5. reduce() output goes to GFS

Fault-tolerance

- **On master failure**:
  - State is checkpointed to GFS: new master recovers & continues
- **On worker failure**:
  - Master detects failure via periodic heartbeats
  - Both completed and in-progress map tasks on that worker should be re-executed (→ output stored on local disk)
  - Only in-progress reduce tasks on that worker should be re-executed (→ output stored in global file system)
- **Robustness**:
  - Example: Lost 1600 of 1800 machines once, but completed successfully

Favoring Data locality

The goal is to **preserve and conserve network bandwidth**

- In GFS, we know that data files are divided into 64 MB blocks and 3 copies of each are stored on different machines
- Master program **schedules map() tasks based on the location of these replicas**:
  - Put map() tasks physically on the same machine as one of the input replicas (or, at least on the same rack/network switch)
  - In this way, the machines can read input at local disk speed. Otherwise, rack switches would limit read rate

Backup tasks

- **Problem: stragglers** (i.e., **slow workers in ending**) significantly lengthen the completion time
  - Other jobs may be consuming resources on machine
  - Bad disks with soft errors (i.e., correctable) transfer data very slowly
  - Other weird things: processor caches disabled at machine init
- **Solution:** Close to completion, **spawn backup copies of the remaining in-progress tasks**
  - Whichever one finishes first, wins
  - Additional cost: a few percent more resource usage.
  - Example: A sort program without backup was 44% longer

Hadoop: a Java-based MapReduce

Hadoop is an open source platform for MapReduce by Apache

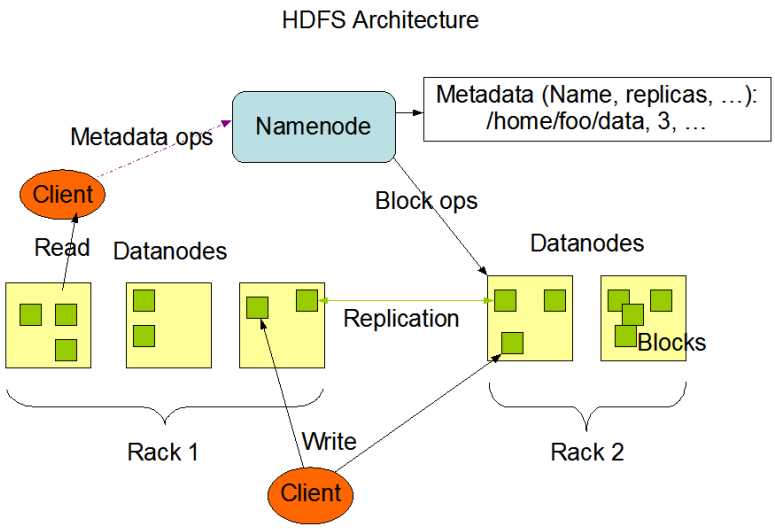
Started as open source MapReduce written in Java, but evolved to support other languages such as Pig and Hive

Hadoop common: set of utilities that support the other subprojects

- FileSystem, RPC, and serialization libraries

- Several essential subprojects:
  - **Distributed file system (*HDFS*)**
  - ***MapReduce***
  - **Yet Another Resource Negotiator** (**YARN**) for cluster resource management

HDFS



- Inspired by GoogleFS
- Master/slave architecture
  - NameNode is master (meta-data operations, access control)
  - DataNodes are slaves: one per node in the cluster

YARN resource manager

YARN provides management for virtual Hadoop clusters over a large physical cluster

- Handles node allocation in a cluster
- Supplies new nodes with configuration
- Distributes Hadoop to allocated nodes
- Starts Map/Reduce and HDFS workers
- Includes management and monitoring

Today, other resource managers are available, such as MESOS

The YARN Scheduler

YARN or **Yet Another Resource Negotiator**

Used underneath Hadoop 2.x +

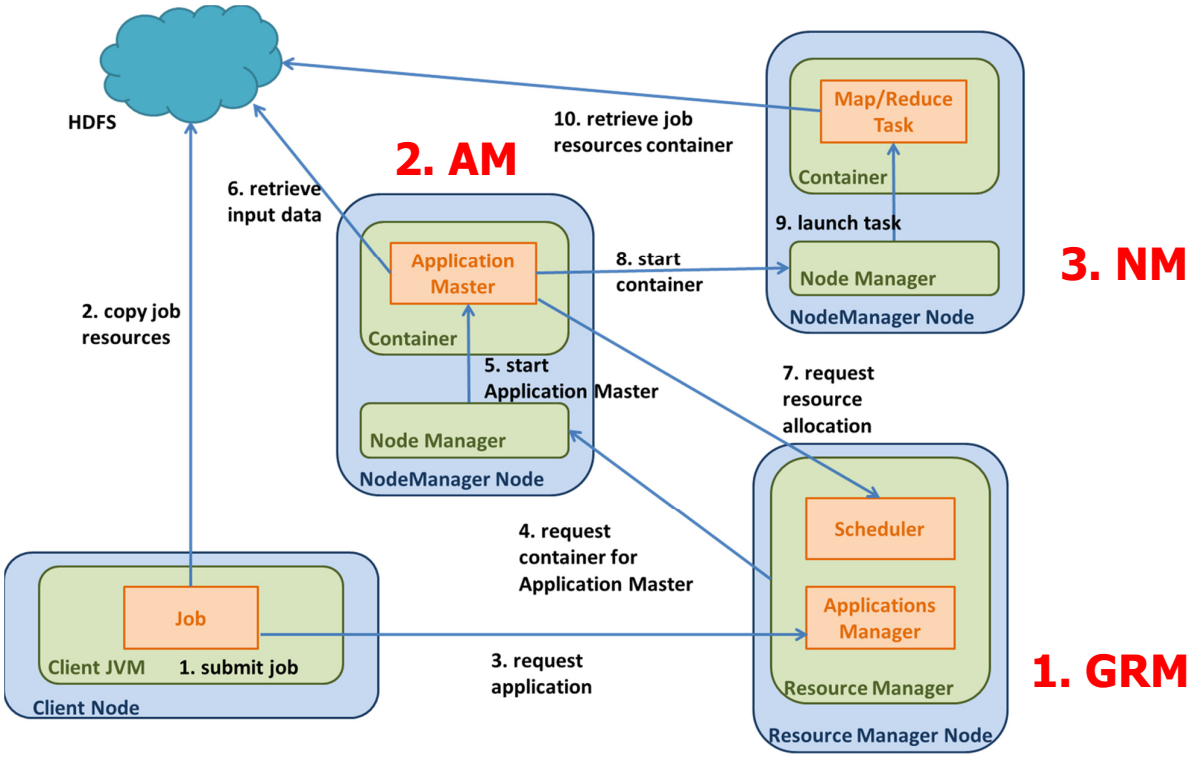
Treats each server as a collection of *containers*

- Container = fixed CPU + fixed memory (think of Linux cgroups, but even more lightweight)

With main components

1. **Global *Resource Manager*** ***(GRM)*** node
   - ***Scheduler*** that globally allocates the required resources
   - ***ApplicationManager*** that coordinates the execution of the job on the other nodes
2. ***Application Master*** ***(AM)*** **Per-application** (per job)
   - Container negotiation with RM and NMs
   - Detecting task failures of that job
3. **Per-server *Node Manager*** ***(NM)***
   - Daemon and server-specific functions that ***manage*** local resources, ***instantiate containers*** to run tasks, ***monitor container*** resource usage

YARN at work



Hadoop extensions (out-of-our-scope…)

***Avro**:* Large-scale data serialization

***Chukwa**:* Data collection (e.g., logs)

***Hbase**:* Structured data storage for large tables

***Hive**:* Data warehousing and management (Facebook)

***Pig**:* Parallel SQL-like language (Yahoo)

***ZooKeeper**:* coordination for distributed apps

***Mahout**:* machine learning and data mining library

***Sahara**:* deployment of Hadoop clusters on OpenStack

Hadoop for OpenStack

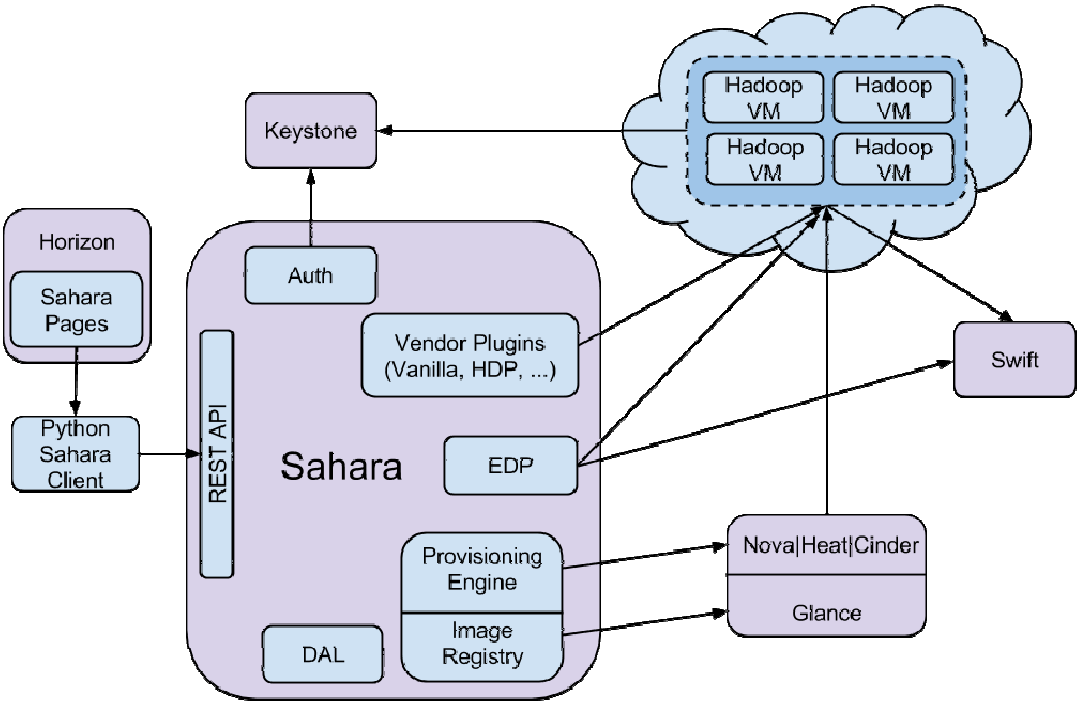
Hadoop can exploit the virtualization provided by **OpenStack** in order to obtain more flexible clusters and a better resource utilization

OpenStack **Sahara** service can allow to **deploy** and **configure Hadoop clusters** in a **Cloud environment by adding services**

- **Cluster scaling functionalities**
- **Analytics as a Service (AaaS) functionalities**

Sahara is accessible by OpenStack in all ways, **via:** ***dashboard, CLI or RESTful API***

Sahara components



Spark

It is not a modified version of Hadoop but a

Separate, **fast, MapReduce-like engine**

- In-memory data storage for very fast iterative queries
- General execution of graphs and powerful optimizations
- Up to 40 times faster than Hadoop

Compatible with Hadoop's storage APIs

- Can read/write to any Hadoop-supported system, including HDFS, HBase, SequenceFiles, etc.

Spark Project History

- Spark project started in 2009, open sourced 2010
- Spark started summer 2011, alpha April 2012
- In use at Berkeley, Princeton, Klout, Foursquare, Conviva, Quantifind, Yahoo! Research & others
- 200+ member meetup, 500+ watchers on GitHub

Why Spark?

Why a New Programming Model?

MapReduce greatly simplified big data analysis

But when it becomes popular, users wanted more:

- More **complex**, **multi-stage applications** (e.g., iterative graph algorithms and machine learning)
- **More interactive ad-hoc queries**

- Both multi-stage and interactive apps require faster **data sharing** across parallel jobs

Spark Basics

Various types of data processing computations available in one single tool

- **Batch/streaming** analysis, interactive queries and iterative algorithms.
- Previously, these would require several distinct and independent tools

APIs available in Java, Scala, Python

- Also R language supported, for data scientists with moderate programming experience

Supports **several storage options and streaming inputs for parsing**

Spark at a glance

Leverages on in-memory data processing:

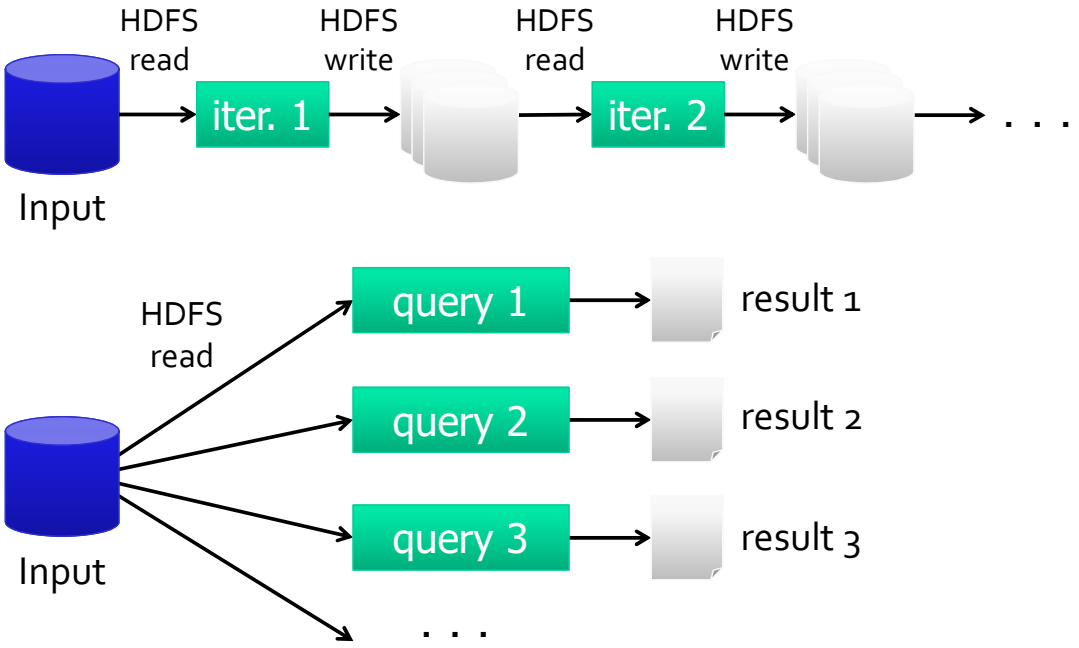
- Removes the MapReduce overhead of writing intermediate results on disk
- Fault-tolerance is still achieved through the concept of **lineage**.

Master/Worker cluster architecture

- Easily deployable in most environments, including existing Hadoop clusters

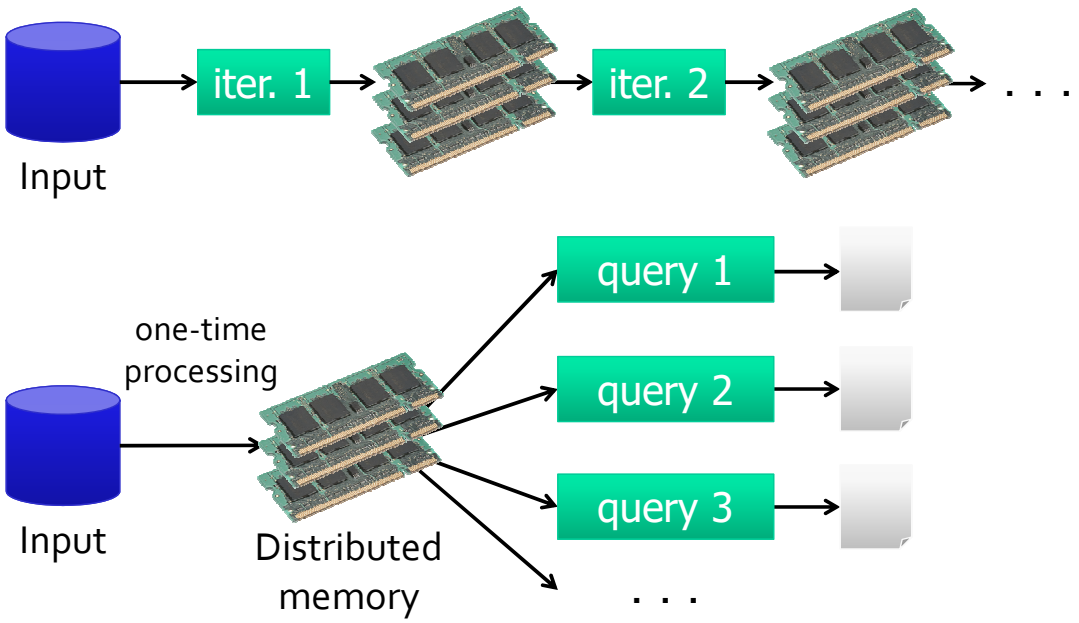
Widely configurable for performance optimization, both in terms of resource usage and application behavior

Data Sharing in MapReduce



Slow due to replication, **serialization**, and **disk IO**

Data Sharing in Spark



10-100× faster than network and disk

Spark Programming Model

Programs can be run both

- From compiled sources, with proper Spark dependencies, with the *Spark-submit* script
- *Interactively* from **Spark Shell**, a console available for Scala and Python languages

Key idea: ***resilient distributed datasets (RDDs)* kept in memory**

- **Distributed, immutable collections of objects**
- **Can be cached in memory across cluster nodes**

RDD Programming Model

Two kinds of operations performed on RDDs

- **Transformations** that act on existing RDDs, by creating new ones
  - **Similar to Hadoop *map* tasks**
  - ***Lazily* evaluated**
- **Actions** that return results from input RDDs
  - **Similar to Hadoop *reduce* tasks**
  - **Force immediate evaluation of pending transformations in the input RDD**

RDD Transformations

In addiction to being *lazily* evaluated, all **transformations** are computed again on every **action** requested

```
val lines = sc.textFile("data.txt")
val lineLengths = lines.map(s => s.length)
val totalLength = lineLengths.reduce((a, b) => a + b)
```

Transformation

Action

Until the third line, no operation is performed

The *reduce()* will then force a read from the text file and the *map()* transformation

Persisting RDDs

However, a further action can trigger **another** file read and another identical *map()*

```
val lines = sc.textFile("data.txt")
val lineLengths = lines.map(s => s.length)
println(lineLengths.count())
val totalLength = lineLengths.reduce((a, b) => a + b)
```

Transformation

Action

Action

This effect is expensive, but can be avoided by using the ***persist()*** method

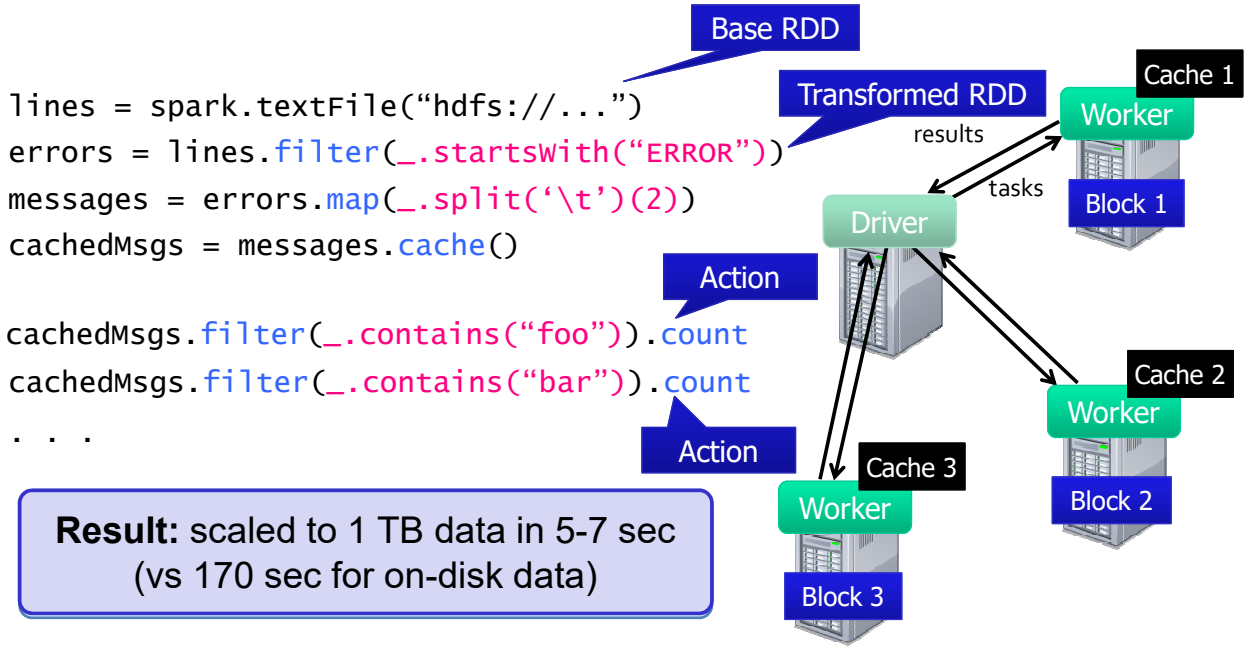
```
val lines = sc.textFile("data.txt")
val lineLengths = lines.map(s => s.length)
lineLengths.persist()
```

The RDD data read and mapped will then be saved for future actions

Example: Log Mining

Load error messages from a log into memory, then interactively search for various patterns

```
lines = spark.textFile("hdfs://...")
errors = lines.filter(_.startsWith("ERROR"))
messages = errors.map(_.split('\t')(2))
cachedMsgs = messages.cache()

cachedMsgs.filter(_.contains("foo")).count
cachedMsgs.filter(_.contains("bar")).count
. . .
```

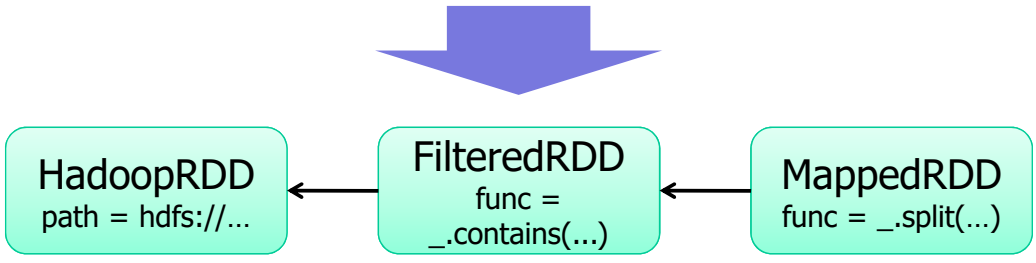


Result: scaled to 1 TB data in 5-7 sec (vs 170 sec for on-disk data)

Fault Tolerance

RDDs track the series of transformations used to build them (their ***lineage***) to re-compute lost data

```
messages = textFile(...).filter(_.contains("error"))
                        .map(_.split('\t')(2))
```



Example: Logistic Regression

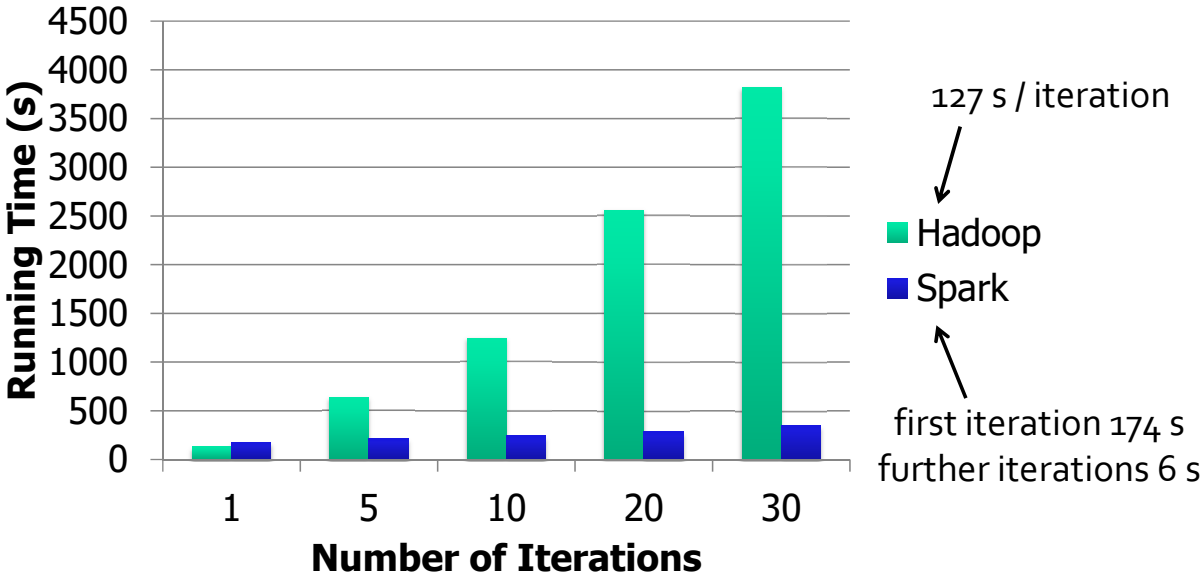
```
val data = spark.textFile(...).map(readPoint).cache()

var w = Vector.random(D)

for (i <- 1 to ITERATIONS) {
  val gradient = data.map(p =>
    (1 / (1 + exp(-p.y*(w dot p.x))) - 1) * p.y * p.x
  ).reduce(_ + _)
  w -= gradient
}

println("Final w: " + w)
```

Load data in memory once

Initial parameter vector

Repeated MapReduce steps to do gradient descent

Logistic Regression Performance

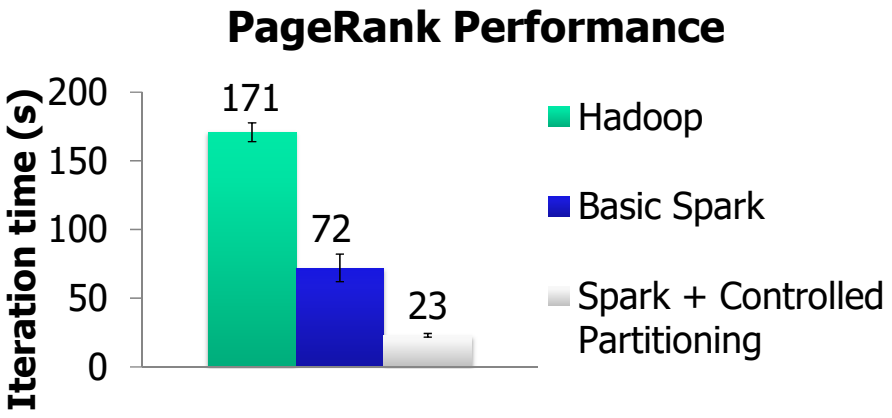


Supported Operators

- `map`
- `filter`
- `groupBy`
- `sort`
- `join`
- `leftOuterJoin`
- `rightOuterJoin`
- `reduce`
- `count`
- `reduceByKey`
- `groupByKey`
- `first`
- `union`
- `cross`
- `sample`
- `cogroup`
- `take`
- `partitionBy`
- `pipe`
- `save`
- `...`

Other Engine Features

- General **graphs of operators** (e.g. map-reduce-reduce)
- **Hash-based reduces** (faster than Hadoop sort)
- **Controlled data partitioning adapted** to lower communication



Spark Architecture

Once submitted, Spark programs create ***directed acyclic graphs (DAGs)*** of all transformations and actions, internally optimized for the execution

The graph is then split into **stages**, in turn composed by **tasks**, the smallest unit of work

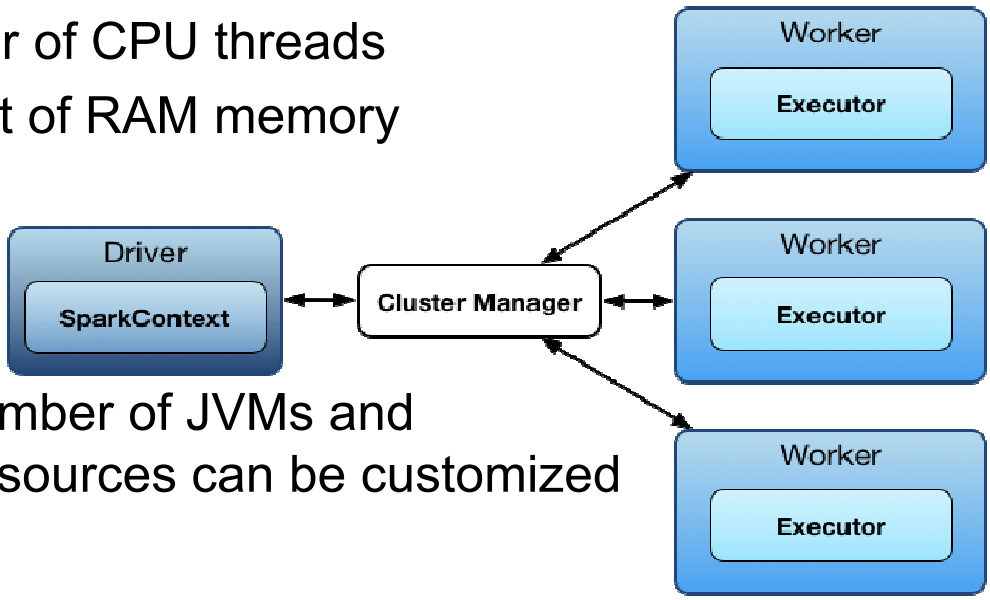
Thus, Spark is a master/slave system composed by:

- **Driver,** central coordinator node running the *main()* method of the program and dispatching tasks
- **Cluster Master**, node that launches and manages actual executors
- **Executors**, responsible for running tasks

Spark Architecture

Each executor spawns at least one dedicated **JVM**, to which a certain share of resources is assigned, in terms of:

- Number of CPU threads
- Amount of RAM memory



- The number of JVMs and their resources can be customized

Spark Deployment

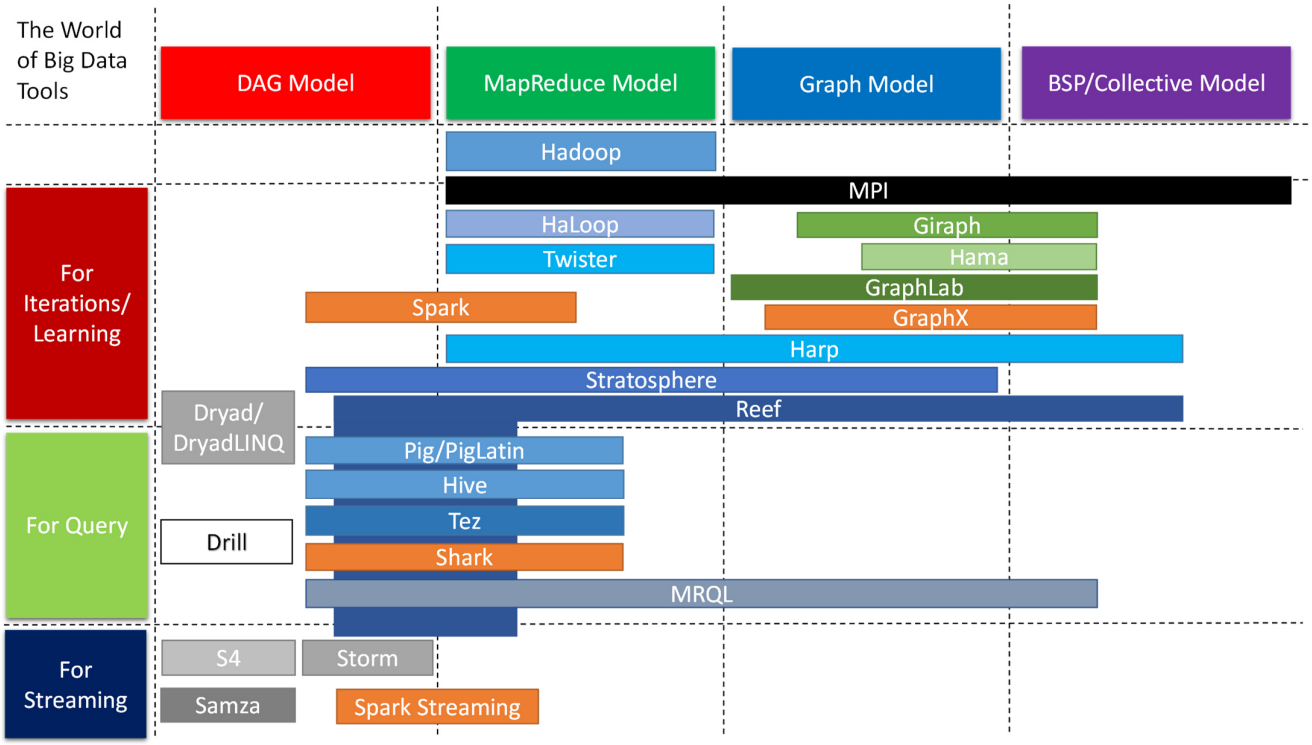
Spark can be deployed in a **standalone** cluster, i.e., its own cluster master independently launches and manages its executors

However, Spark can rely upon **external resource managers**, such as:

- **Hadoop YARN** **(already seen before…)**
- **Apache MESOS**

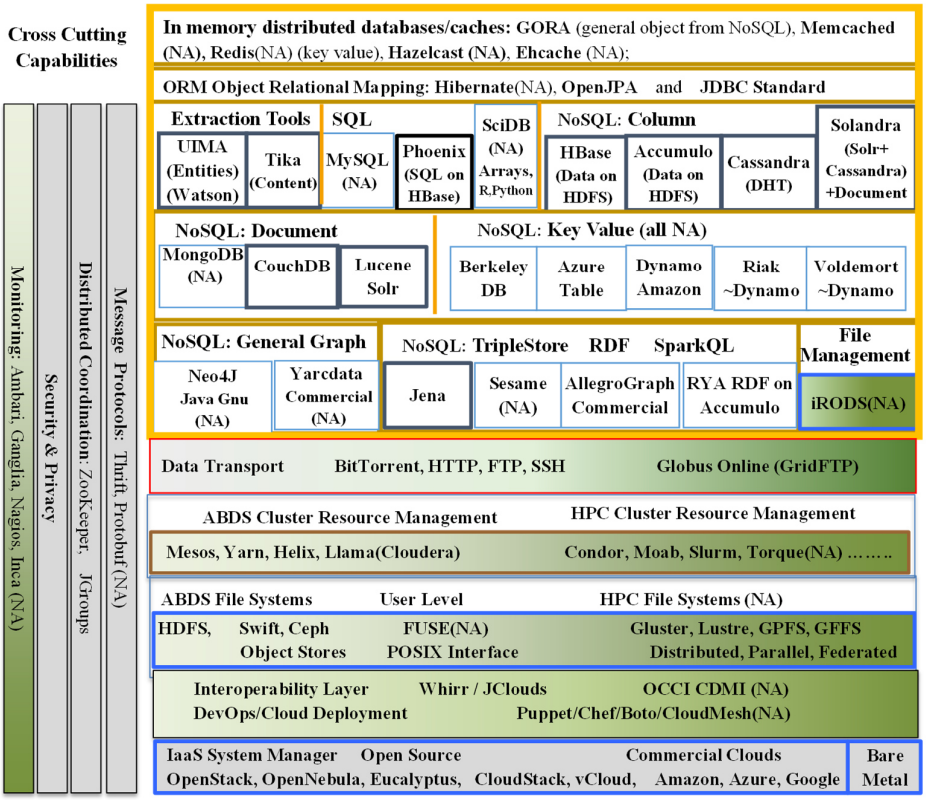
These others can provide richer functionalities, such as resource scheduling queues, not available in the standalone mode

The Big Data Tools Ecosystem



The figure of layered architecture is from Bingjing Zhang

A Layered Architecture view



- **NA** – Non Apache projects
- Green layers are Apache/ Commercial Cloud (light) to HPC (darker) integration layers

The figure of layered architecture is from Prof. Geoffrey Fox