Principles of Big Data

Dario Colazzo

Professor at Université Paris Dauphine

Adjoint Professor at Ecole Polytechnique

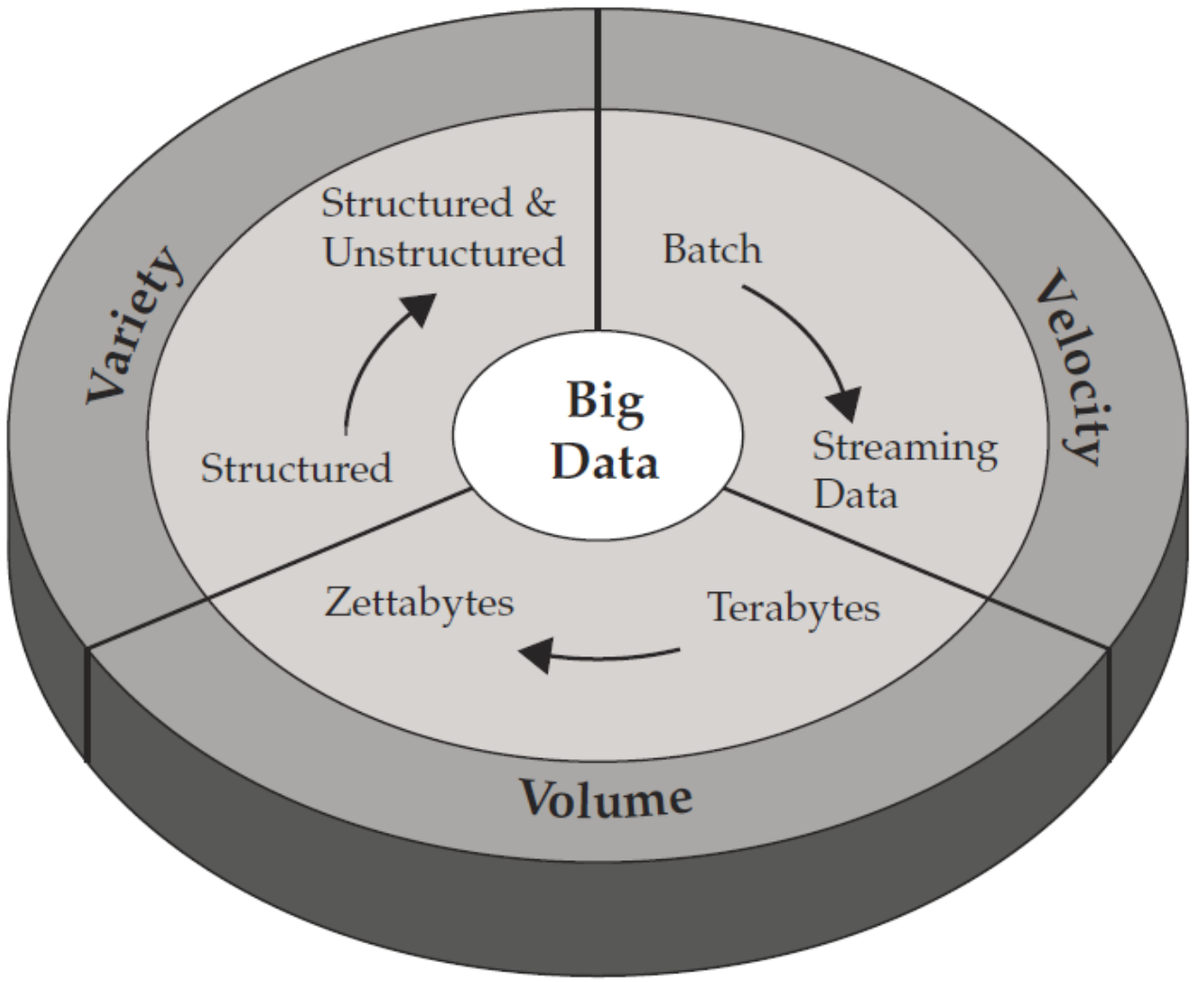
dario.colazzo@dauphine.fr

Course outline

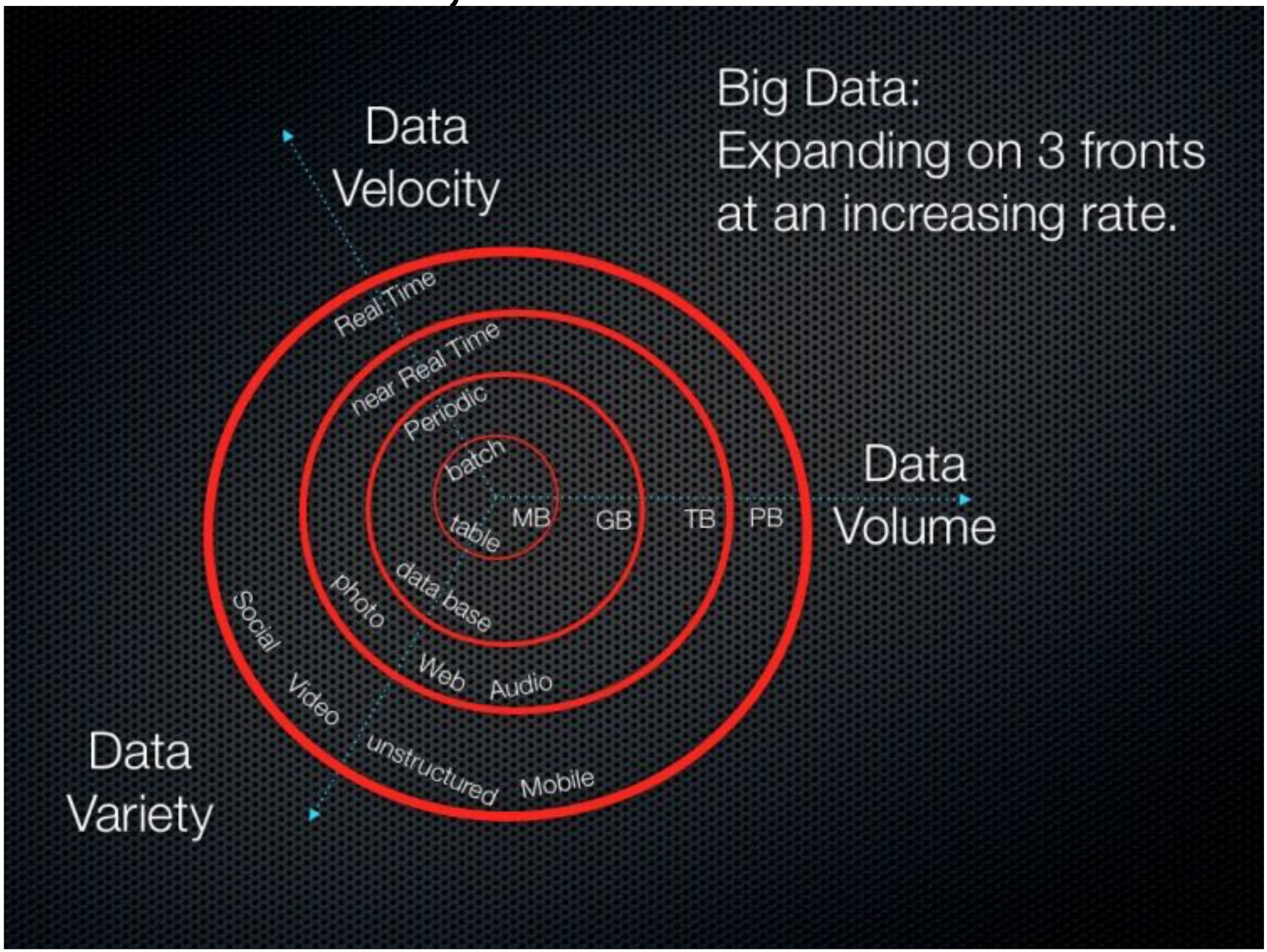
- Introduction to Big Data
- The ubiquitous frameworks: Hadoop, MapReduce and Spark
- Hands-on/lab-session

What is Big Data?

- Hint: you are part of it.
- The 3 V's characterising Big data



The 3 V's, more in detail



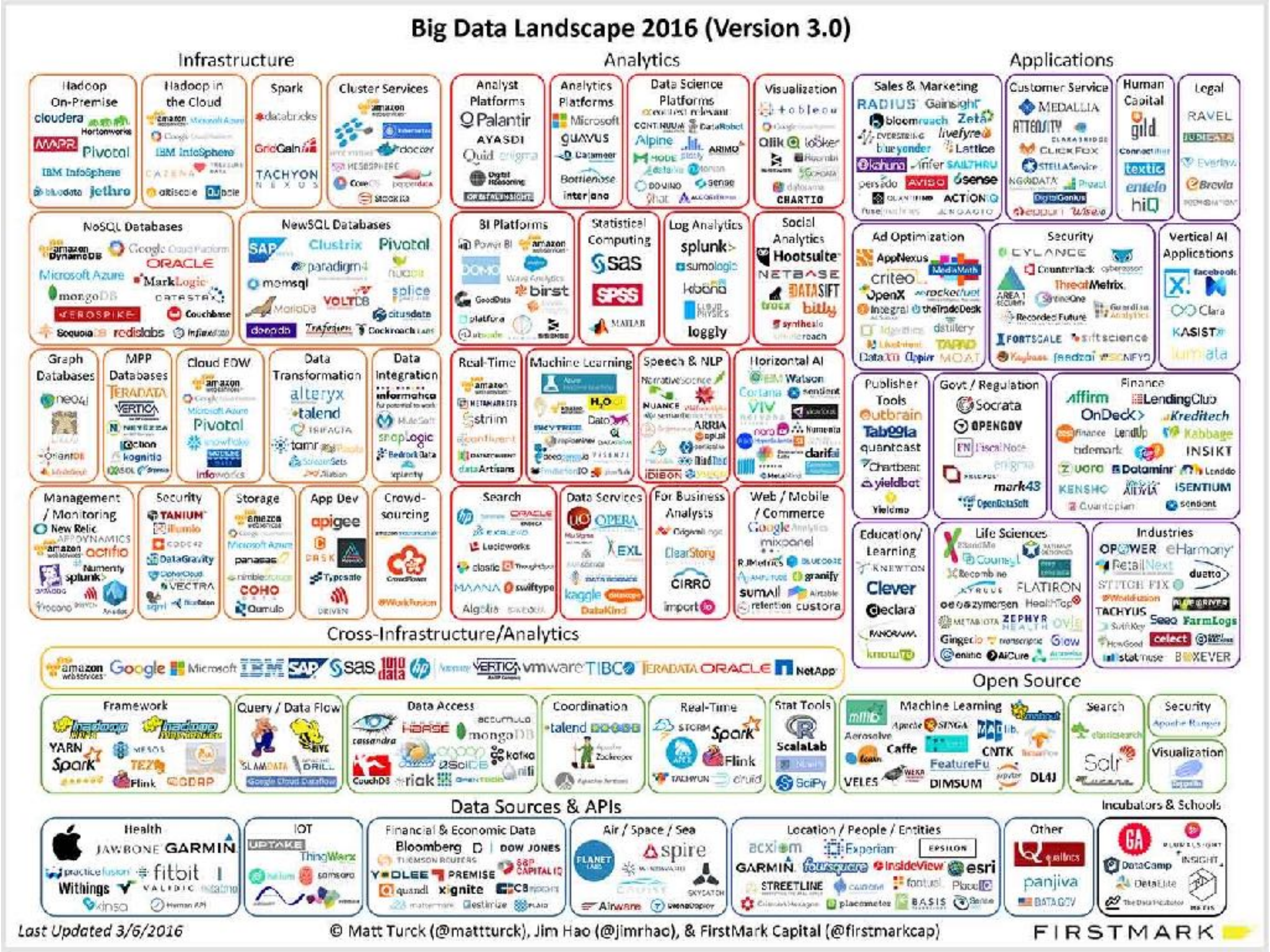
Enablers

- Increasing of storage capacity
- Increasing of processing power
- Availability of massive amounts of data

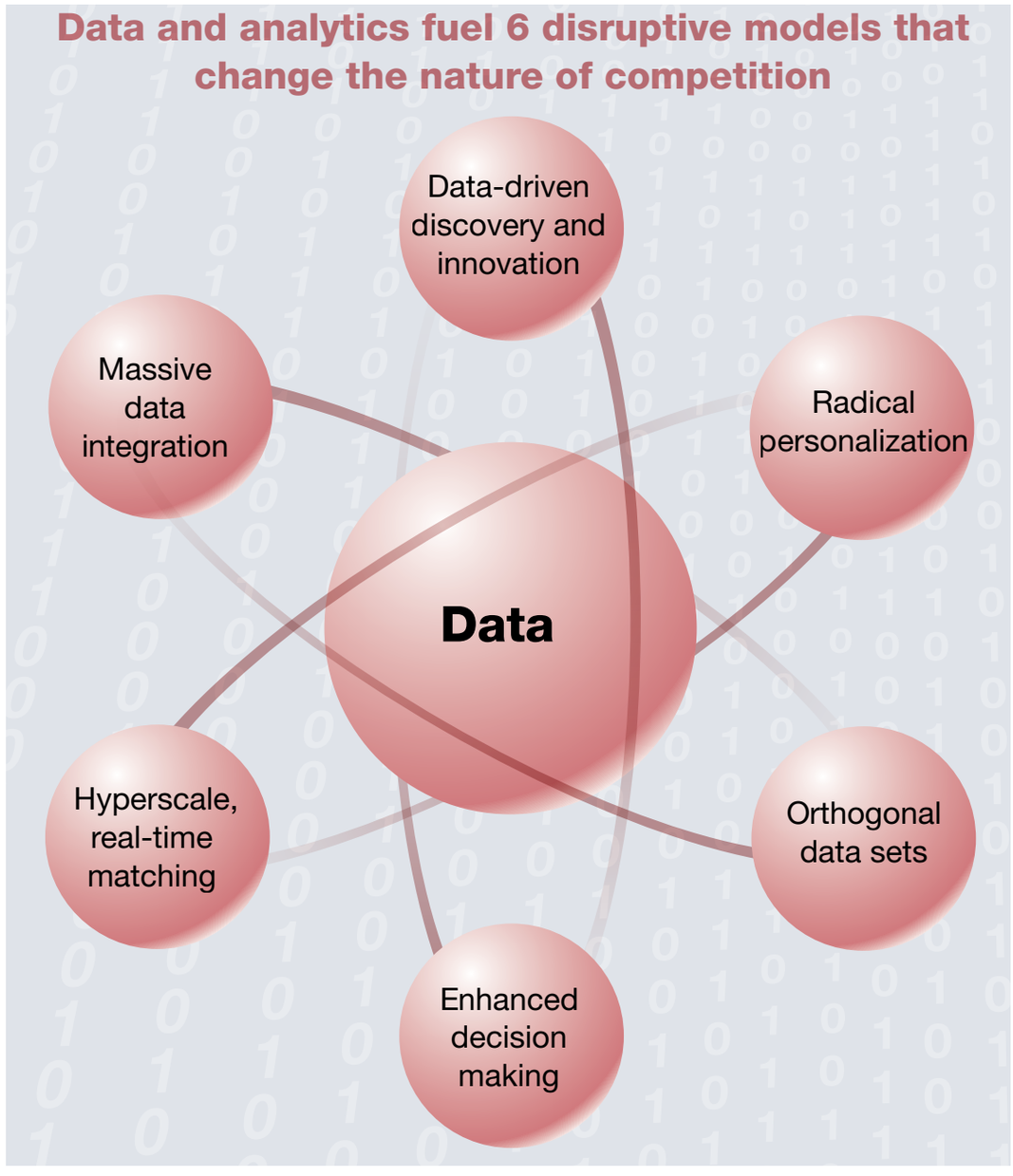
Enablers

- Increasing of storage capacity
- Increasing of processing power
- Availability of massive amounts of data

Are we missing anything?


Big Data Landscape 2016 (Version 3.0)
Infrastructure
Analytics
Applications
Hadoop On-Premise
Hadoop in the Cloud
Spark
Cluster Services
Analyst Platforms
Analytics Platforms
Data Science Platforms
Visualization
Sales & Marketing
Customer Service
Human Capital
Legal
NoSQL Databases
NewSQL Databases
BI Platforms
Statistical Computing
Log Analytics
Social Analytics
Ad Optimization
Security
Vertical AI Applications
Graph Databases
MPP Databases
Cloud EDW
Data Transformation
Data Integration
Real-Time
Machine Learning
Speech & NLP
Horizontal AI
Publisher Tools
Govt / Regulation
Finance
Management / Monitoring
Security
Storage
App Dev
Crowd-sourcing
Search
Data Services
For Business Analysts
Web / Mobile / Commerce
Education/ Learning
Life Sciences
Industries
Cross-Infrastructure/Analytics
Open Source
Framework
Query / Data Flow
Data Access
Coordination
Real-Time
Stat Tools
Machine Learning
Search
Security
Visualization
Data Sources & APIs
Incubators & Schools
Health
IOT
Financial & Economic Data
Air / Space / Sea
Location / People / Entities
Other
Last Updated 3/6/2016
© Matt Turck (@mattturck), Jim Hao (@jimrhao), & FirstMark Capital (@firstmarkcap)
FIRSTMARK

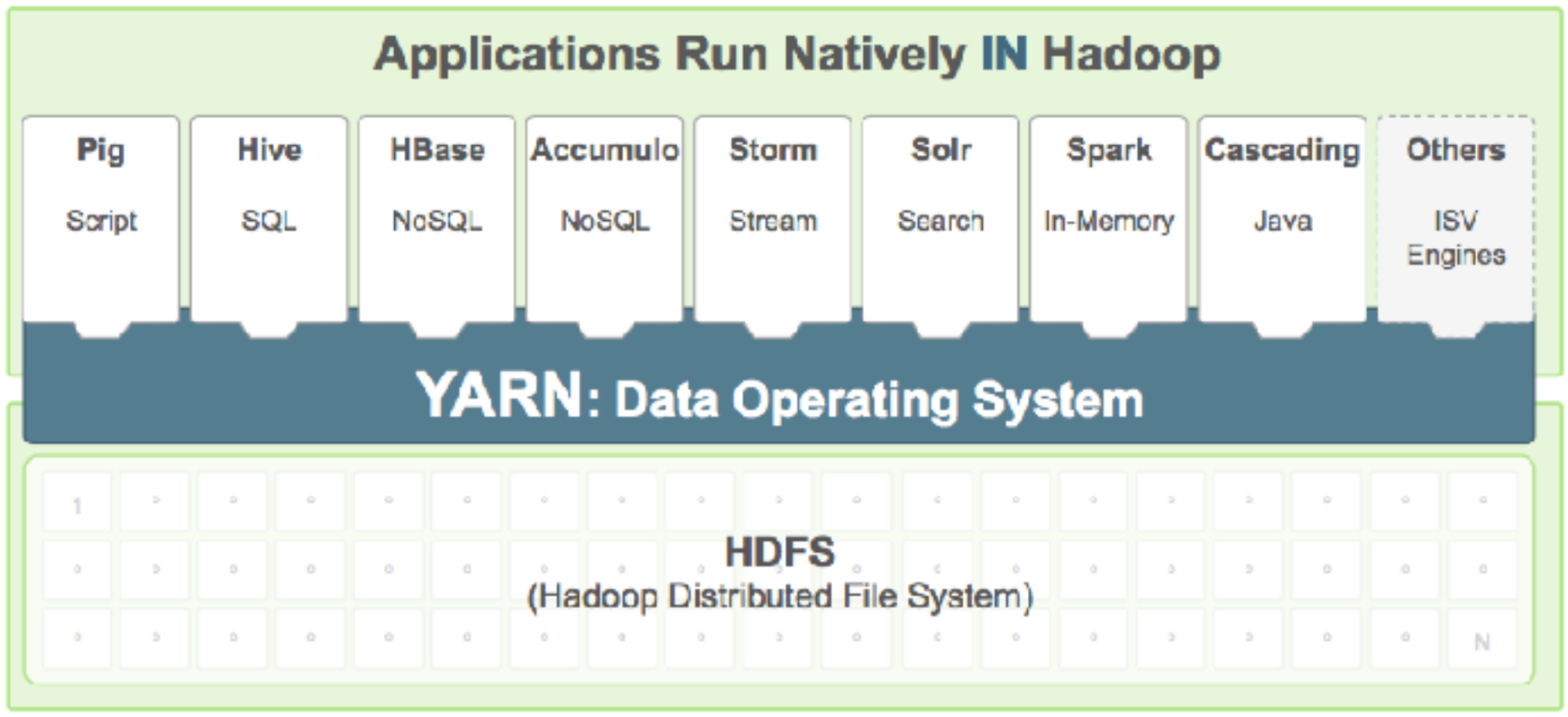
Trends for data analytics



The age of analytics: Competing in a data-driven world.
Report McKinsey Global Institute December 2016

Fortunately we have a winner

Typical Hadoop stack:



Three main steps in data analytics

- **Data generation and collection:** The source and platform where data are initially captured.
- **Data aggregation:** Processes and platforms for combining data from multiple sources.
- **Data analysis:** The gleaning of insights from data that can be acted upon.

The age of analytics: Competing in a data-driven world.
Report McKinsey Global Institute April 2016

- Hadoop ecosystem plays a crucial role in *each* of them
- The first twos are about *data preparation*: at least 50% of the data scientist work!
- As seen before, companies often struggle in recruiting and retain talents for each of these 3 tasks

The Hadoop Distributed Filesystem - HDFS

- Highly scalable, distributed, load-balanced, portable, and fault-tolerant (with built-in redundancy at the software level) *storage component* of Hadoop.

- It provides a layer for storing Big Data in a traditional, hierarchical, Linux-like file organization of directories and files.

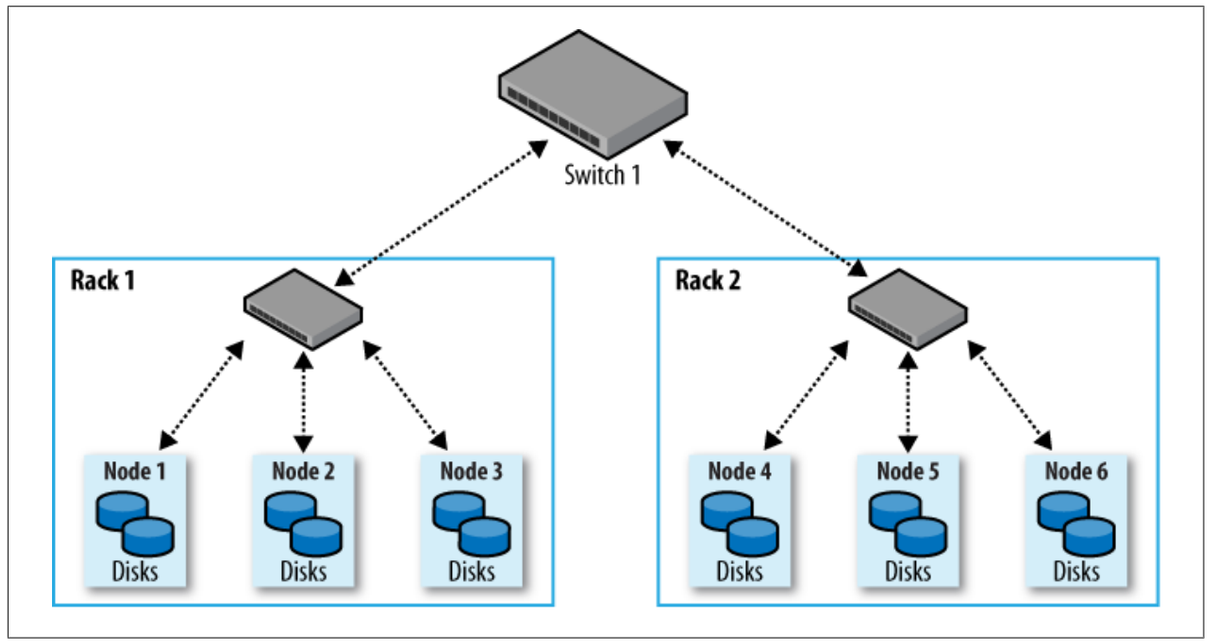
- It has been designed to run on *commodity* hardware.

Main assumptions behind its design

- Horizontal scalability
- Fault tolerance
- Capability to run on commodity hardware
- Write once, read many times
- Data locality
- File system namespace, relying on traditional hierarchical file organization.
- Streaming access and high throughput:
    - reading the data in the fastest possible way (instead of focusing on the speed of the data write).
    - reading data from multiple nodes, in parallel.

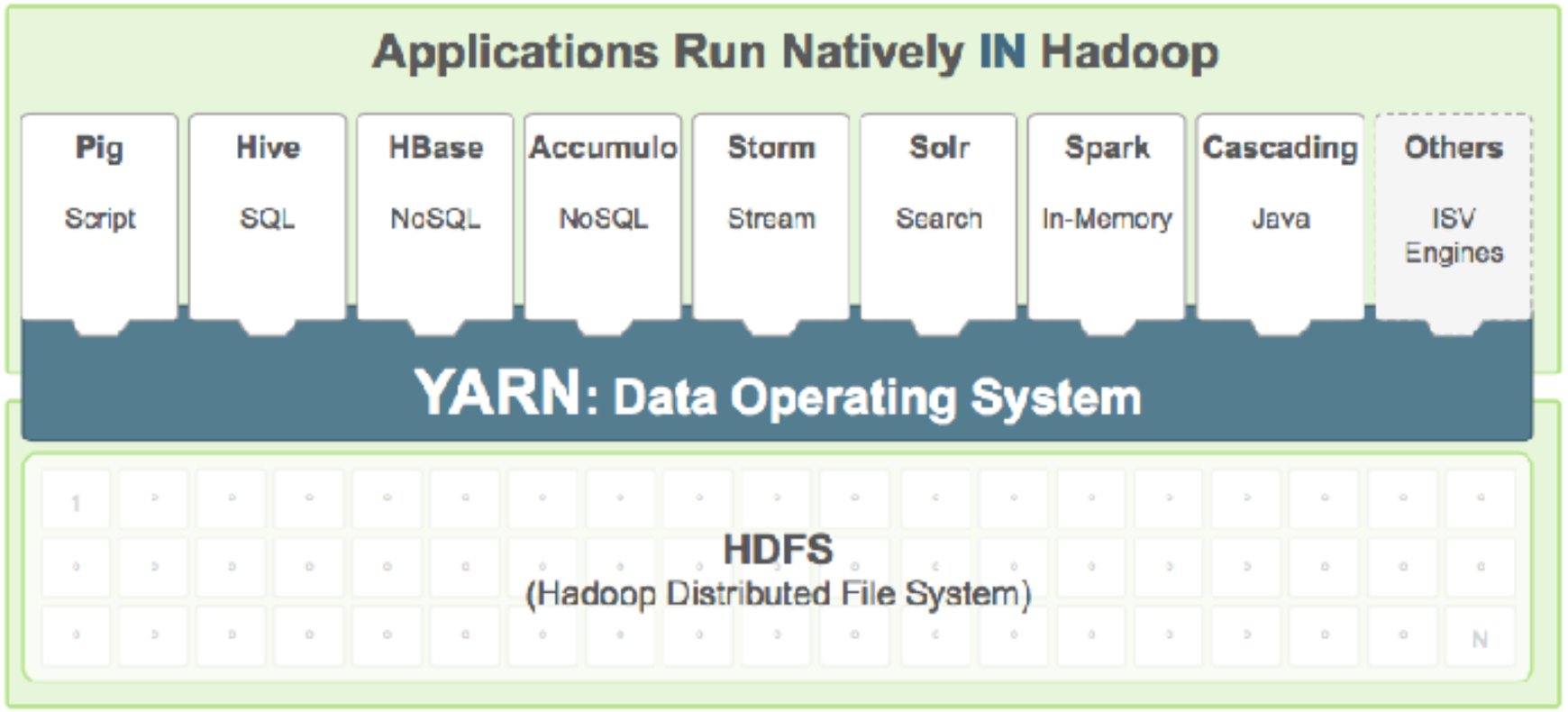
Typical cluster architecture

- One or more racks.
- Typically 30 to 40 node servers per rack with a 1GB switch for the rack.
- The cluster switch is normally 1GB or 10GB.
- Architecture of single node server can vary.
  - *More disk capacity and network throughput* for operations like indexing, grouping, data importing/ exporting, data transformation.
  - *More CPU capacity* for operations like clustering/classification, NLP, feature extraction.



- Example of balanced single node architecture proposed by Cloudera.
  - 2-24 1-4TB hard disks in a JBOD (Just a Bunch Of Disks) configuration (no RAID)
  - 2 quad-/hex-/octo-core CPUs, running at least 2-2.5GHz
  - 64-512GB of RAM

HDFS

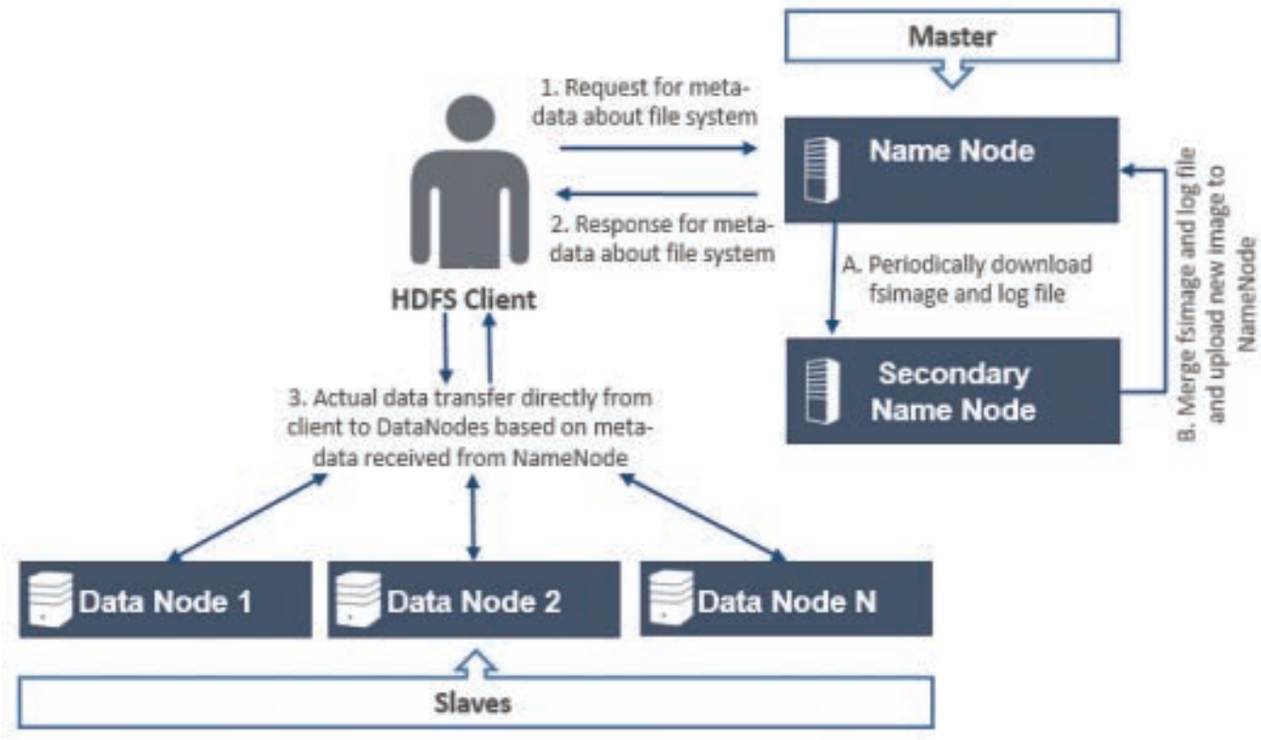


References : *Hadoop: The Definitive Guide* - Tom White.

Apache Hadoop Yarn - Arun C.Murty, Vinod Kumar Vavilapalli, et al.

Big Data Analytics with Microsoft HDInsight - Manpreet Singh, Ashrad Ali.

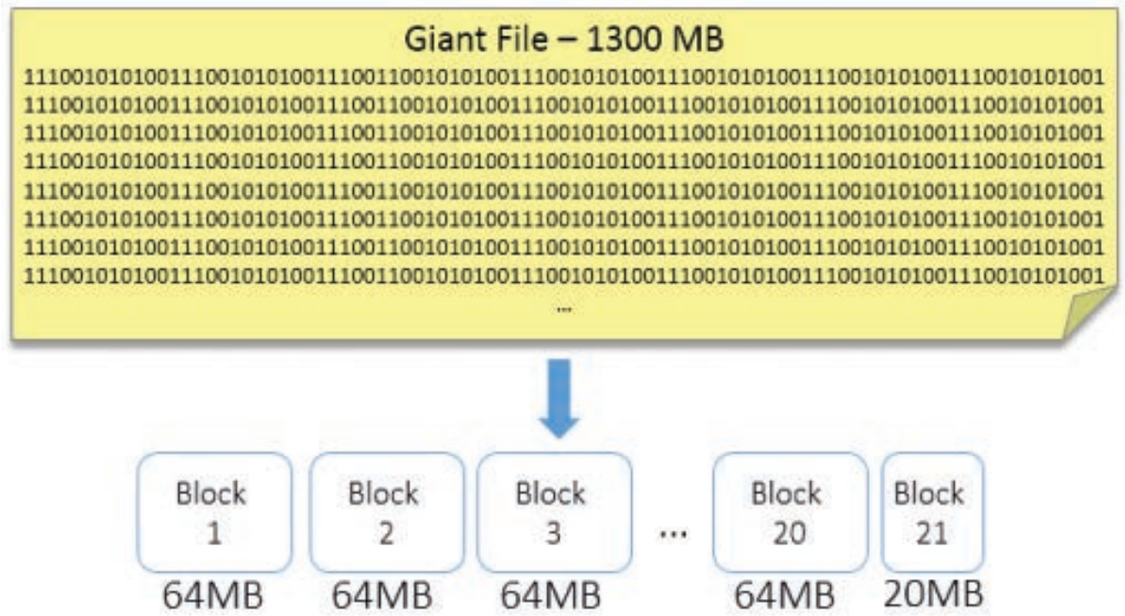
Name nodes and Data nodes



- Name node = a server node running the Name node daemon (service in Windows)
- Data Node = a server node running the Data node daemon

- To store a file, HDFS client asks meta information to the Name node
- The client then interacts *only* with Data Nodes
- It splits the file into one or more **chunks** or **blocks** (64 MB by default, configurable)
- And send them to a set of Data Nodes slaves previously indicated by the Name node
- Each block is replicated n times (n=3 by default, configurable)

File split in HDFS



- In Hadoop you have plenty of configuration files
- For instance, block size is set in the hdfs-site.xml file

Name	Value	Description
`dfs.blocksize`	134217728	The default block size for new files, in bytes. You can use the following suffix (case insensitive): k (kilo), m (mega), g (giga), t (tera), p (peta), e (exa) to specify the size (such as 128k, 512m, 1g, and so on). Or, provide the complete size in bytes, such as 134217728 for 128MB.

Block placement and replication

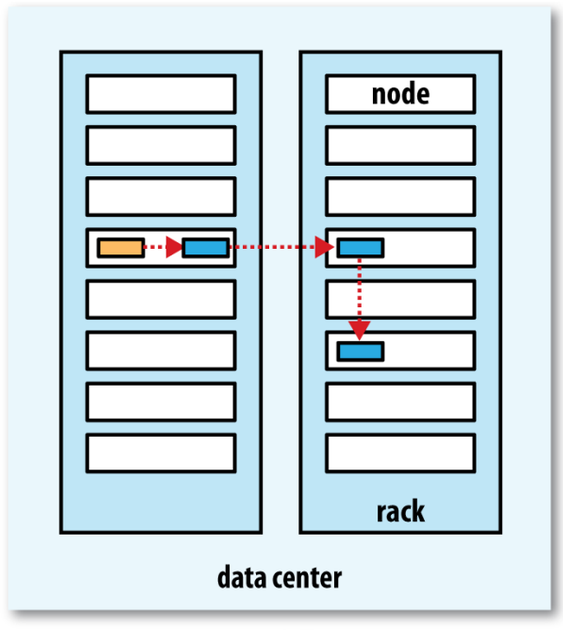
- By default each block is stored 3 times in three different Data nodes, *for fault tolerance*
- When a file is created, an application can specify the number of replicas of each block of the file that HDFS must maintain. The upper bound dfs.replication.max must be respected.
- Settings in hdfs-site.xml

Name	Value	Description
`dfs.replication`	3	Default block replication. The actual number of replications can be specified when the file is created. The default is used if replication is not specified in create time.
`dfs.replication.max`	512	Maximum block replication.
`dfs.namenode.replication.min`	1	Minimal block replication.

Block placement and replication

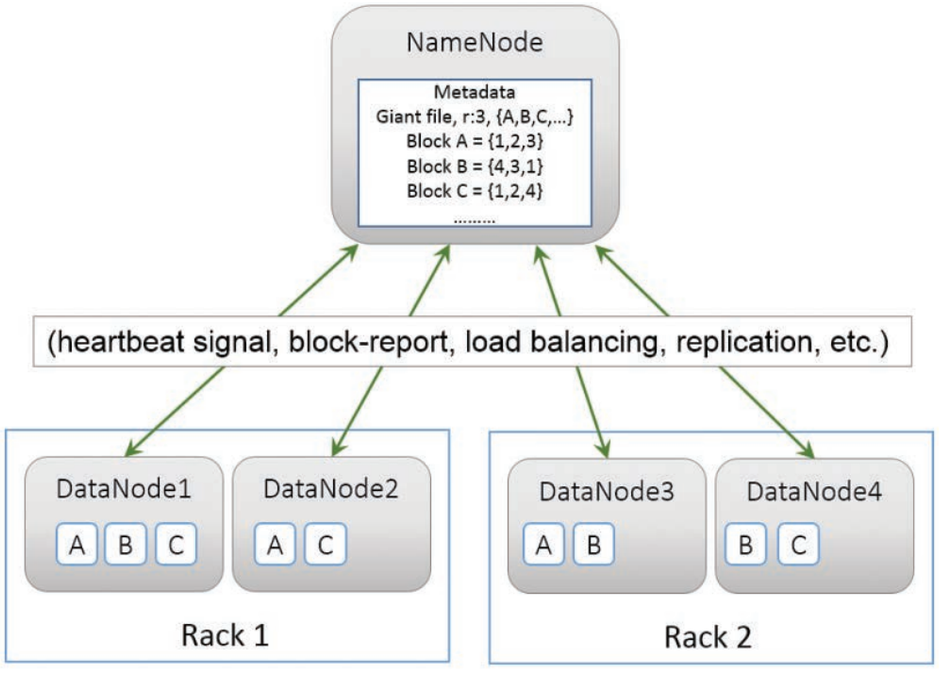
- For robustness, the ideal approach would be to store block replicas in different racks.
- For efficiency, it is better to store all replicas in the same rack.
- Balanced Hadoop approach: store one block on the *client* Data node where the file originates (or a not too busy node chosen by the Name node) and the two other blocks in a different rack (if any).
  - This requires to configure the cluster for RackAwareness

    https://hadoop.apache.org/docs/r2.7.2/hadoop-project-dist/hadoop-common/RackAwareness.html



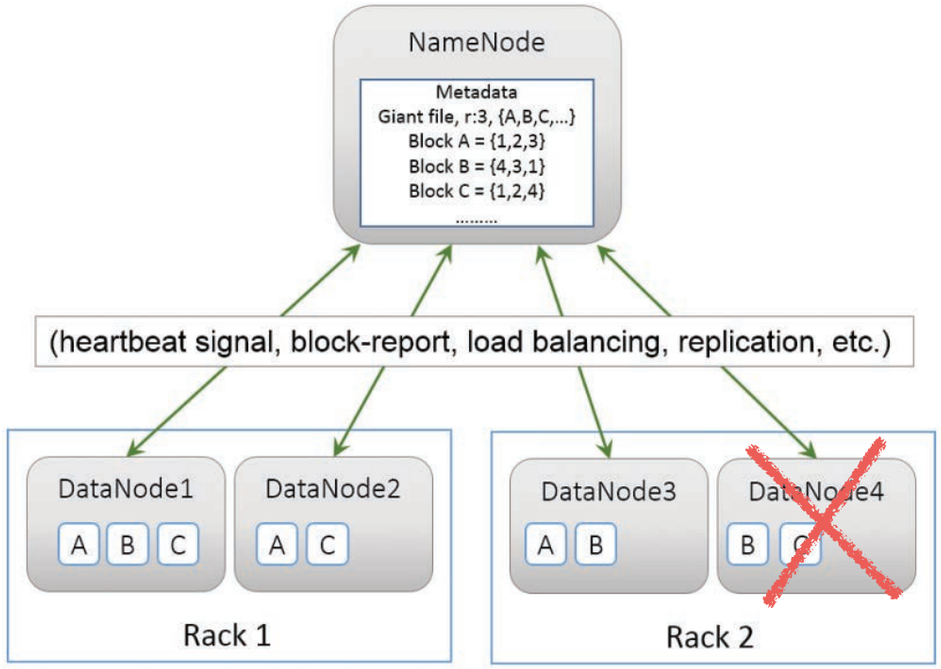
Heartbeats

- All data nodes periodically (each 3 seconds) send **heartbeat** signals to the name node.
- They contain crucial information about stored blocks, percentage of used storage, current communication load, etc.
- Hearth beat contents are crucial for the Name Node to build and maintain metadata information
- The NameNode does not *directly* call the Data Nodes. It uses replies to heartbeats to ask replication to other nodes, remove local block replicas, etc.



Node failure and replication

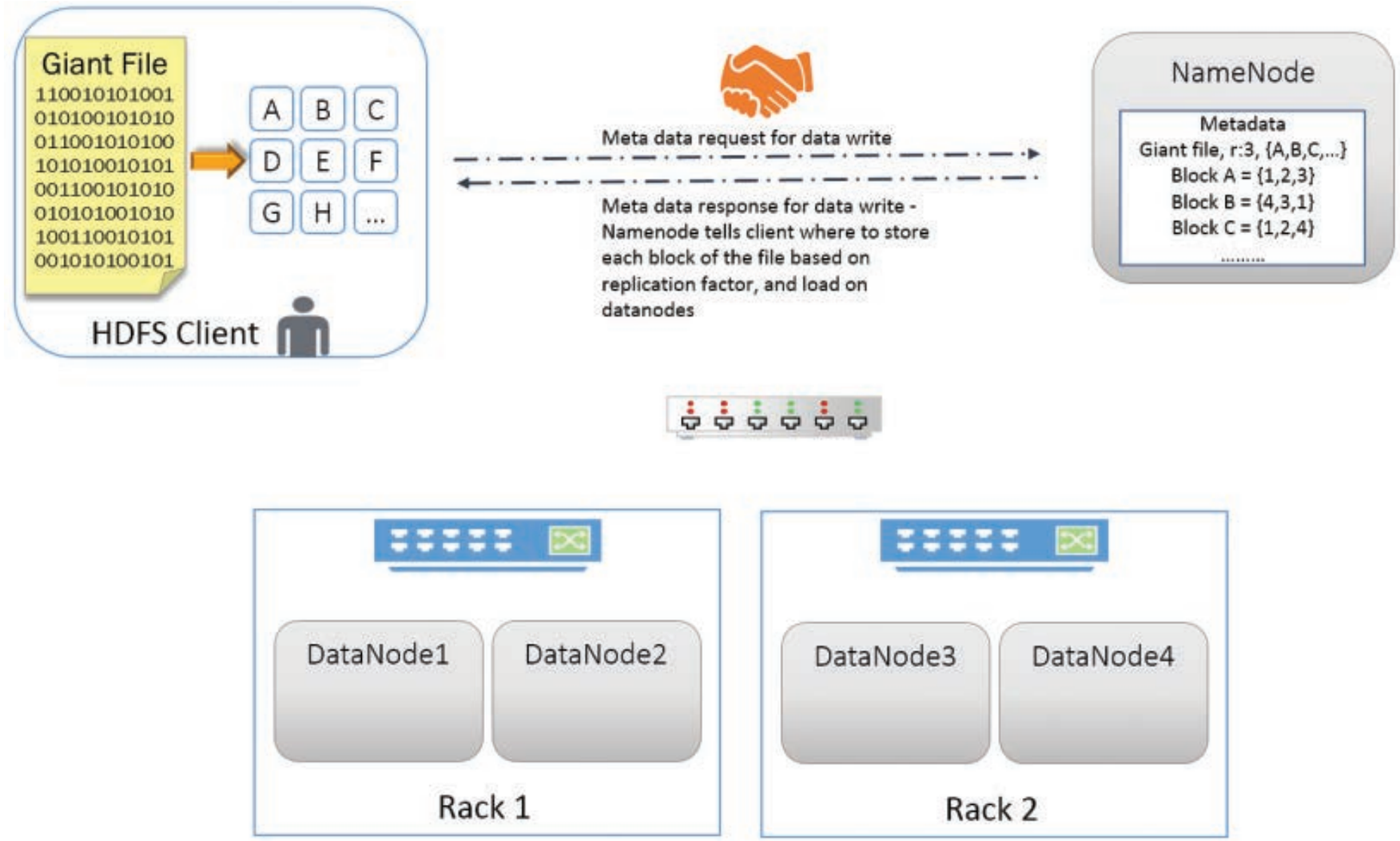
- Assume Data node 4 stops working
- This means that no heart beats is sent anymore
- The Name node then instructs another living Data Node including blocks B and C to *replicate* them on other Data Nodes.
- Data transmission for B and C replication does not involve the Name Node.



Writing a file to HDFS

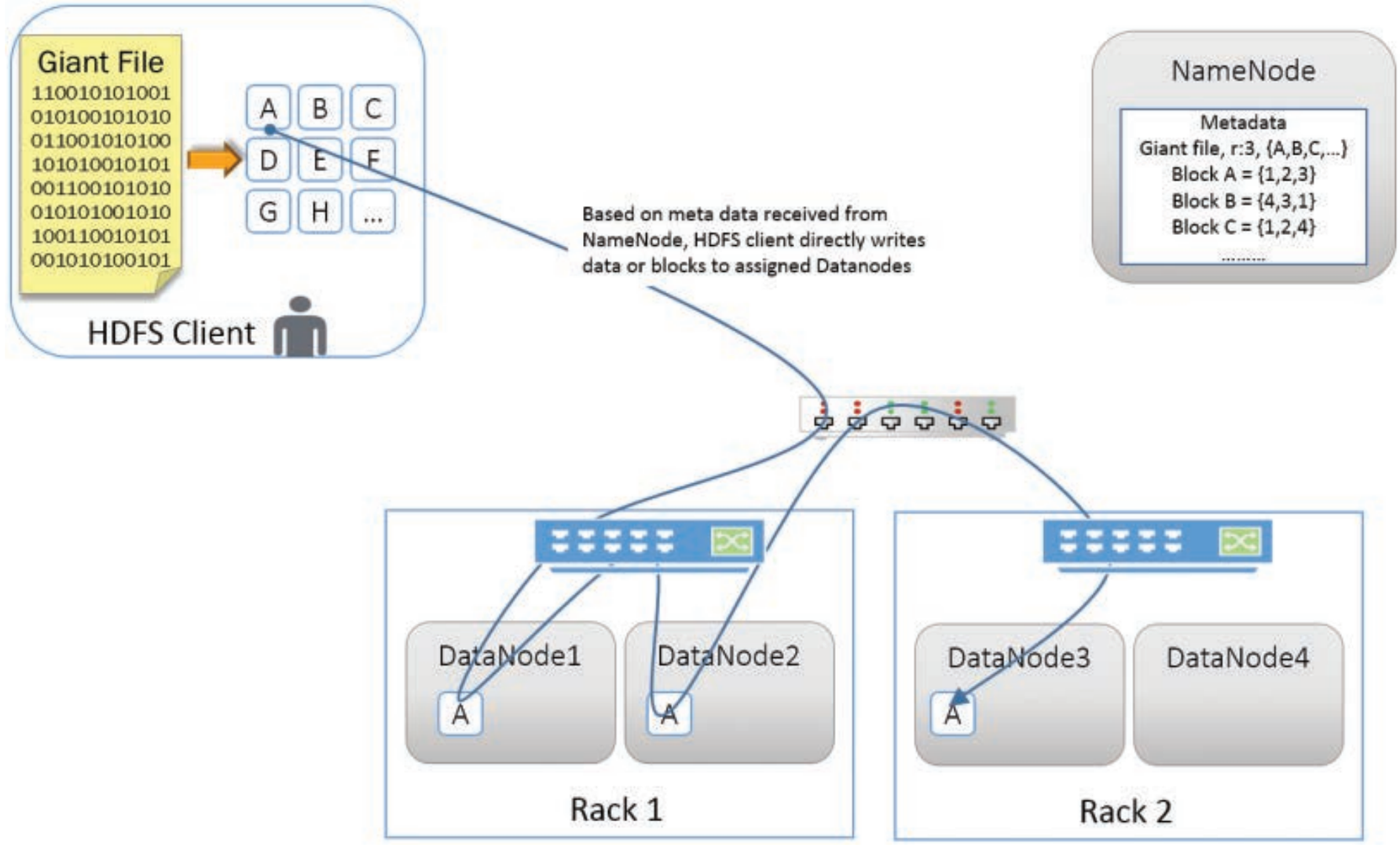
- This can happen, for instance, by command-line or by means of a client program requesting the writing operation.

- First step:



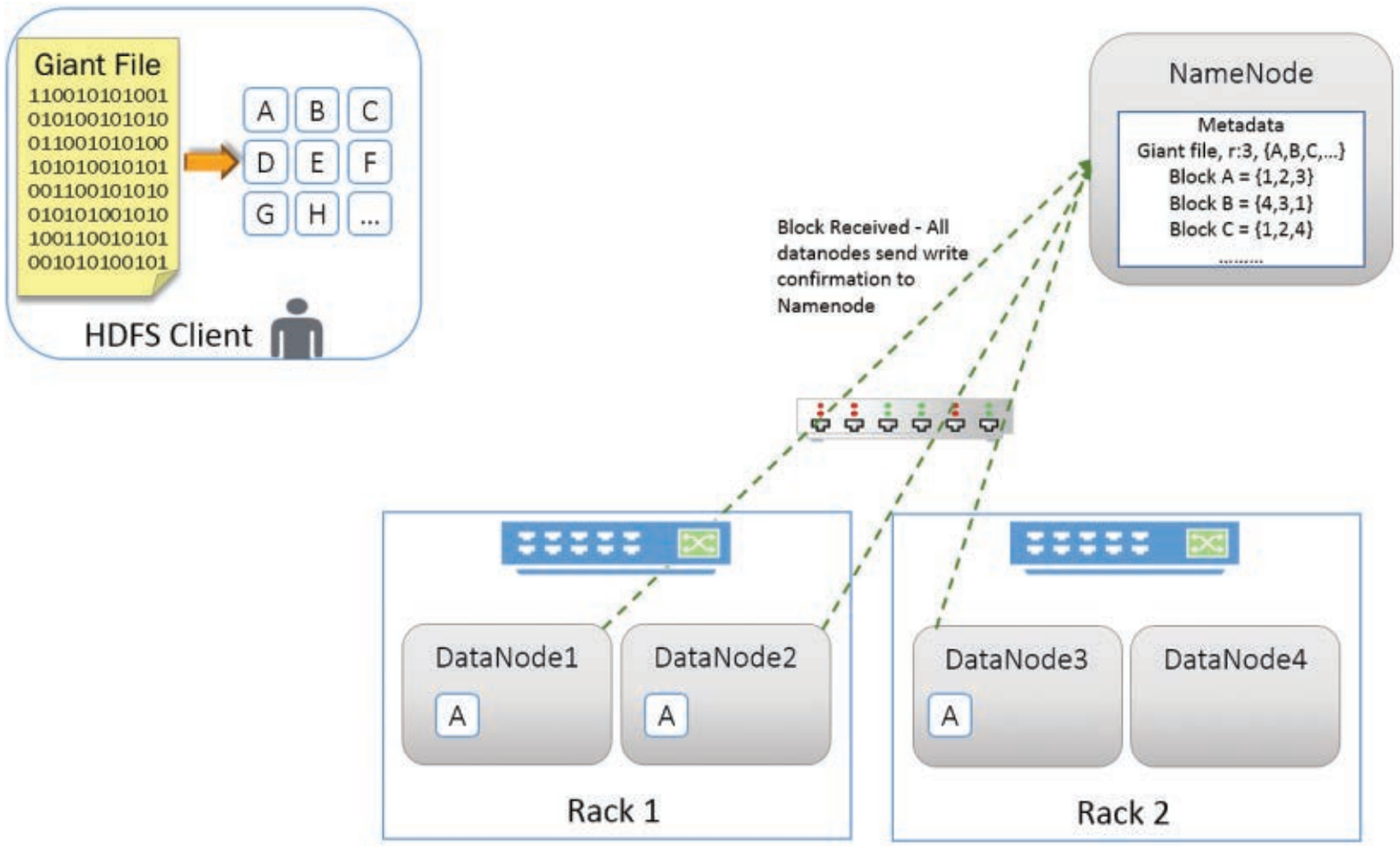
Writing a file to HDFS

- Second step, the first block is sent to Data Nodes:



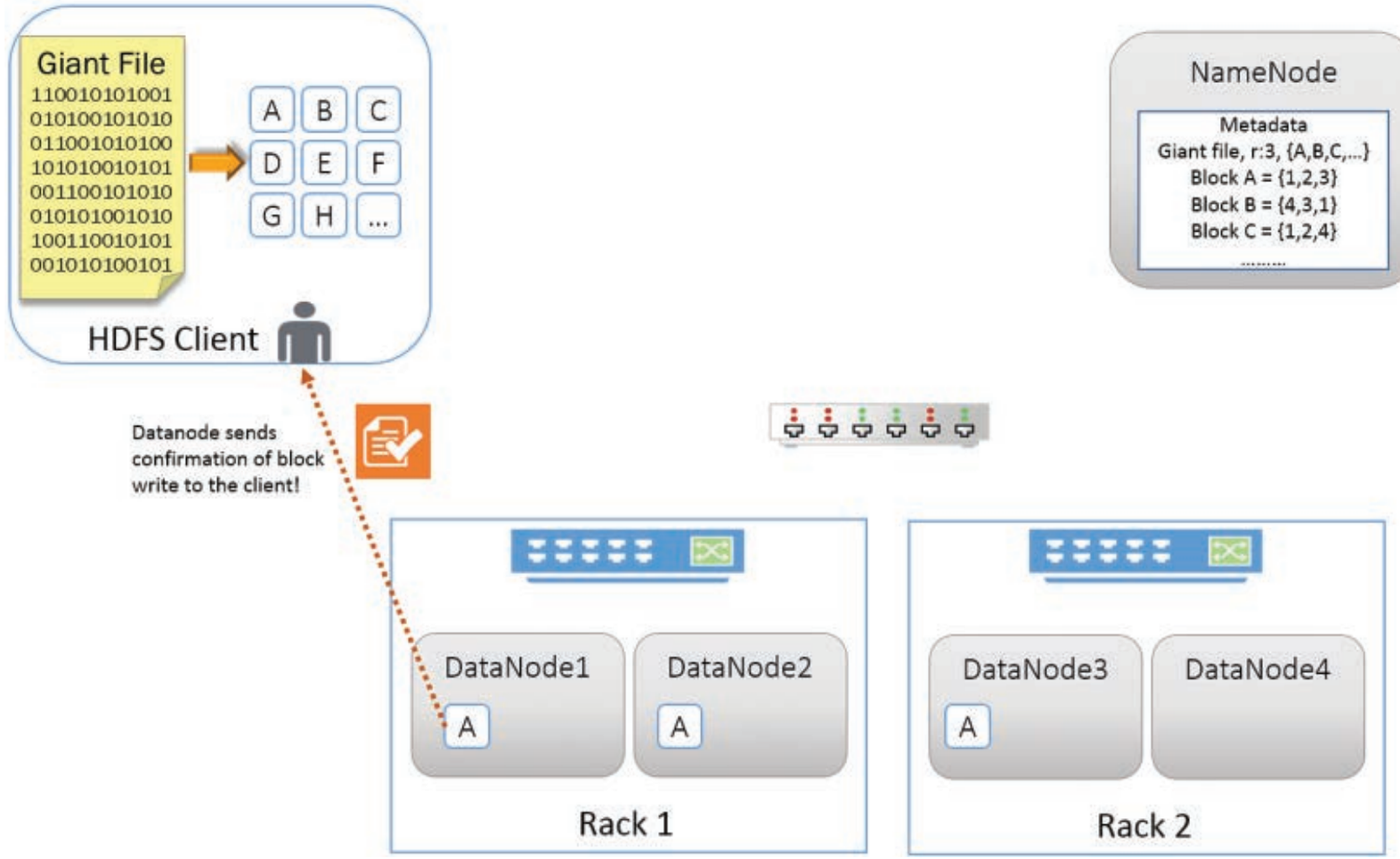
Writing a file to HDFS

- Third step, Data Nodes notify the Name Node about the stored block replicas



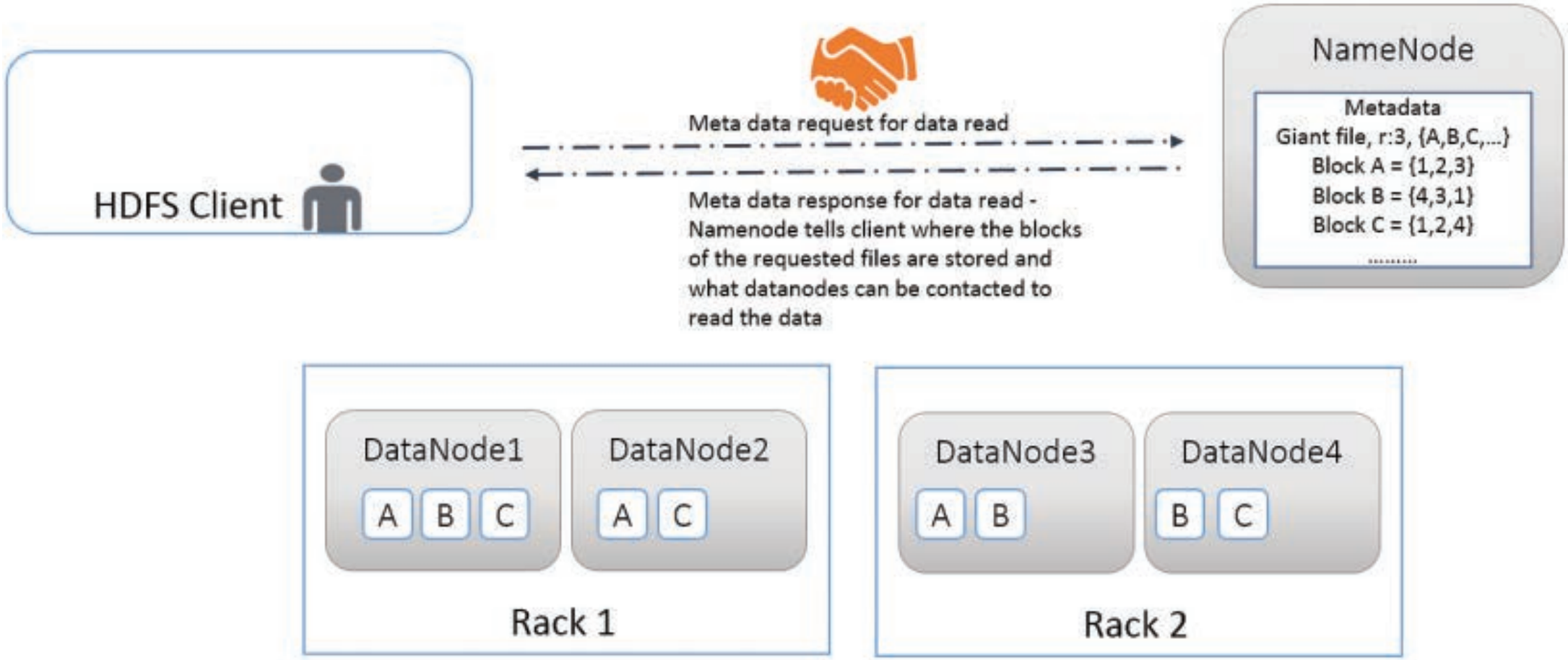
Writing a file to HDFS

- Fourth step, the first Data Node storing the block sends conformation to the client.



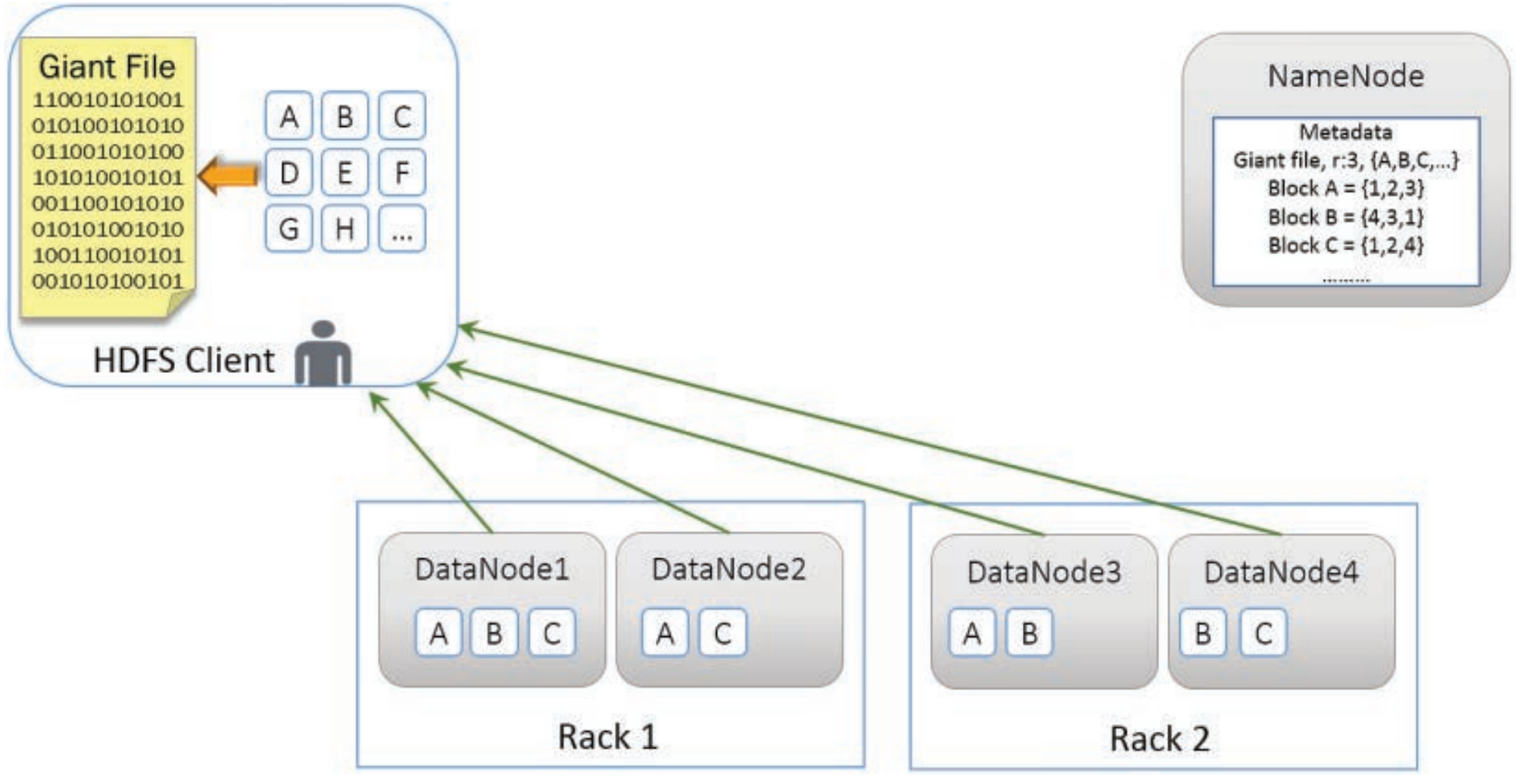
Reading a file from HDFS

- First step.



Reading a file from HDFS

- Second step.



Accessing and Managing HDFS

- HDFS command-line interface (CLI), or FS Shell

  http://hadoop.apache.org/docs/current/hadoop-project-dist/hadoop-common/FileSystemShell.html

- Leverage the Java API available in the classes of the `org.apache.hadoop.fs`

  http://hadoop.apache.org/docs/current/api/org/apache/ hadoop/fs/package-frame.html

- By means of high level languages (e.g., Pig Latin, Hive, Scala in Spark)

FS Shell, examples

- Creating a directory

```
> hdfs dfs -mkdir /example/sampledata
```

- Copying a directory to HDFS (from the local FS)

```
> hdfs dfs -copyFromLocal \
/apps/dist/examples/data/gutenberg \
/example/sampledata
```

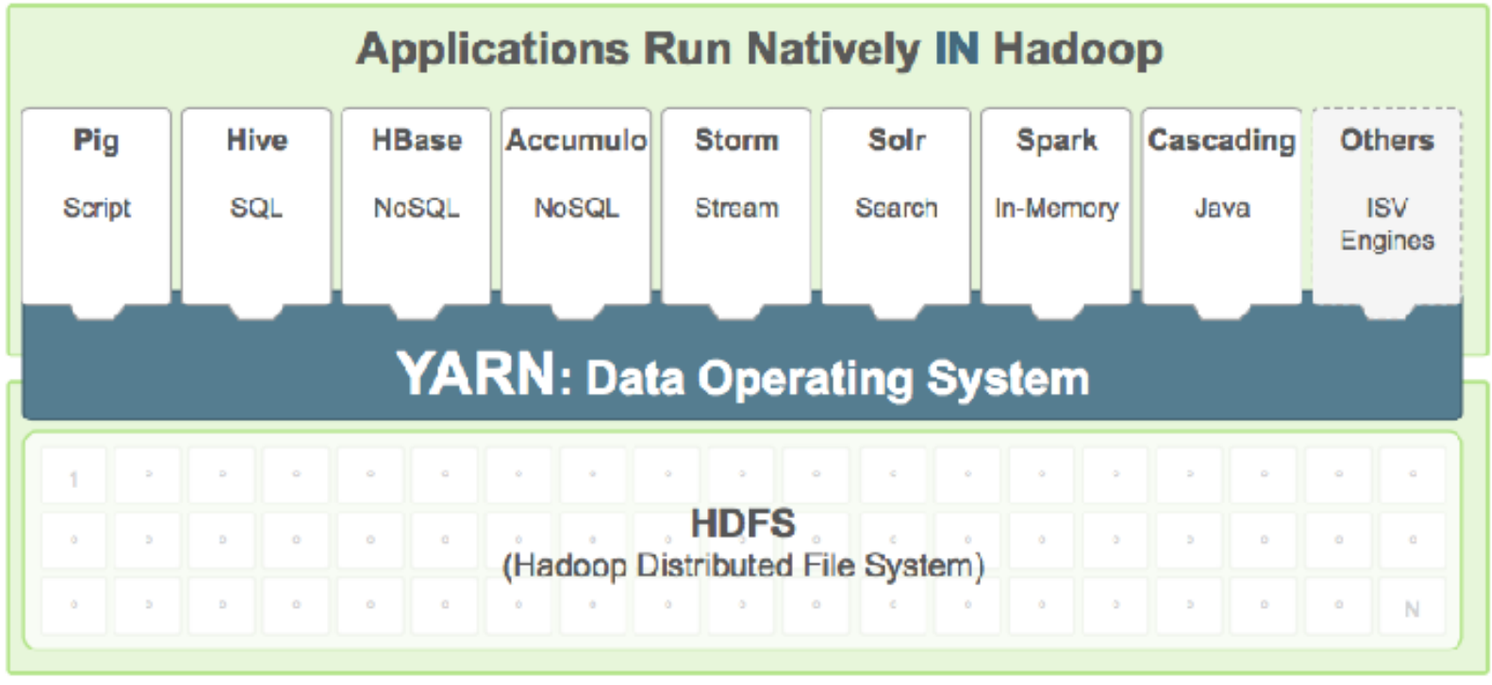
- Listing content of a directory

```
> hdfs dfs -ls /example/sampledata
```

- Copying a file to FS

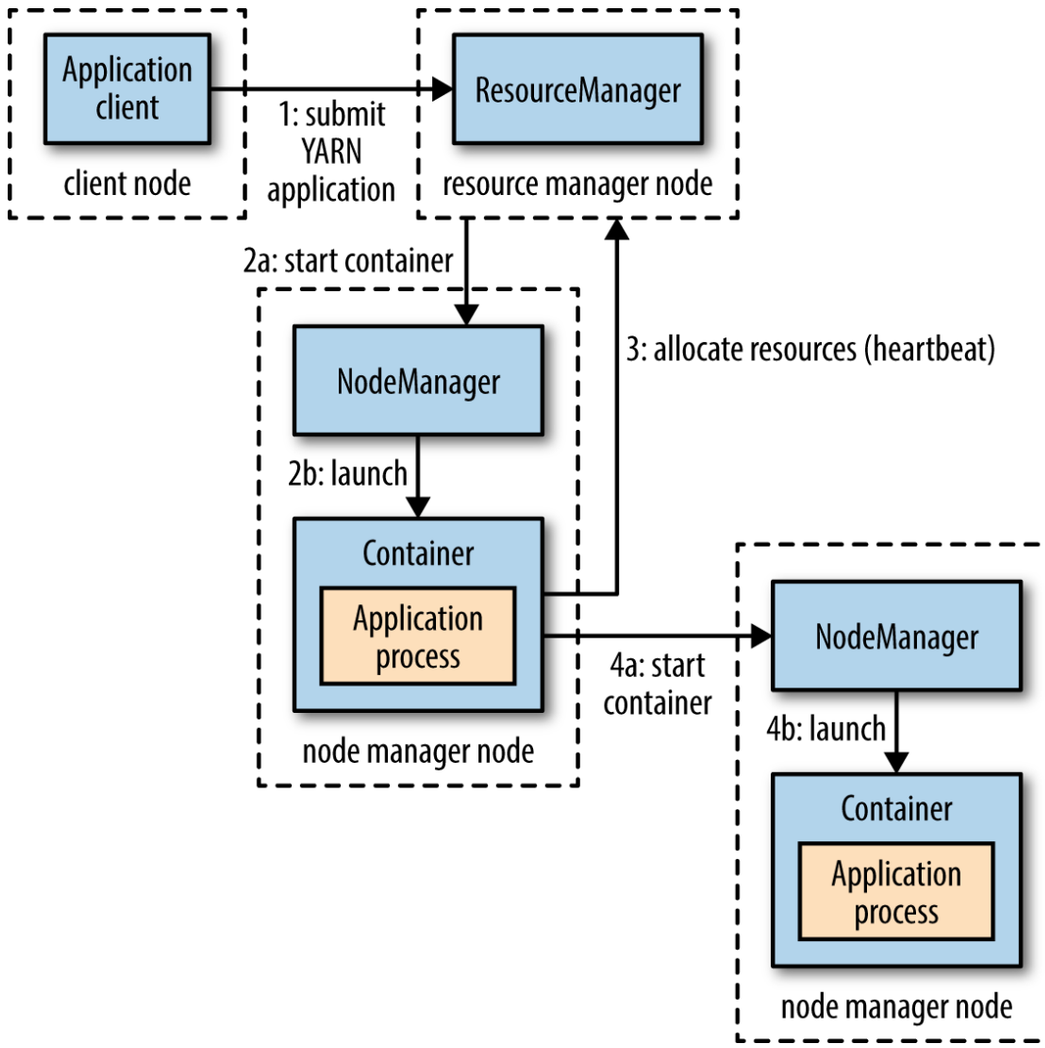
```
> hdfs dfs -copyToLocal  \
/user/hadoop/filename \
/apps/dist/examples/data/gutenberg
```

YARN


Applications Run Natively IN Hadoop
Pig
Script
Hive
SQL
HBase
NoSQL
Accumulo
NoSQL
Storm
Stream
Solr
Search
Spark
In-Memory
Cascading
Java
Others
ISV Engines
YARN: Data Operating System
HDFS
(Hadoop Distributed File System)

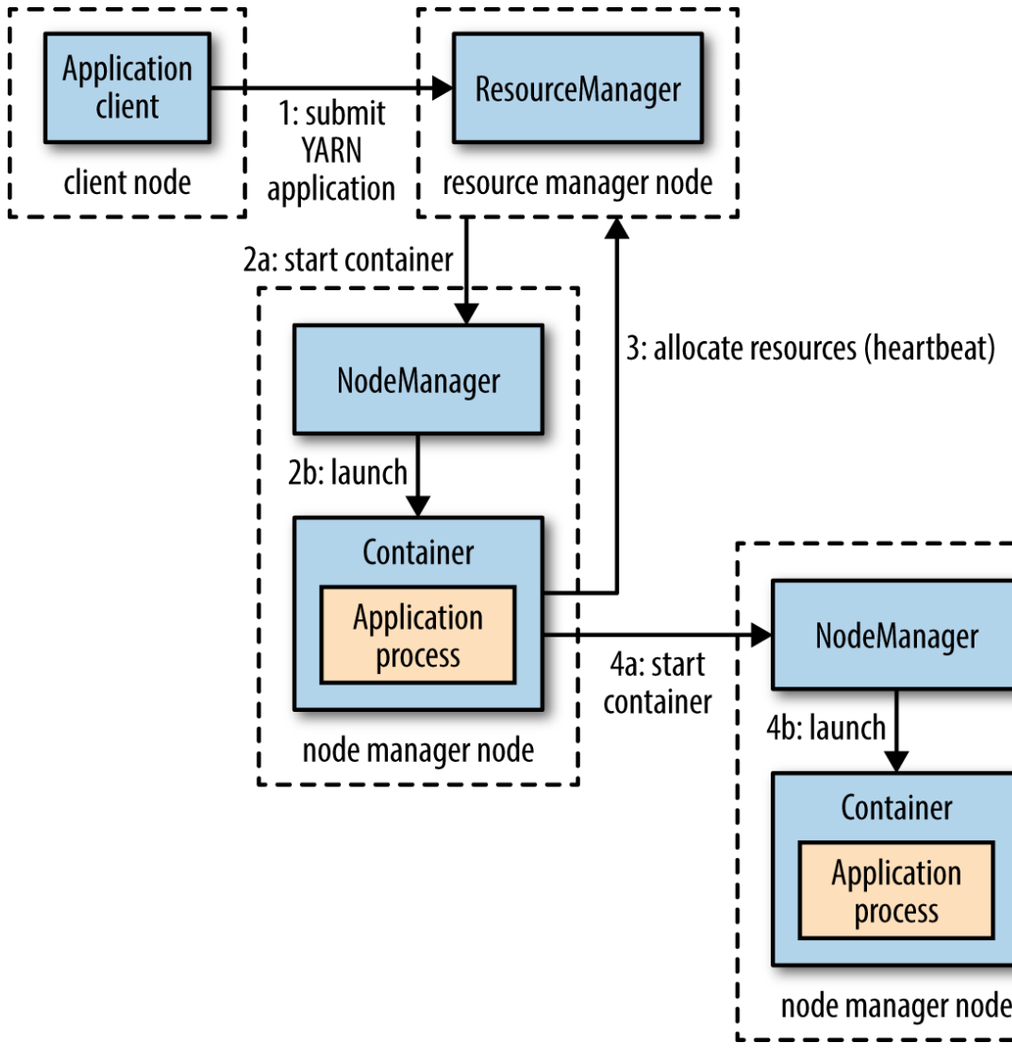
YARN

- YARN is a general purpose **data operating systems**
- It accepts requests of task executions on the cluster and allocate resources for them
- For instance YARN can accept requests for executing MR jobs or MPI programs on the same cluster
- The set of resources for a task on a given node is called ***container*** and it includes given amounts of memory space and CPU power (cores)
- YARN keeps track of allocated resources in order to schedule container allocation for new requests
- Containers can be demanded all in advance like for MR jobs and Spark tasks, or at run time

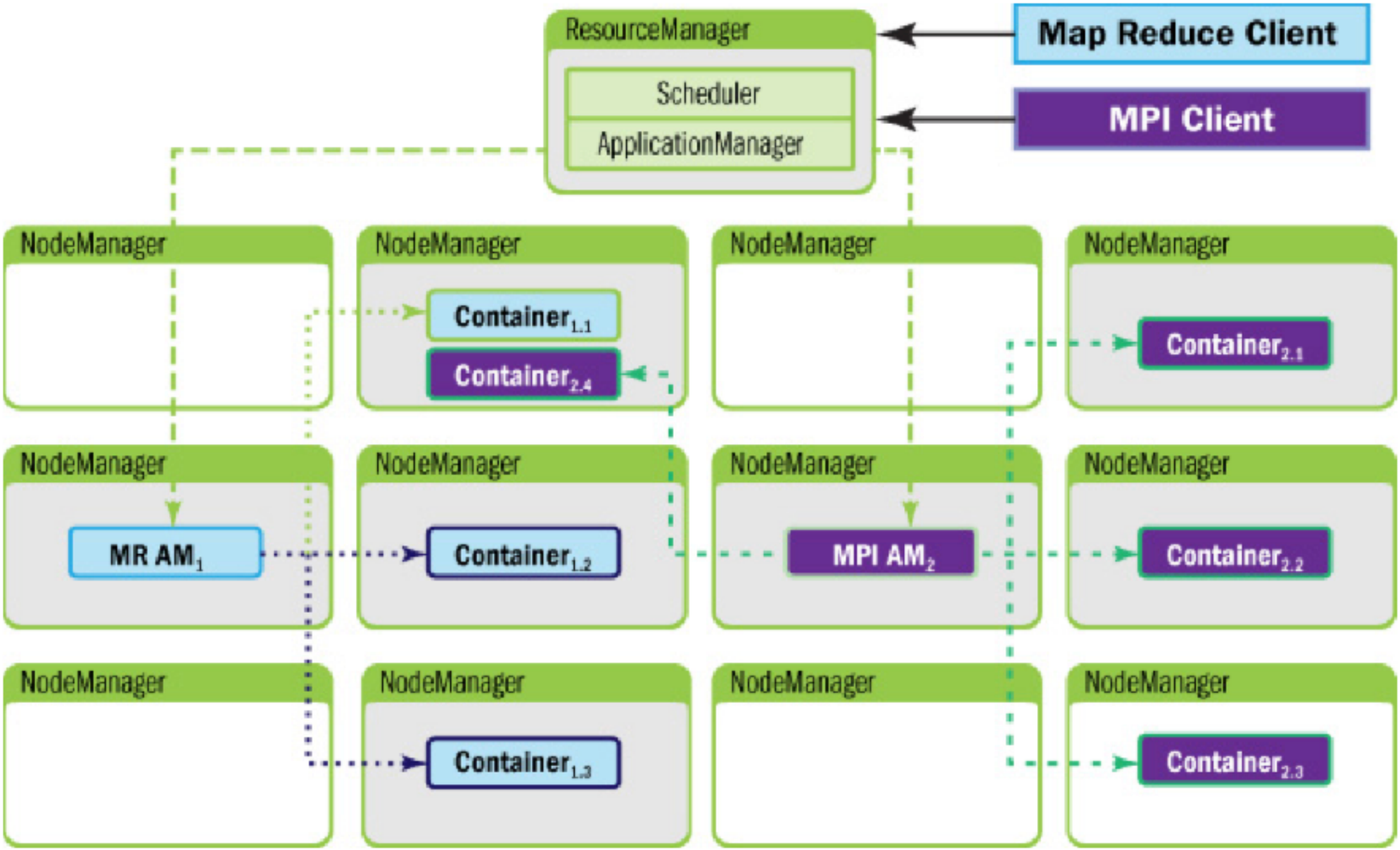


YARN

- Step 1: a client contact the **resource manager** (*running on the master node*)
- Step 2.a: the resource manager then finds a **node manager** (*running on a slave*) that can launch and manage the running operations of the application; this is the **application manager**.
- Step 2.b: the node manager allocates the container indicated by the resource manager. The application runs within the container
- Step 3: eventually, the application manager can request new containers to the resource manager
- Step 4: a parallel container is then started after the acknowledge of the resource manager. Started container informs the application manager about their status upon request.
- Containers can be demanded all in advance or at run time (step 4)
- At the end of the process, the application manager informs the resource manager, which will kill the allocated containers.



YARN as a data OS



MapReduce

Why MapReduce

- After more than 10 years since its introduction, it still plays a crucial role
- Powerful paradigm to express and implement parallel algorithms that scale
- At the core of its evolutions : Pig Latin, Hive, Spark, Giraph, Flink.
- Companies adopts and maintain a considerable amount of MapReduce code

MapReduce

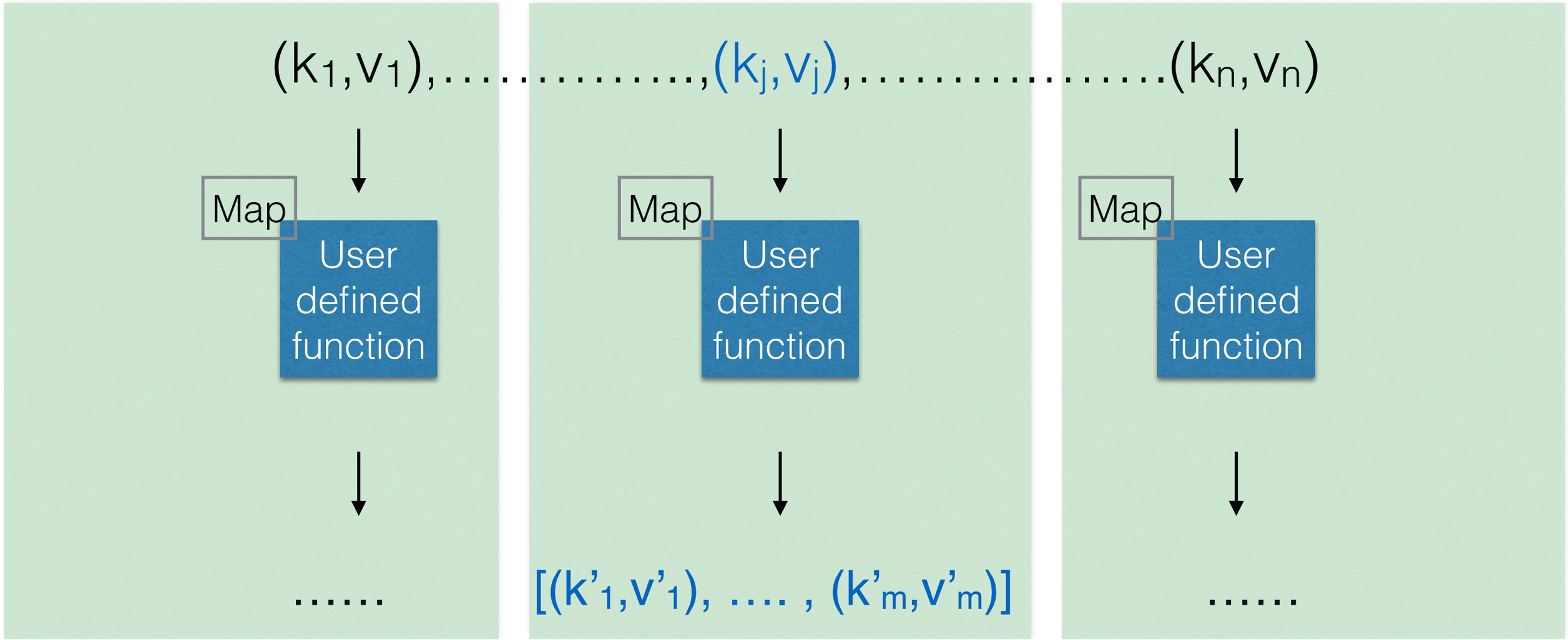
- It is a paradigm to design algorithms for large scale data analytics.
- Several programming languages can be used to implement MapReduce algorithms (Java, Python, C++, etc.)
- Its main runtime support is Hadoop
- Starting from its 2.0 version, Hadoop is a general purpose run time support for large scale data processing, and supports in particular MapReduce

Main principles

- Data model: data collections are represented in terms of collections of key-value pairs (k, v)
- Paradigm model: a MapReduce algorithm (or job) consists of two functions Map and Reduce specified by the developer
- Actually, three phases at least during data processing
  - **Map** phase
  - **Shuffle and sort** phase, pre-defined
  - **Reduce** phase

MAP phase

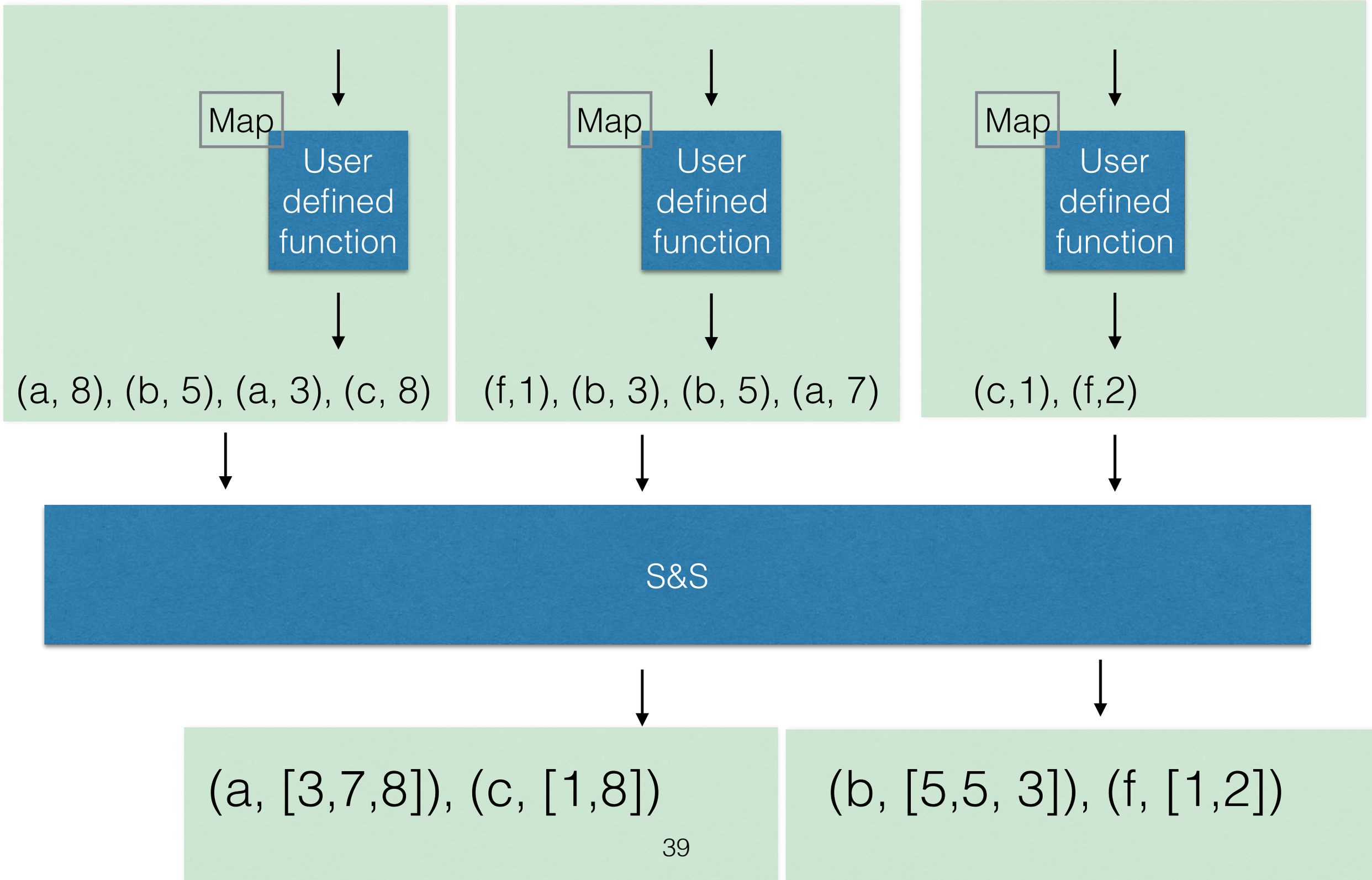
- The **Map** *second-order* function is intended to be applied to each **input** pair **(k,v)** and for this pair **returns** a, possibly empty, list of pairs



Shuffle and sort

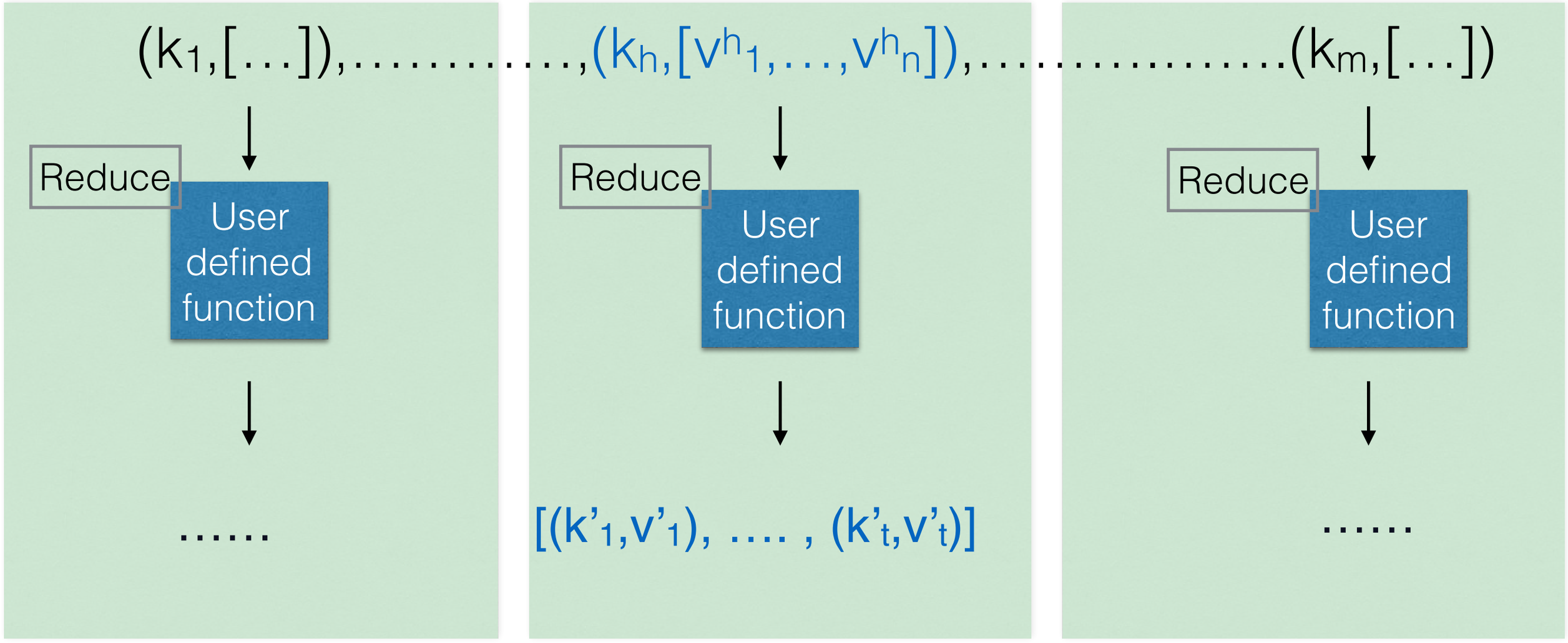
- The **shuffle and sort** phase **groups** Map outputs on the k component producing paris of the form **(k’, [v1’, …. , vn’])**
- For instance:

Shuffle and sort



Reduce phase

- applied to each pair **(k, [v_1, …. , v_n])** produced by S&S for which returns a list of key-value pairs that takes part of the final result



Example: WordCount

- Problem: counting the number of occurrences for each word in a big collection of documents
- Input: a directory containing all the documents
- Pair preparation done by Hadoop: starting from documents, pairs (k,v) where k is unspecified and v is a text line of a document are prepared and passed to the Map phase.

Example: WordCount

- **Map**: takes as input a couple (k,v) returns a pair (w,1) for each word w in v
- **Shuffle&Sort**: groups all of pairs output by Map and produce pairs of the form (w, [1, …,1])
- **Reduce**: takes as input a pair (w, [1, …,1]), sums all the 1's for w obtaining s, and outputs (w,s)

Pseudo code

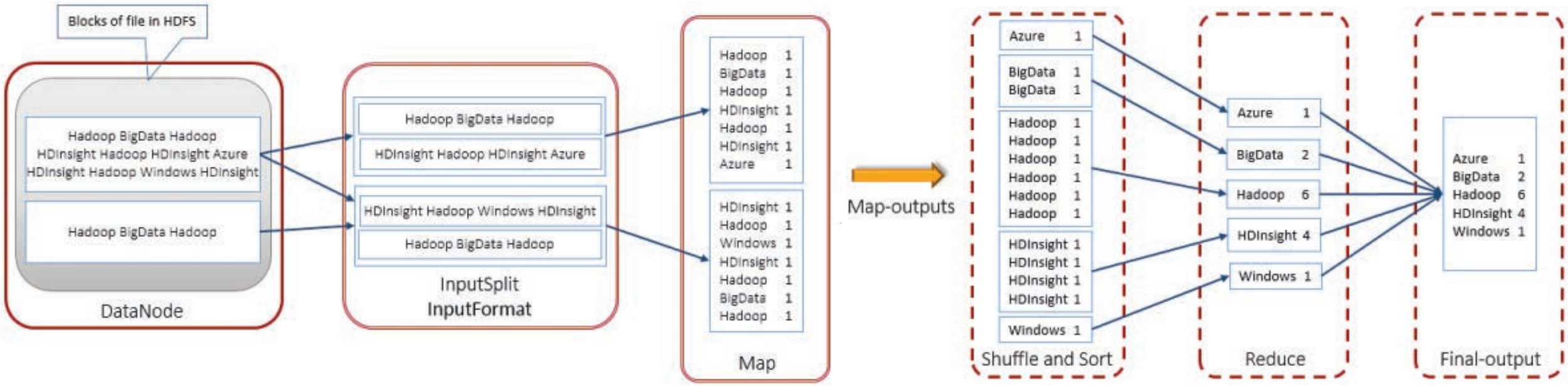
- Map(k, v)

```
for each w in v
    emit(w, 1)
```

- Reduce(k, v)

```
c=0
for x in v
    c = c +1
emit(k, c)
```

Example of data flow



Scalability issues

- Ideal scaling characteristics
  - Twice the data, twice the running time
  - Twice the resources, half the time
- Difficult to achieve in practice
  - Synchronisation requires time
  - Communication kills performance (networks is slow!)
- Thus…minimise inter-node communication
  - Local aggregation can help: reduce size of Map phase output
  - Use of combiners can help in this direction

Combiner

- Goal: pre-aggregate Map output pairs just after the Map task (on the same machine) in order to decrease the number of pairs sent to shuffle&sort (trough the network)
- Its input has the shape of that of Reduce and its output has to be compatible with that of Map (*)
- Like Reduce, it performs aggregation
- Attention: it is up to Hadoop to decide whether a Mapper node runs a Combiner
- So some of the Map nodes run the combiner, some do not; this is why we need (*)

Local aggregation: Combine

- Map(k, v)
  for each w in v
    emit(w, 1)

- Combine(k,v)
  c=0
  for x in v
    c = c +1
  emit(k, c)

- Reduce(k, v)
  c=0
  for x in v
    c = c +**x**
  emit(k, c)

Note that, as usual, Combine is isomorphic to Reduce.
This is because the sum operation performed by WordCount is associative.

Exercise

- Given a collection of (url,time) pairs where url's may repeat, design a MapReduce job to compute the average time for each url
- Define a combiner. Is the Reduce a combiner too?

Hadoop streaming

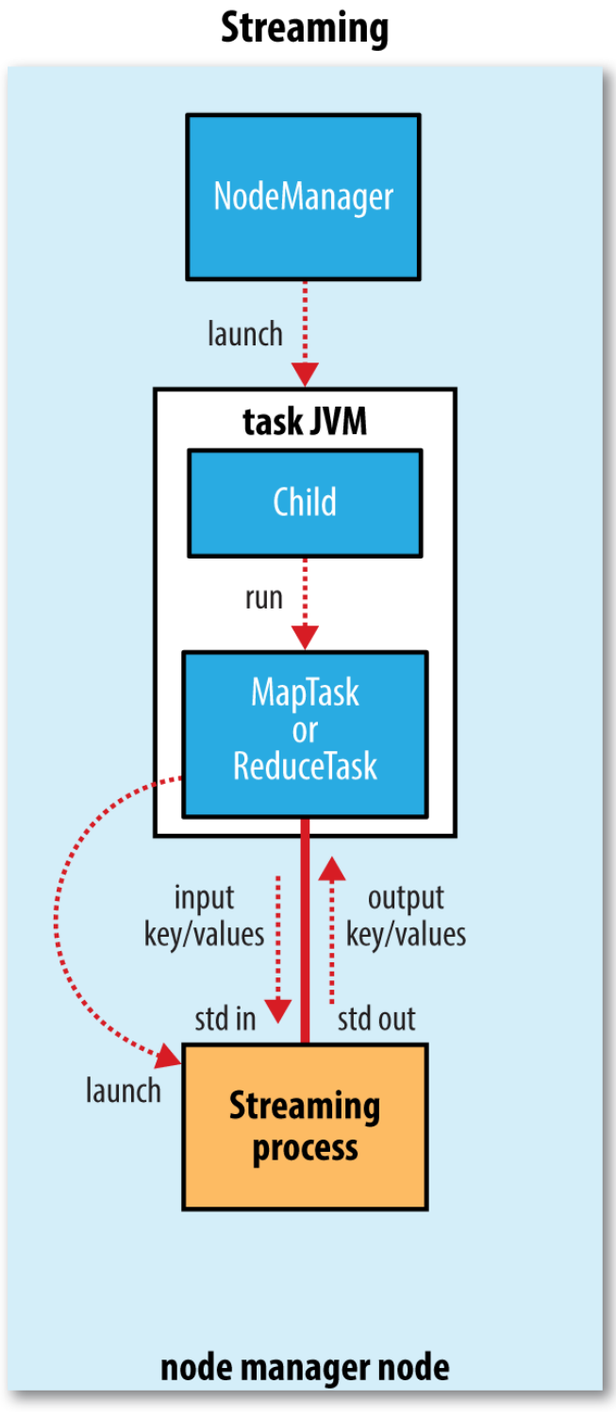
- In a nutshell, the task JVM runs all the auxiliary operations (split and record reading) output writing, etc.

- The Map/Reduce algorithms are executed on the node-manager and can read records (pairs) on the Linux/Unix **standard input stream (stdin)** and write records **(pairs)** on the **standard output stream (stdout).**

- For instance, the Map task running on the JVM reads records and put them on the stdin stream so that the Map program can read and process them.

- The Map program emits pairs by performing simple print operations, that put pairs on the stdout stream, that are in turn read by the Map task and sent to the Shuffle&Sort phase.

- So streaming implies an overhead, that is negligible for complex (time consuming) tasks



Word count in Python

- The mapper

```
#!/usr/bin/env/python

import sys

# input comes from STDIN (standard input)
for line in sys.stdin:
# remove leading and trailing whitespace
line = line.strip()
# split the line into words
words = line.split()
# increase counters
for word in words:
    # write the results to STDOUT (standard output);
    # what we output here will be the input for the
    # Reduce step, i.e. the input for reducer.py
    #
    # tab-delimited; the trivial word count is 1
    print '%s\t%s' % (word, 1)
```

The reducer

- **Important:** if Python is used for MapReduce on Hadoop then
  - pairs (k, [v1,..,vn]) are unfolded to a list of pairs (k, v1),…,(k,vn)
  - In practice each pair (k, vi) is a text line in stdin, where k and vi are separated by the tab character \t
- The Reduce algorithm must identify the groups of lines sharing the same k
- Fortunately lines are ordered on k by shuffle-and-sort.
- Notice that you do not have this unfolding/folding if you use Java for MapReduce

```
#!/usr/bin/env/python

import sys

current_word = None
current_count = 0
word = None

# input comes from STDIN
for line in sys.stdin:
    # remove leading and trailing whitespace
    line = line.strip()

    # parse the input we got from mapper.py
    word = line.split('\t')[0]
    count = line.split('\t')[1]

    # convert count (currently a string) to int
    try:
        count = int(count)
    except ValueError:
        # count was not a number, so silently
        # ignore/discard this line
        continue

    # this IF-switch only works because Hadoop sorts map output
    # by key (here: word) before it is passed to the reducer
    if current_word == word:
        current_count += count
    else:
        if current_word:
            # write result to STDOUT
            print '%s\t%s' % (current_word, current_count)
        current_count = count
        current_word = word

# do not forget to output the last word if needed!
if current_word == word:
    print '%s\t%s' % (current_word, current_count)
```

Scalability issues

- Ideal scaling characteristics
  - Twice the data, twice the running time
  - Twice the resources, half the time
- Difficult to achieve in practice
  - Synchronisation requires time
  - Communication kills performance (network is slow!)
- Thus…minimise inter-node communication
  - Local aggregation can help: reduce size of Map phase output
  - Use of combiners can help in this direction

Combiner

- It is an additional function you are allowed to define and adopt in MapReduce
- Its input has the shape of that of Reduce and its output has to be compatible with that of Map (*)
- Important: a local shuffle-and-sort is performed locally to prepare couples (k, [v1,]
- Like Reduce it performs aggregation
- Differently from Reduce it is run locally, on slaves executing Map
- Goal: pre-aggregate Map output pairs in order to lower number of pairs sent (trough the network) to shuffle-and-sort
- Attention: it is up to Hadoop to decide whether a Mapper node runs a Combiner
    - so some of the Map nodes run the combiner, some do not; this is why we need (*) above
    - we will see in which cases the Combiner is triggered.

Pseudo code with Combine

- Map(k, v)
  for each w in v
    emit(w, 1)

- Combine(k,v)
  c=0
  for x in v
    c = c +1
  emit(k, c)

- Reduce(k, v)
  c=0
  for x in v
    c = c +x
  emit(k, c)

Note that Combine is isomorphic to Reduce.
This is because the sum operation performed by WordCount is *associative*.

We will see cases next where things are more complex.

Spark

Motivation

- Both iterative and interactive queries need one thing that MapReduce lacks

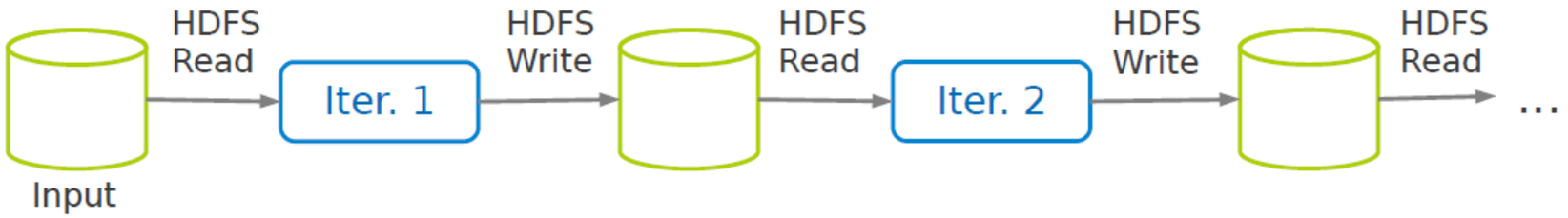
Efficient primitives for data sharing.

- In MapReduce, the only way to share data across processing step is stable storage (disk)
- Replication also makes the system slow, but it is necessary for fault tolerance.

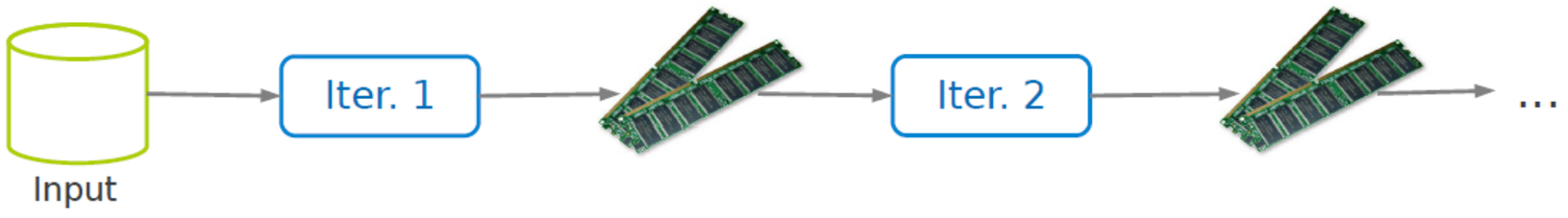
Solution

In-memory data processing and sharing

Hadoop MapReduce

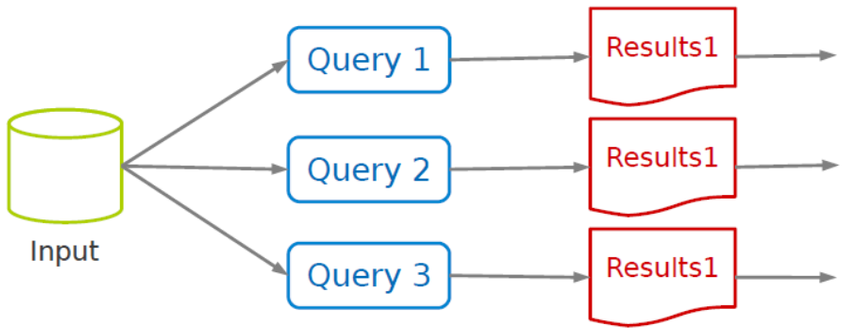


Hadoop Spark

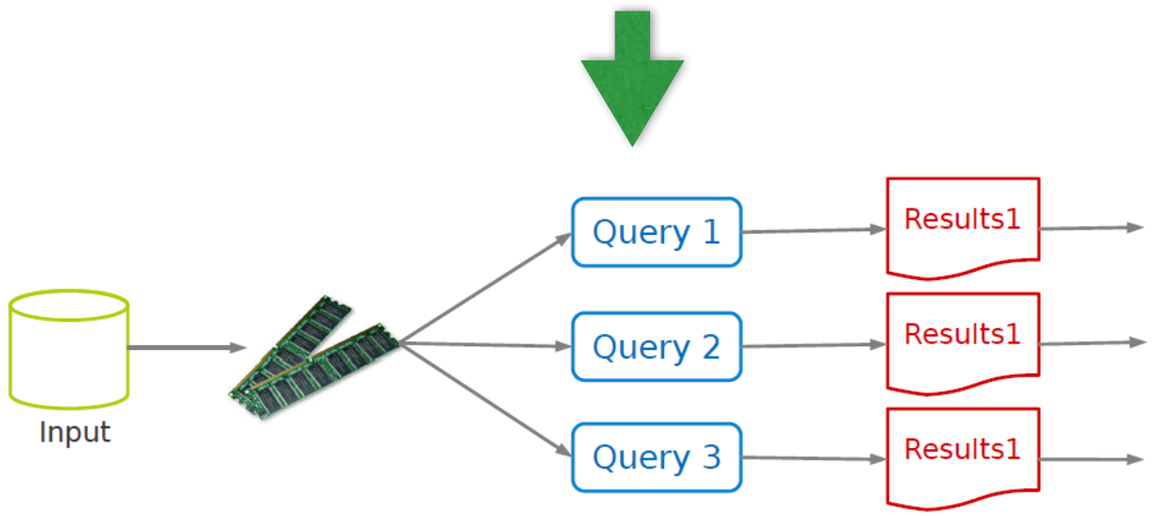


Solution

Hadoop
MapReduce



Hadoop
Spark

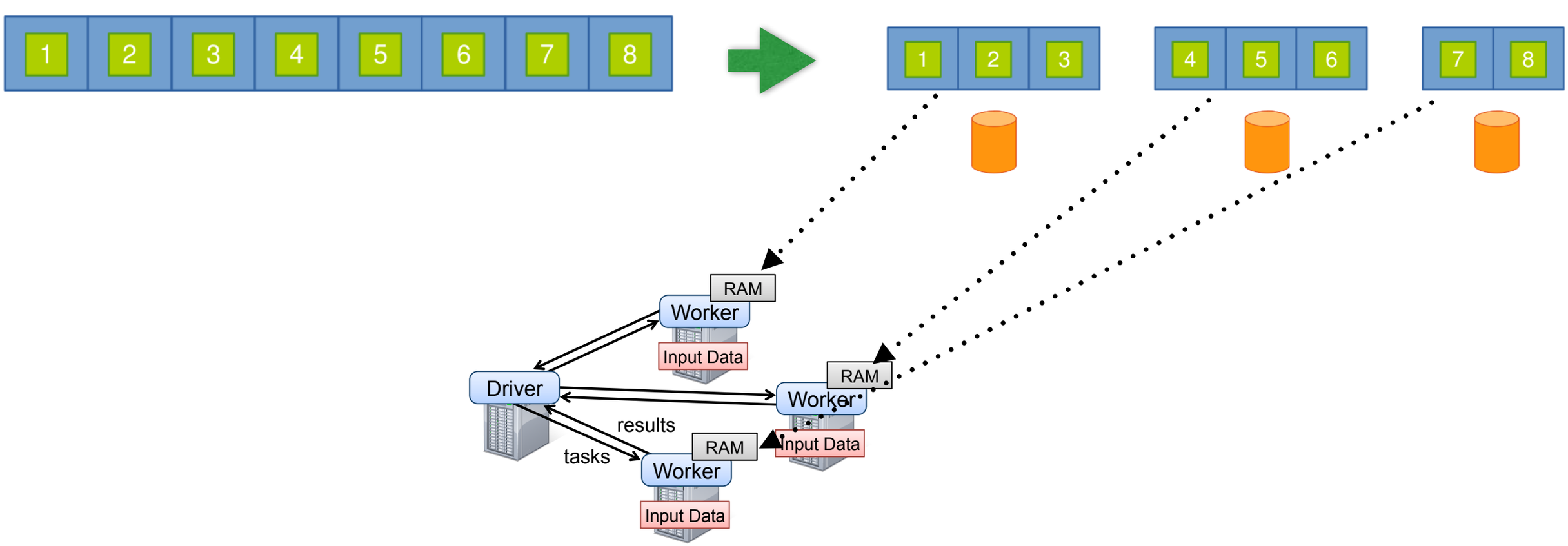


Challenge

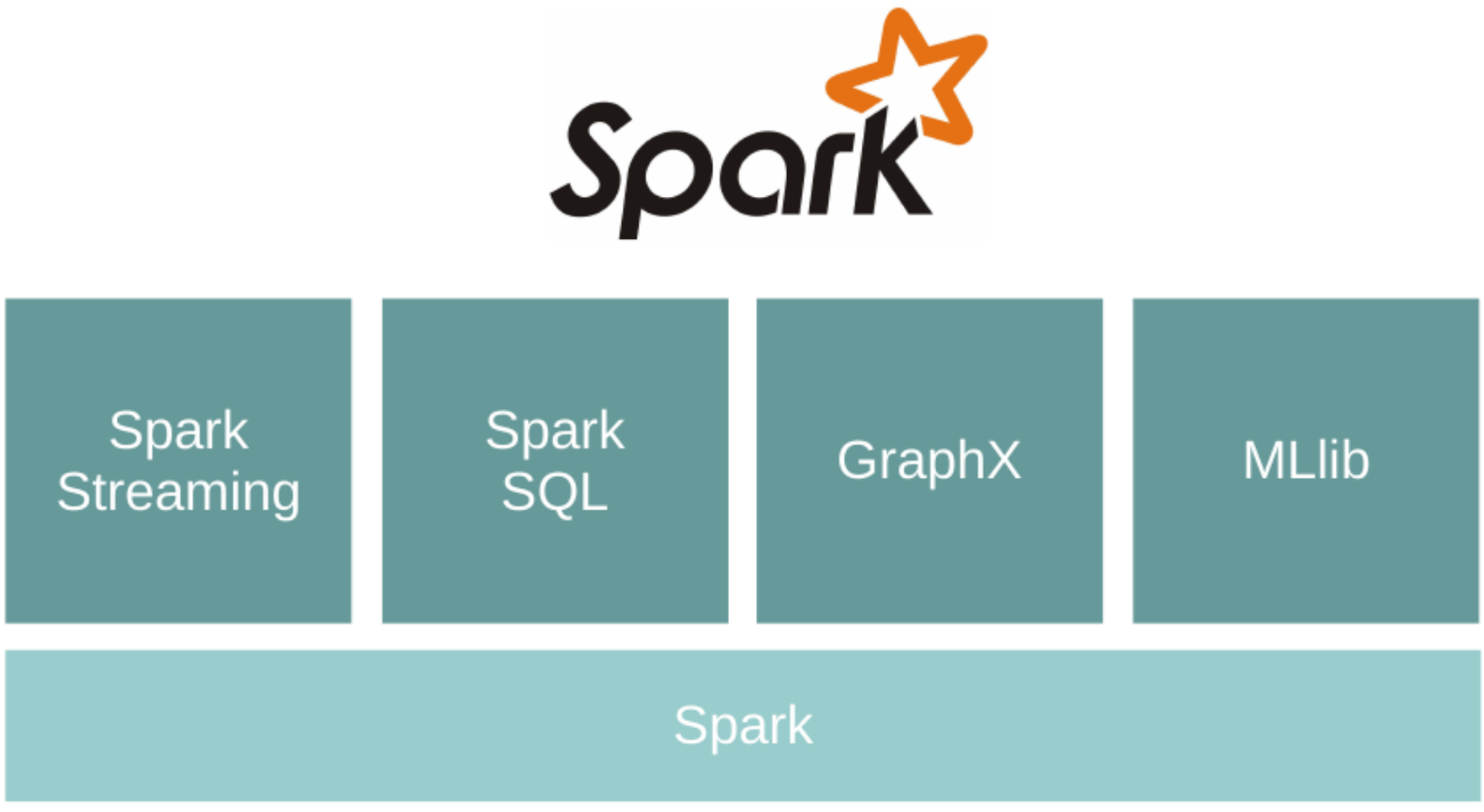
- How to design a distributed memory abstraction that is both fault tolerant and efficient?
- Solution: Resilient Distributed Datasets (RDD)
  - A distributed main-memory abstraction.
  - Immutable collections of objects spread across a cluster.
  - Lineage among RDDs to enable their re-evaluation in case of cluster node failures

Resilient Distributed Dataset

- An RDD is a data collection which is divided into a number of partitions, which can be independently processed.



Spark Processing engine

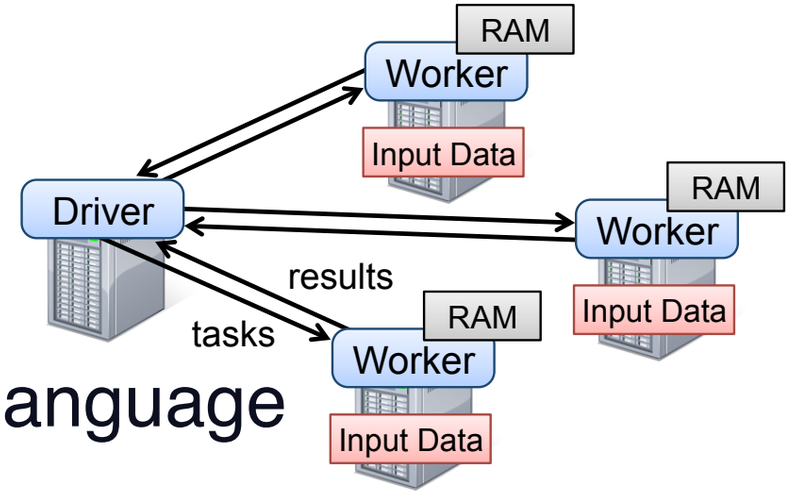


Programming model

- Based on parallelizable operators.
- Parallelizable operators are higher-order functions that execute user-defined functions in parallel, on each partition of an RDD.
- There are two types of RDD operators : transformations and actions.

Programming model

- Transformations : lazy operators that create new RDDs.
- Actions : lunch a computation and return a value to the program driver or write data to the external storage



- Implemented in Scala:
  - a strongly and statically typed functional-OO language
  - compiled and run over the JVM
  - designed at EPFL (Switzerland).
- Java and Python can be used too for Spark programming.

Example

- Suppose that a web service is experiencing errors and a operator wants to search terabytes of logs in the Hadoop filesystem (HDFS) to find the cause.

```
lines = spark.textFile("hdfs://...")
errors = lines.filter(_.startsWith("ERROR"))
errors.persist()
```

- Actions can be used to count errors:

```
errors.count()
```

- Or counting errors mentioning MySQL:

```
// Count errors mentioning MySQL:
errors.filter(_.contains("MySQL")).count()
```

Example

- Suppose that a web service is experiencing errors and a operator wants to search terabytes of logs in the Hadoop filesystem (HDFS) to find the cause.

```
lines = spark.textFile("hdfs://...")
errors = lines.filter(_.startsWith("ERROR"))
errors.persist()
```

- Actions can be used to count errors:

lines is not loaded in memory only **errors** is (simple static analysis)

```
errors.count()
```

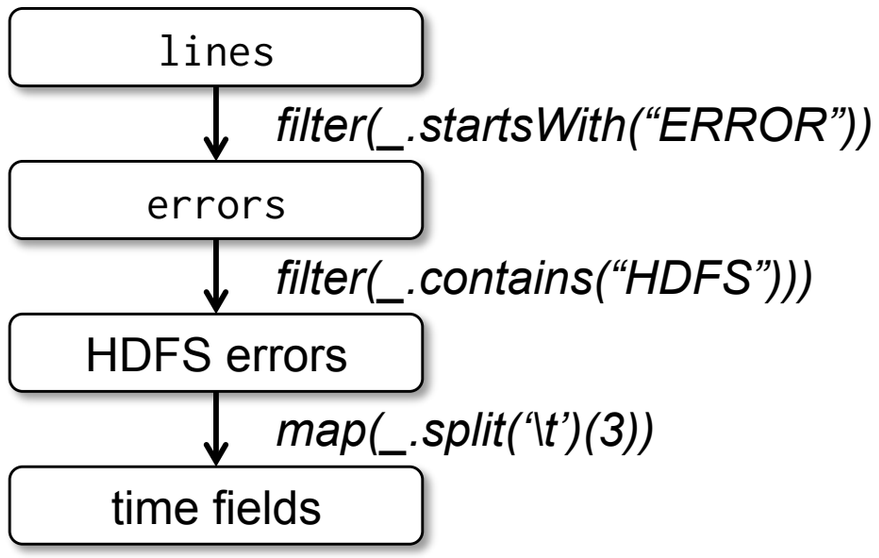
lazy evaluation: **errors** is actually calculated and put in memory when the **count()** action is evaluated

- Or counting errors mentioning MySQ

```
// Count errors mentioning MySQL:
errors.filter(_.contains("MySQL")).count()
```

Fault tolerance via lineage

```
// Count errors mentioning MySQL:
errors.filter(_.contains("MySQL")).count()

// Return the time fields of errors mentioning
// HDFS as an array (assuming time is field
// number 3 in a tab-separated format):
errors.filter(_.contains("HDFS"))
      .map(_.split('\t')(3))
      .collect()
```



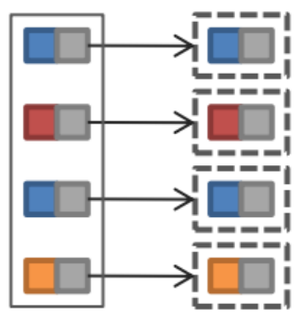
the lineage graph enables RDD re-evaluation in case of failure

RDD transformations and actions

Transformations	$map(f : \mathrm{T} \Rightarrow \mathrm{U})$	:	RDD[T] $\Rightarrow$ RDD[U]
	$filter(f : \mathrm{T} \Rightarrow \mathrm{Bool})$	:	RDD[T] $\Rightarrow$ RDD[T]
	$flatMap(f : \mathrm{T} \Rightarrow \mathrm{Seq[U]})$	:	RDD[T] $\Rightarrow$ RDD[U]
	$sample(fraction : \mathrm{Float})$	:	RDD[T] $\Rightarrow$ RDD[T] (Deterministic sampling)
	$groupByKey()$	:	RDD[(K, V)] $\Rightarrow$ RDD[(K, Seq[V])]
	$reduceByKey(f : (\mathrm{V}, \mathrm{V}) \Rightarrow \mathrm{V})$	:	RDD[(K, V)] $\Rightarrow$ RDD[(K, V)]
	$union()$	:	(RDD[T], RDD[T]) $\Rightarrow$ RDD[T]
	$join()$	:	(RDD[(K, V)], RDD[(K, W)]) $\Rightarrow$ RDD[(K, (V, W))]
	$cogroup()$	:	(RDD[(K, V)], RDD[(K, W)]) $\Rightarrow$ RDD[(K, (Seq[V], Seq[W]))]
	$crossProduct()$	:	(RDD[T], RDD[U]) $\Rightarrow$ RDD[(T, U)]
	$mapValues(f : \mathrm{V} \Rightarrow \mathrm{W})$	:	RDD[(K, V)] $\Rightarrow$ RDD[(K, W)] (Preserves partitioning)
	$sort(c : \mathrm{Comparator[K]})$	:	RDD[(K, V)] $\Rightarrow$ RDD[(K, V)]
	$partitionBy(p : \mathrm{Partitioner[K]})$	:	RDD[(K, V)] $\Rightarrow$ RDD[(K, V)]
Actions	$count()$	:	RDD[T] $\Rightarrow$ Long
	$collect()$	:	RDD[T] $\Rightarrow$ Seq[T]
	$reduce(f : (\mathrm{T}, \mathrm{T}) \Rightarrow \mathrm{T})$	:	RDD[T] $\Rightarrow$ T
	$lookup(k : \mathrm{K})$	:	RDD[(K, V)] $\Rightarrow$ Seq[V] (On hash/range partitioned RDDs)
	$save(path : \mathrm{String})$	:	Outputs RDD to a storage system, *e.g.,* HDFS

RDD transformations : Map

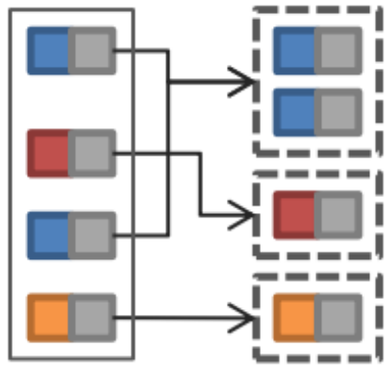
- All pairs are independently processed



```
# passing each RDD element trough a function
nums = sc.parallelize([1,2,3])
squares = nums.map(lambda x: x * x)

# selecting elements making a boolenba function returning true
even = squares.filter(lambda x : x % 2 ==0)

# map + flattening
m = nums.map(lambda x: range(x))
# [[0], [0, 1], [0, 1, 2]]
fm = nums.flatMap(lambda x: range(x))
# [0, 0, 1, 0, 1, 2]
```

RDD transformations : Reduce

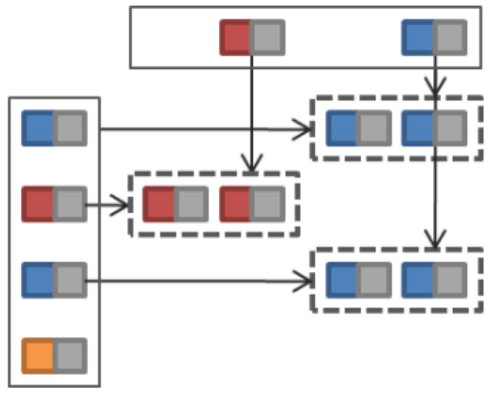
- Pairs with identical key are grouped
- Each group is independently processed



```
pets = sc.parallelize([("cat", 1), ("dog", 1), ("cat", 2),  ("dog", 3) ])

pets.reduceByKey(lambda x, y : x +y)
# [(‘dog', 4), ('cat', 3)]

pets.groupByKey()
pets.groupByKey().map(lambda x : (x[0], list(x[1])))
# [('dog', [1,3]), ('cat', [1, 2])]
```

RDD transformations : Join

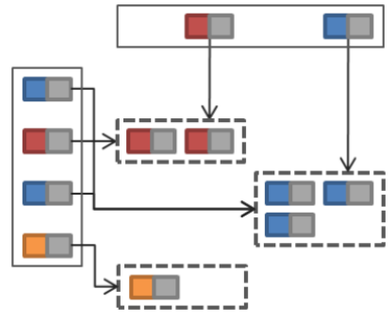
- Equi-join on the key



```
visits = sc.parallelize( [("h", “1.2.3.4"), ("a", "3.4.5.6"), ("h", “1.3.3.1")] )

pageNames = sc.parallelize( [("h", “Home"), ("a", “About")] )

visits.join(pageNames)

# [('a', ('3.4.5.6', 'About')), ('h', ('1.2.3.4', Home')),
    ('h', ('1.3.3.1', 'Home'))]
```

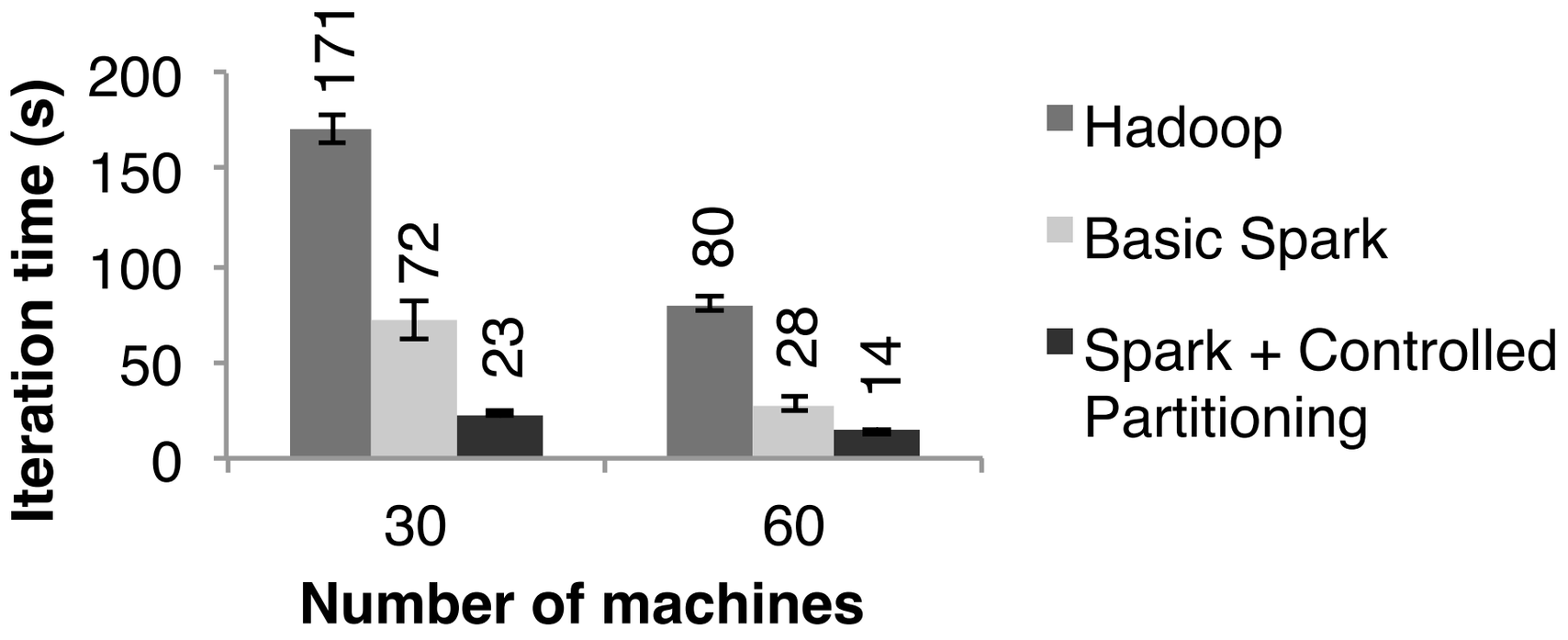
RDD transformations : CoGroup

- Groups each input on key
- Groups with identical keys are processed together



```
visits = sc.parallelize([("h", “1.2.3.4"), ("a", "3.4.5.6"), ("h", "1.3.3.1")] )

pageNames = sc.parallelize([("h", "Home"), ("a", "About"), ("o", "Other")])

visits.cogroup(pageNames)

visits.cogroup(pageNames).map(lambda x :(x[0], ( list(x[1][0]), list(x[1][1]))))

# [('a', (['3.4.5.6'], ['About'])), ('h', (['1.2.3.4', '1.3.3.1'], ['Home'])),
('o', ([], ['Other']))]
```

Some experiments on PageRank



Borrowed from *Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing. Matei Zaharia et al, NSDI 2012.*