Big Data
Systems, Paradigms, Algorithms

Dario Colazzo

Professeur à l'Université Paris Dauphine
Responsable du pole Data Science

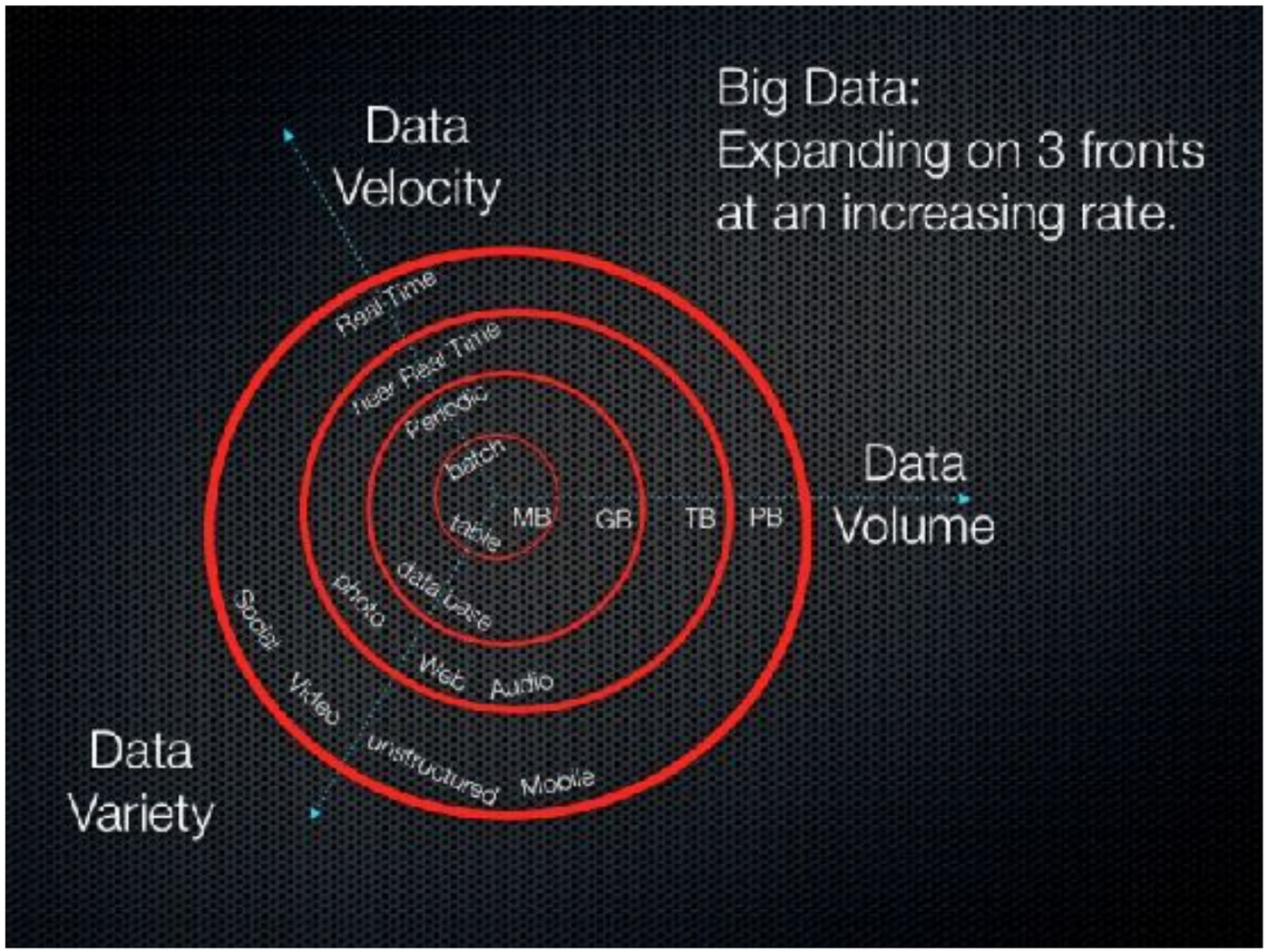
Professeur chargé de cours à l'École Polytechnique

Course outline

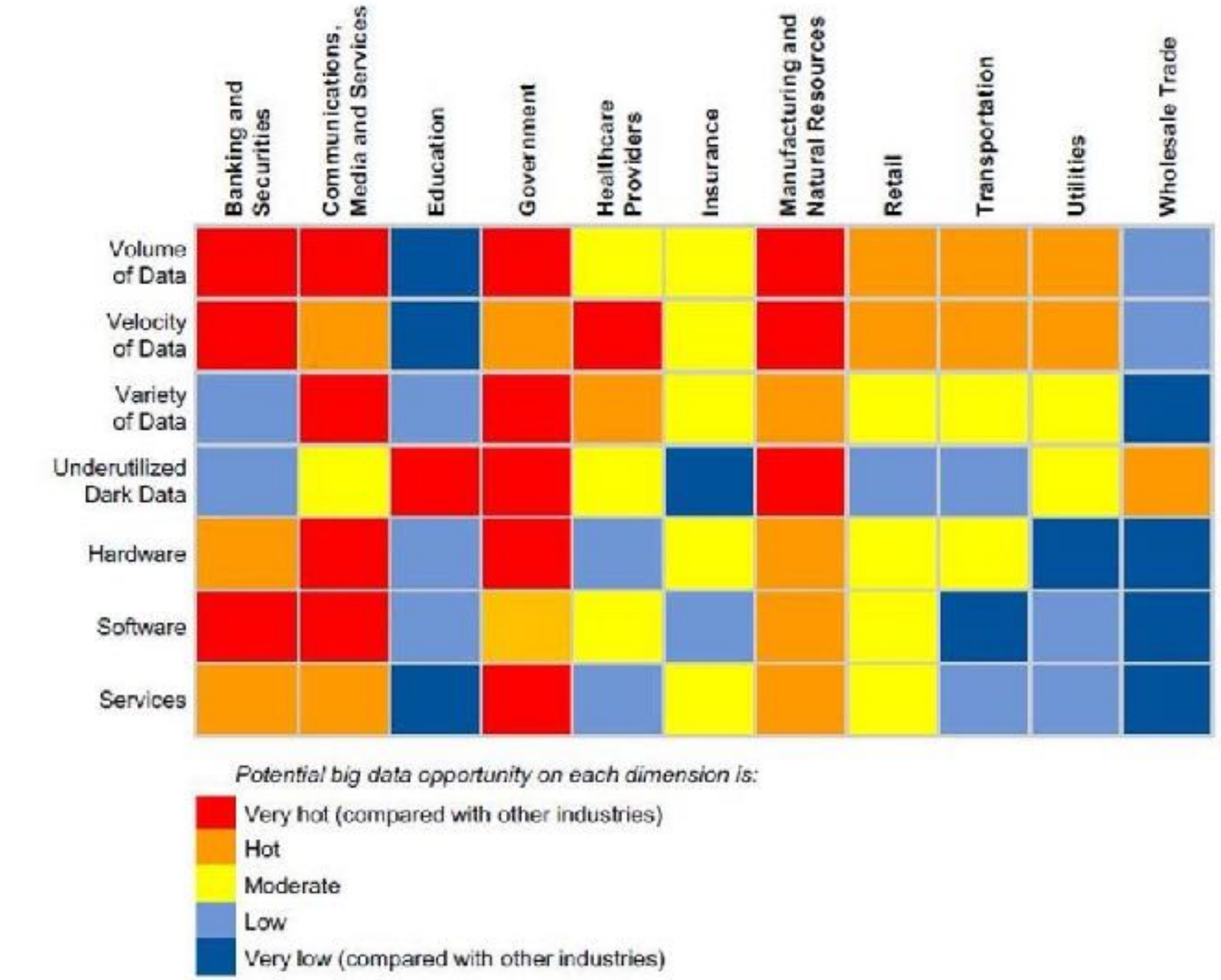
- Introduction Big Data
- Hadoop and MapReduce
    - use case: file manipulation on HDFS, MapReduce algorithms and Python-based programming for SQL queries
    - development both local and on the cluster@Dauphine
- Hive
    - use case: web analytics
- Spark
    - use cases: large scale matrix manipulation, graph processing, clustering (k-means)

What is Big Data?

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



Potential opportunities



Source: Gartner (July 2012)

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?

New computing frameworks and paradigms

- Traditional RDMS can not cope with the 3 V's
  - Why?
- Need of:
  - Cluster of *commodity* computers for distribution and parallelisms
  - File systems to distribute huge file storage and manipulation
  - Programming paradigms to speed up SW development
  - Data models to cope with Variability
  - Scalability, flexibility and robustness: both at HW and SW level

The Big Data hype

- There has been tremendous research and development activity on Big Data tools starting from ~2006 (Hadoop-MapReduce)

- Research on Big Data is still very active (Spark, Flink,…) and will be active in the future

- Emergent applications : IA (large scale Machine Learning), Internet of Things, cybersecurity, ….

Still disruptive ?

- Definitely. We are only at the beginning of the story.

- Increase rate of data volume will accelerate in a wide class of crucial tasks of both companies and public administration

- Important recent advances on techniques and algorithms relying on massive data

  - ML (e.g., deep learning)

  - Large scale SQL, Semantic Web and Graph analytics

  - Combined real-time analytics of data streams and warehouse.

Still disruptive

Back in 2011, the McKinsey Global Institute published a report highlighting the transformational potential of big data.[1] Five years later, we remain convinced that this potential has not been overhyped. In fact, we now believe that our 2011 analyses gave only a partial view. The range of applications and opportunities has grown even larger today.

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

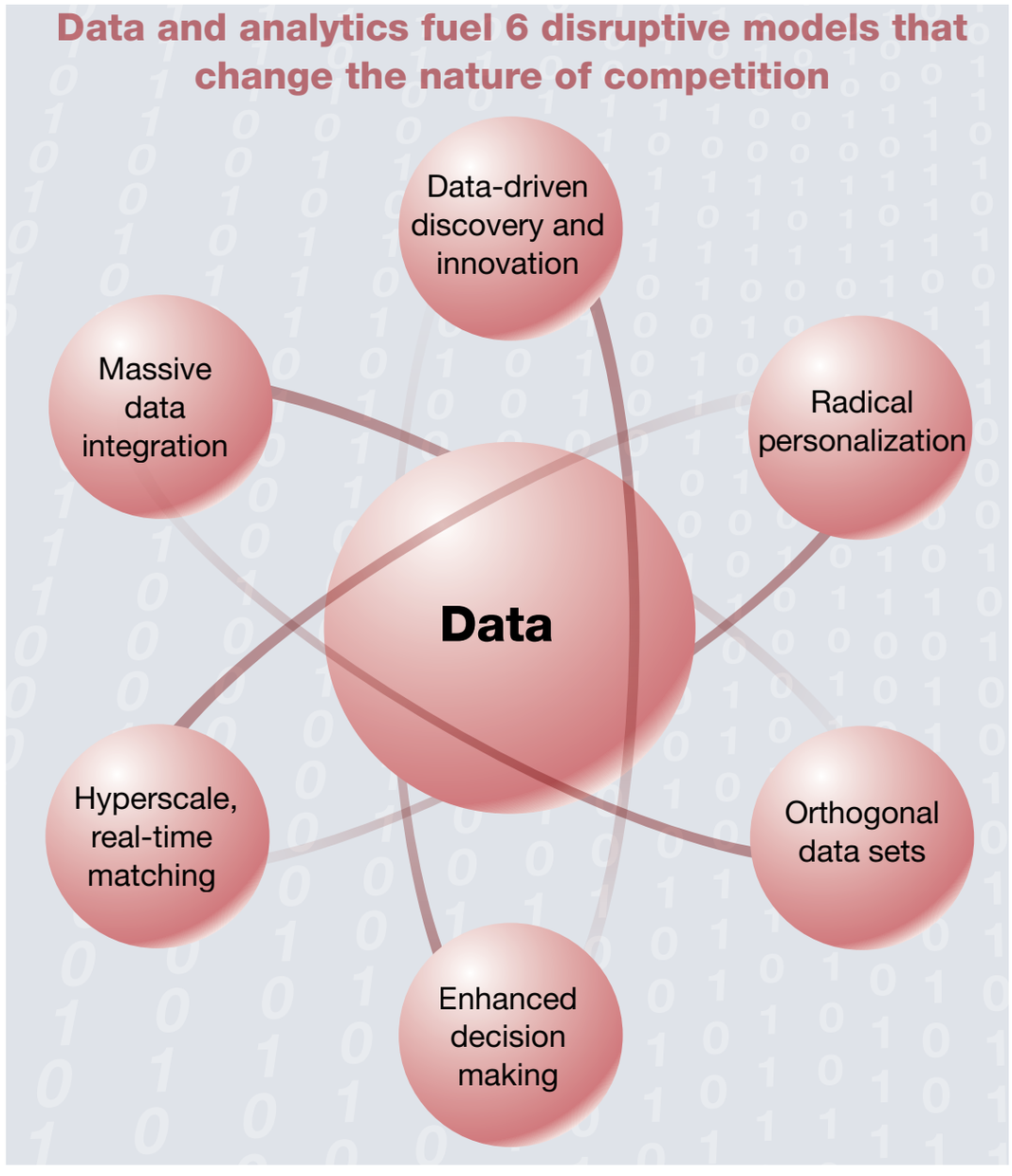
Recently, I talked to **Jeffrey Immelt**, the chief executive of **General Electric**, America's largest industrial company. GE is investing heavily to put data-generating sensors on its jet engines, power turbines, medical equipment and other machines — and to hire software engineers and data scientists.

Immelt said if you go back more than a century to the origins of the company, dating back to **Thomas Edison**'s days, GE's technical foundation has been materials science and physics. Data analytics, he said, will be the third fundamental technology for GE in the future.

I think that's a pretty telling sign of where things are headed.

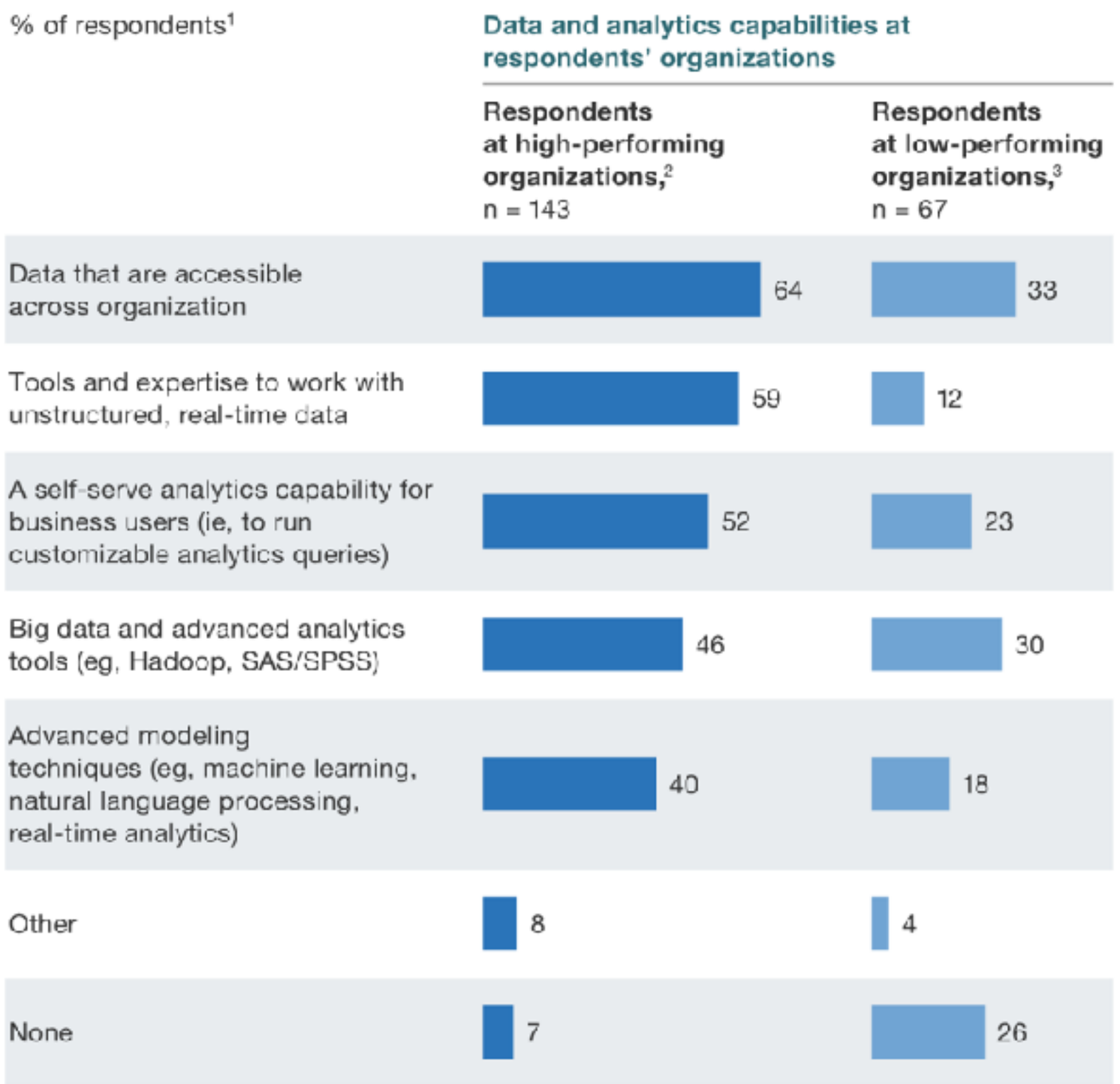
Roberto Zicari Interview with Steve Lohr. December 2016
http://www.odbms.org/blog/

Still disruptive



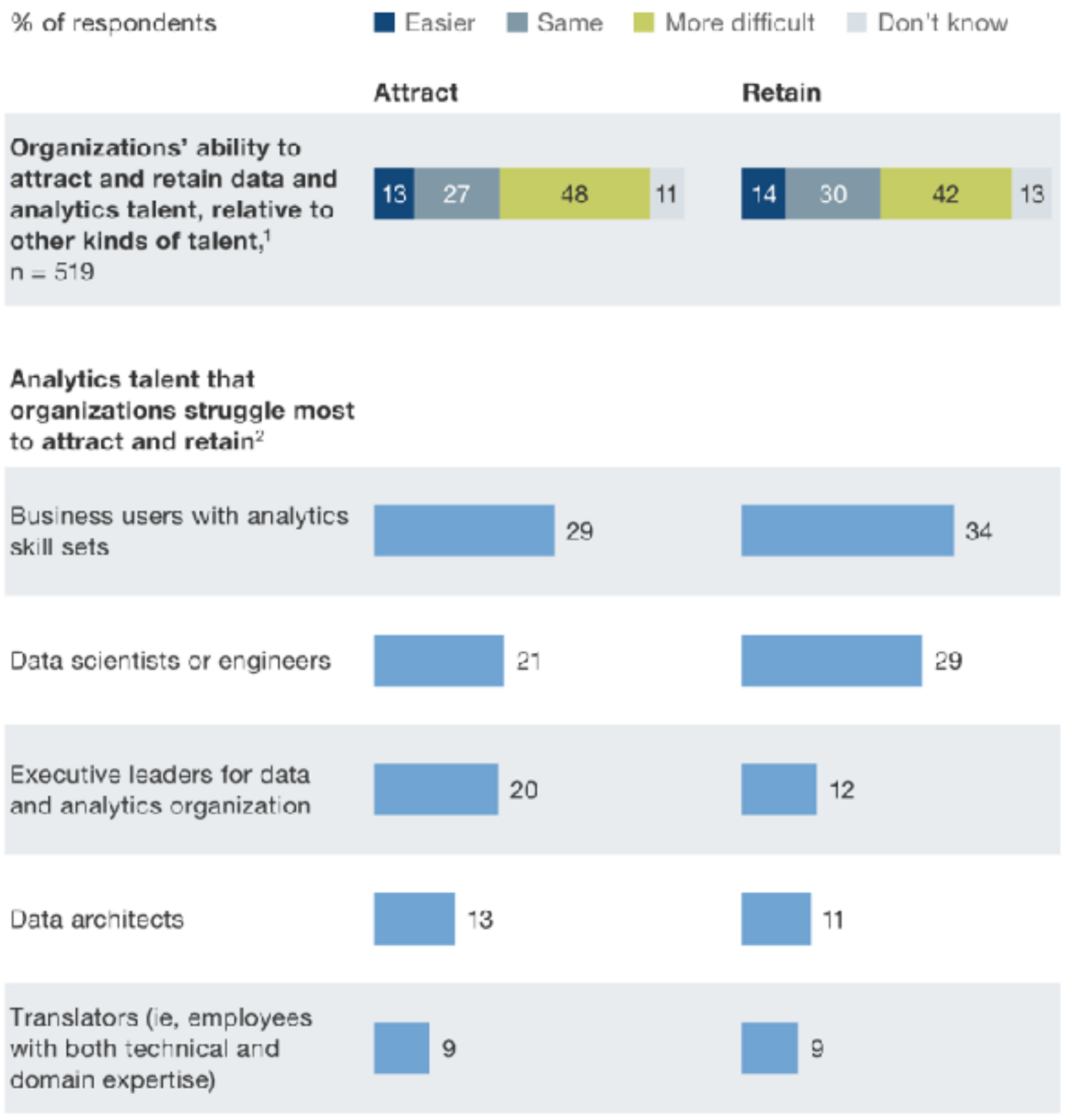
The age of analytics: Competing in a data-driven world.
Report McKinsey Global Institute December 2016

At high-performing organizations, respondents report much more advanced data and analytics capabilities than their peers.



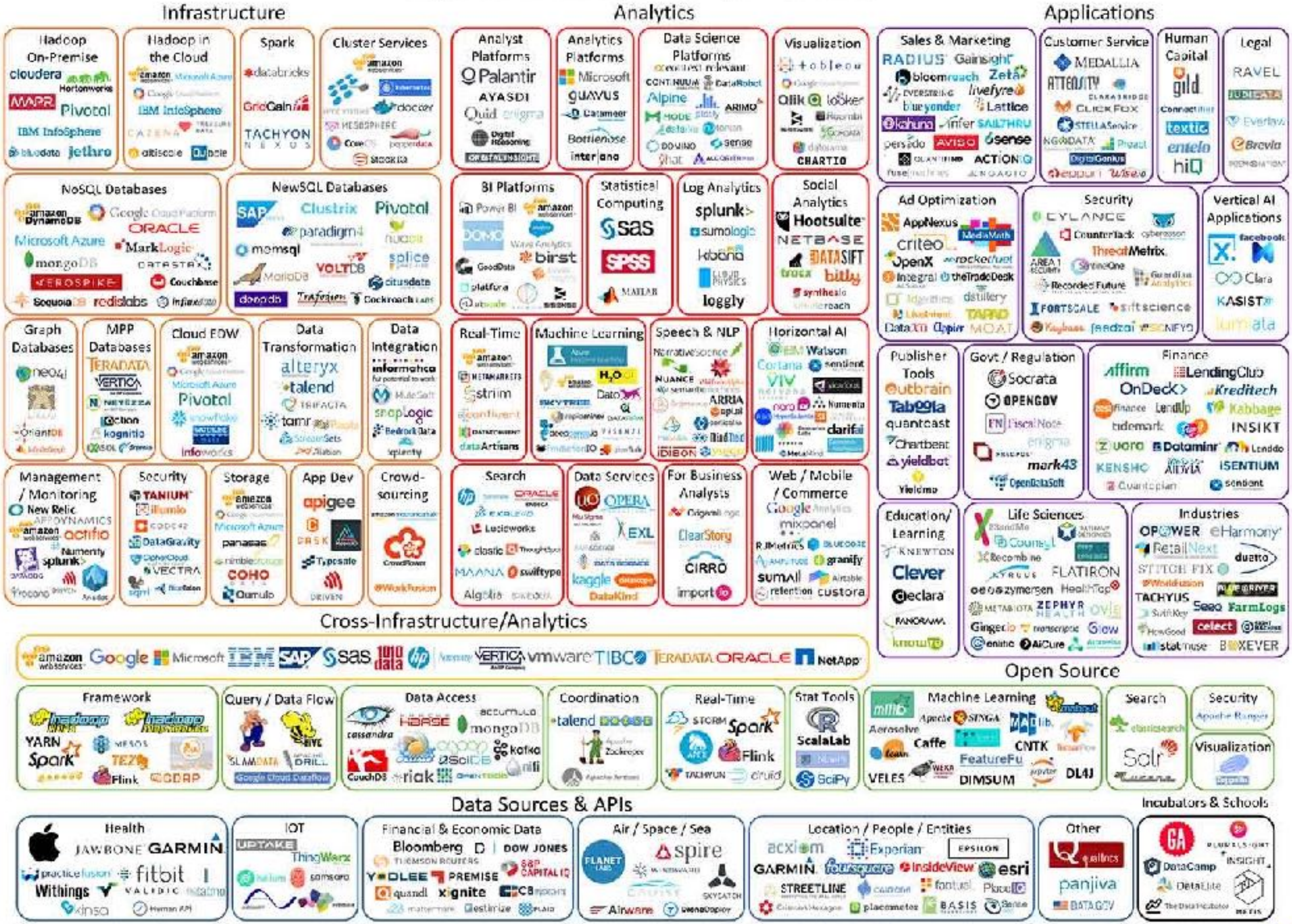
The age of analytics: Competing in a data-driven world.
Report McKinsey Global Institute April 2016

Many companies find it difficult to attract and retain analytics talent—especially business users with analytics-related skills.



The need to lead in data and analytics.
Report McKinsey Global Institute April 2016

Big Data Landscape 2016 (Version 3.0)

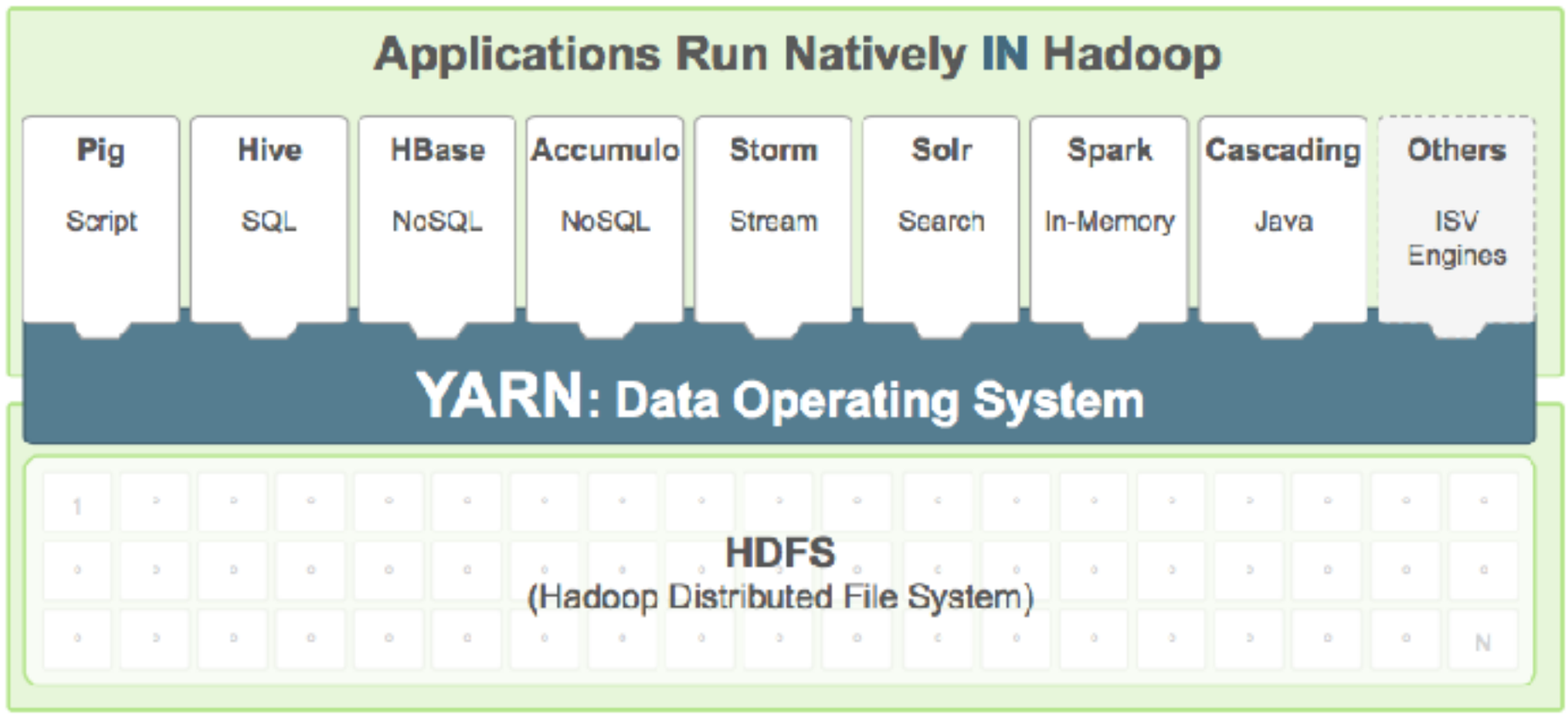


© Matt Turck (@mattturck), Jim Hao (@jimrhao), & FirstMark Capital (@firstmarkcap)

FIRSTMARK

Fortunately we have a winner

Typical Hadoop stack:



Three main players in data and 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 struggles in recruiting and retain talents for each of these 3 tasks

MapReduce and Hadoop timeline

- 2002 Hadoop predecessor Apache Nutch, an open source web search engine, part of the Lucene project, created by Doug Cutting.

- 2003 Publications of the 2 papers describing the architecture of Google's distributed filesystem, called GFS, and MapReduce.

- 2004 Initial versions of what is now Hadoop Distributed Filesystem and MapReduce implemented by Doug Cutting and Mike Cafarella at Yahoo!

- 2006 Apache Hadoop project officially started.

- 2009 Won the minute sort by sorting 500 GB in 59 seconds (on 1,400 nodes) and the 100 terabyte sort in 173 minutes (on a cluster of 3,400 nodes).

- 2009- Intense research activity based on MapReduce, many companies adopting Hadoop.

- 2010 High level, Hadoop-based Apache projects for Hive and Pig and frameworks

- 2013 Development and large adoption of Hadoop 2.x, separating MapReduce from Hadoop core and replace it with YARN. Much more stability and flexibility for applications.

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 file organization of directories and files.

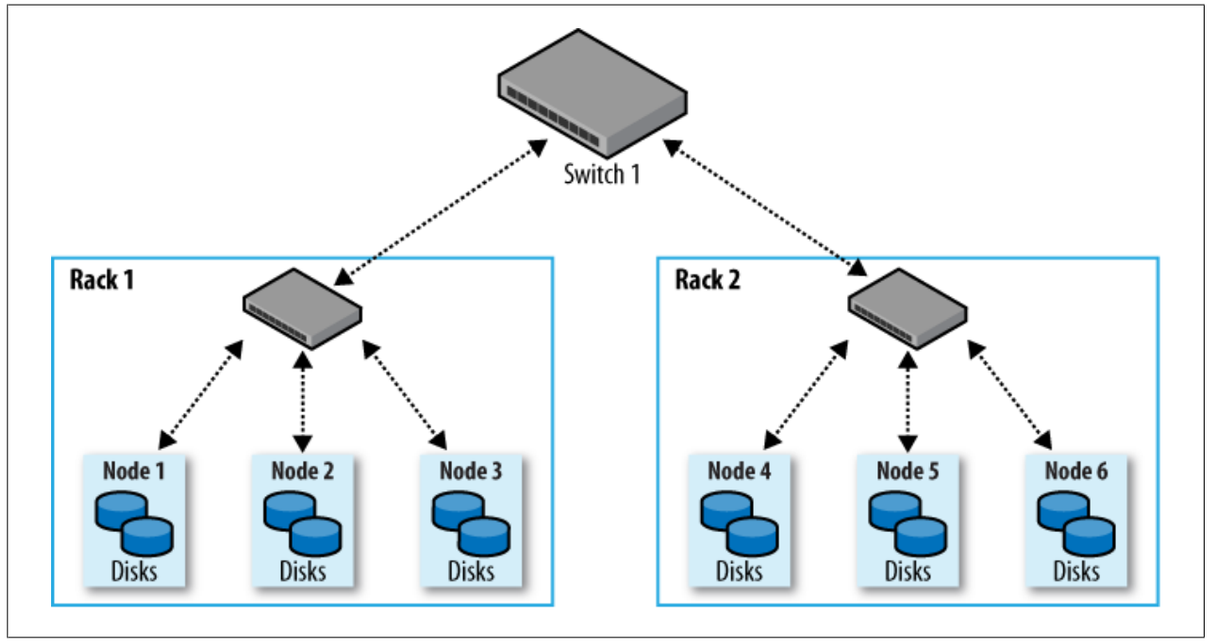
- It has been designed to run on commodity hardware.

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.

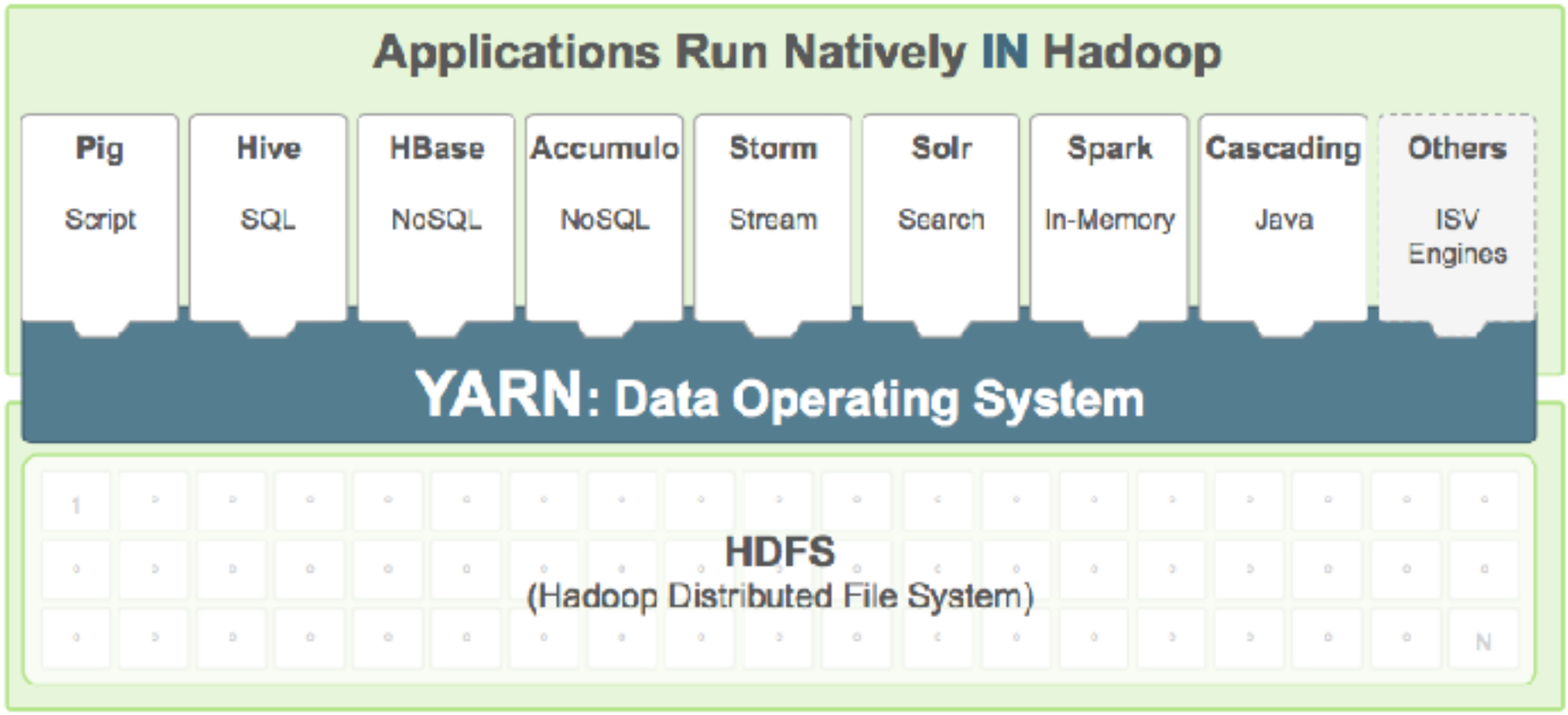
Typical cluster architecture

- 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

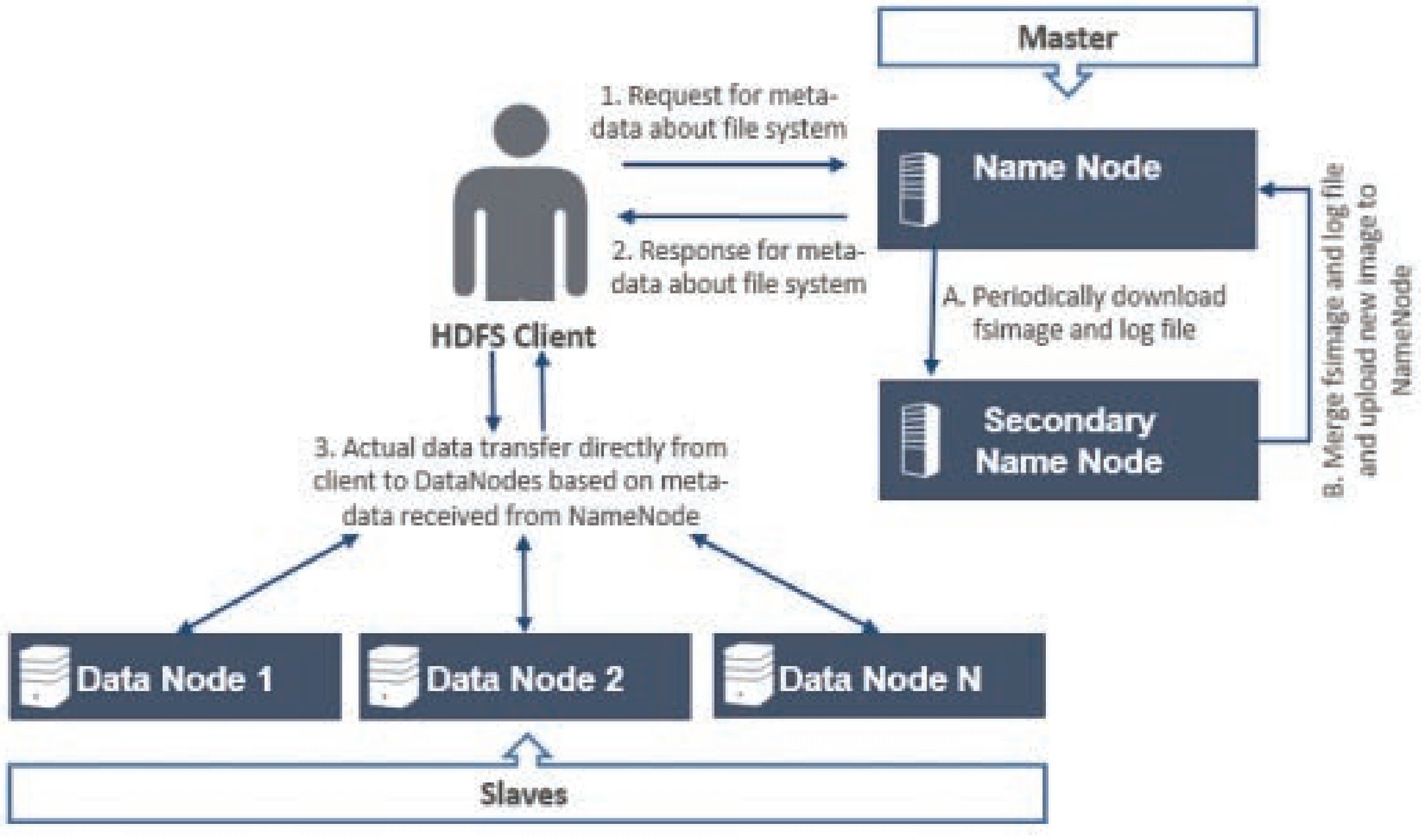
HDFS



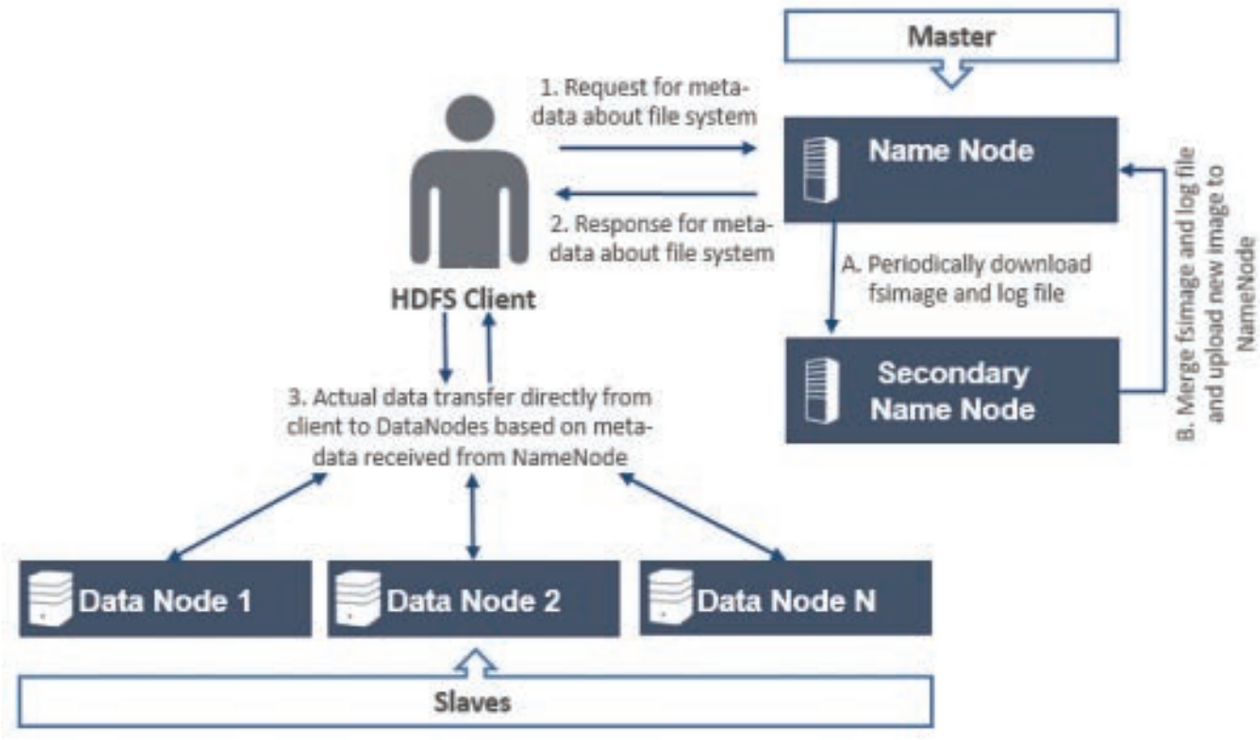
References : *Hadoop: The Definitive Guide* - Tom Wite.

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

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


Master
1. Request for meta-data about file system
2. Response for meta-data about file system
HDFS Client
Name Node
A. Periodically download fsimage and log file
Secondary Name Node
B. Merge fsimage and log file and upload new image to NameNode
3. Actual data transfer directly from client to DataNodes based on meta-data received from NameNode
Data Node 1
Data Node 2
Data Node N
Slaves

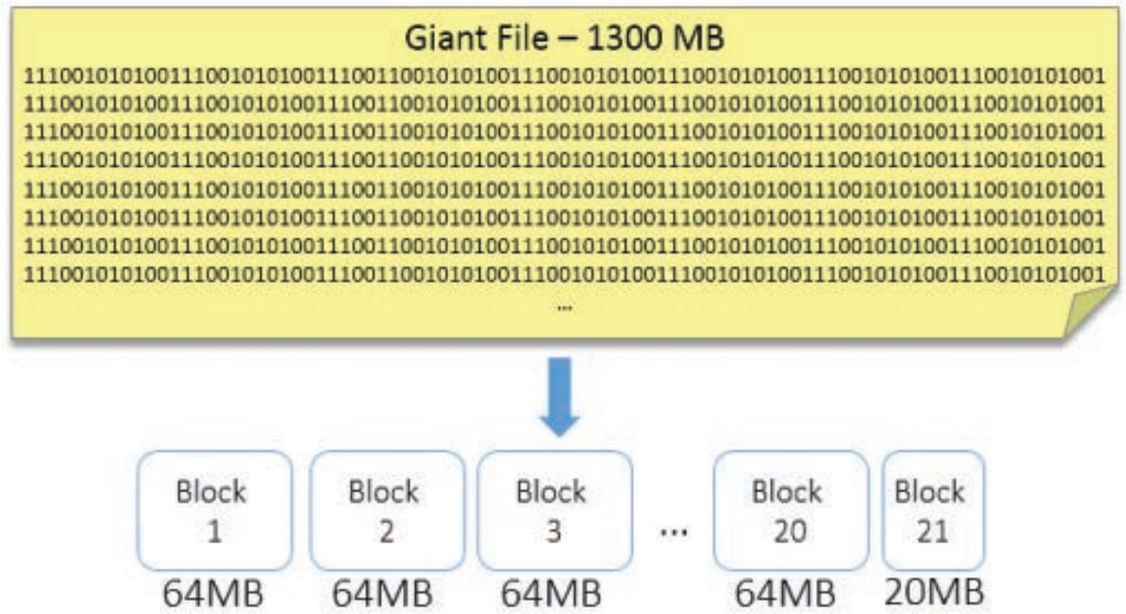
Name nodes and Data nodes



- 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

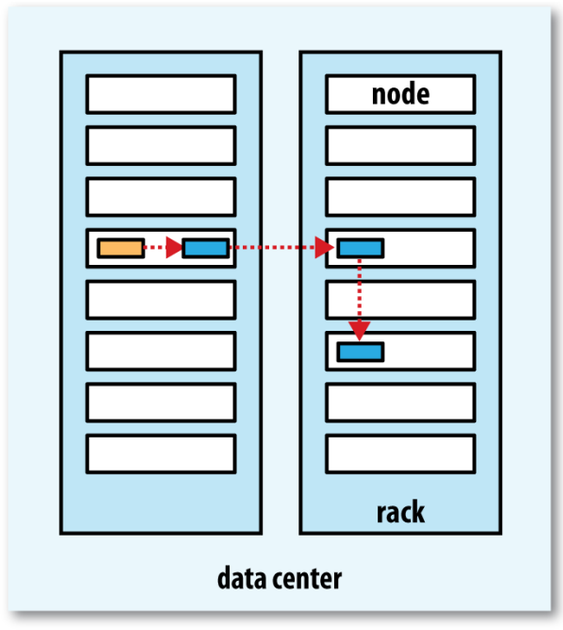
- By default each block is stored 3 times in three different Data nodes, *for fault tolerance*
- When a file is created, an application can specify the number of replicas of each block of the file that HDFS must maintain. The upper bound dfs.replication.max must be respected.
- Settings in hdfs-site.xml

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

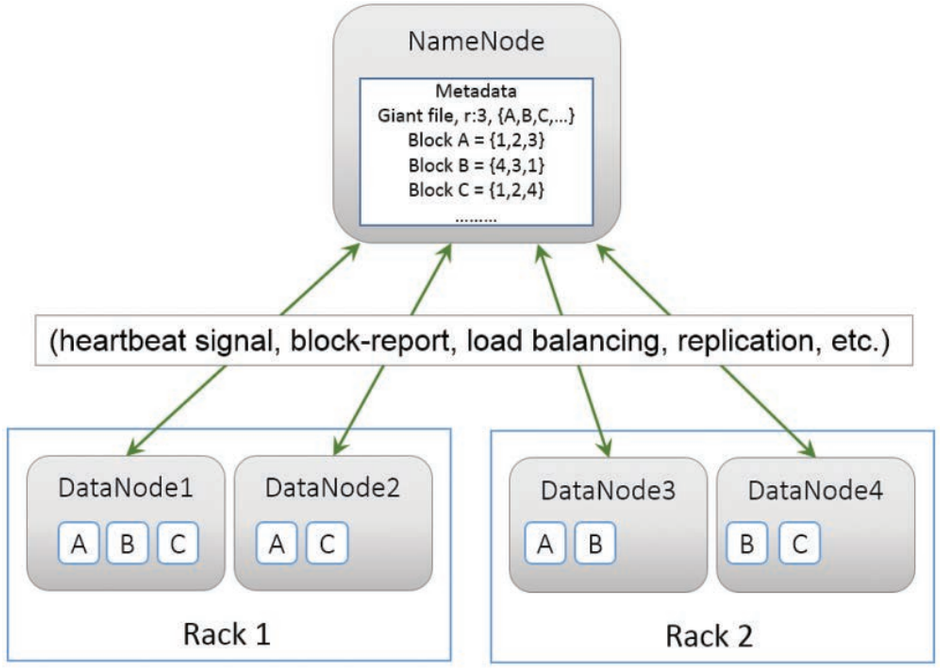
- For robustness, the ideal approach would be to store block replicas in different racks.
- For efficiency, it is better to store all replicas in the same rack.
- Balanced Hadoop approach: store one block on the *client* Data node where the file originates (or a not too busy node chosen by the Name node) 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



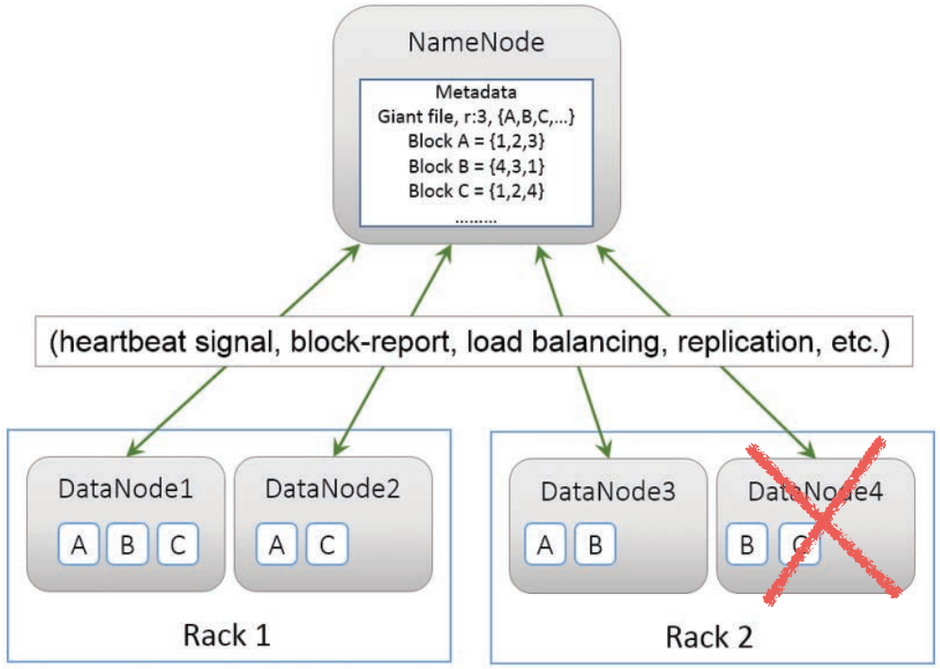
Heartbeats

- 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

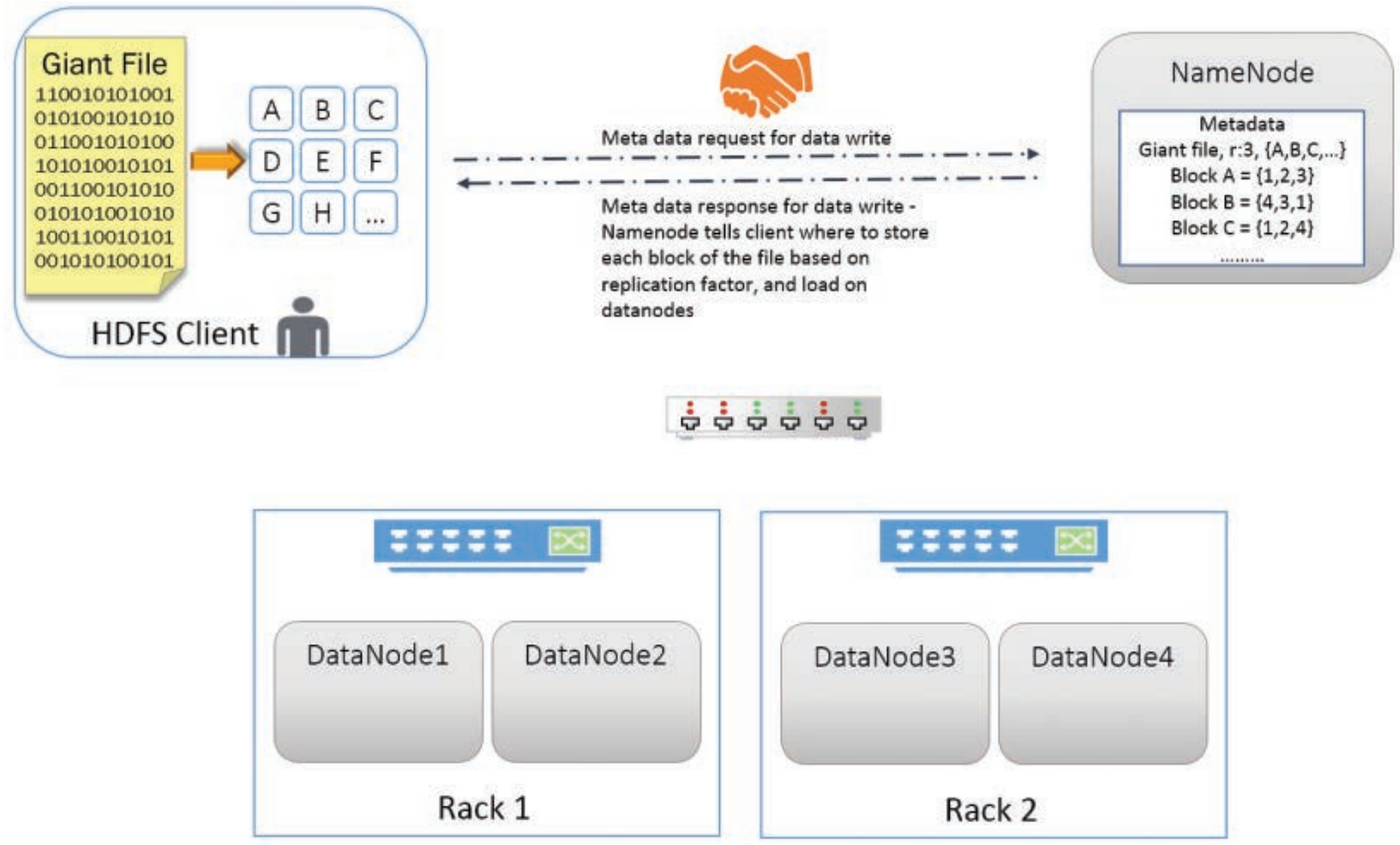
- Assume Data node 4 stops working
- This means that no heart beats is sent anymore
- The Name node then instructs another living Data Node including blocks B and C to *replicate* them on other Data Nodes.
- Data transmission for B and C replication does not involve the Name Node.



Writing a file to HDFS

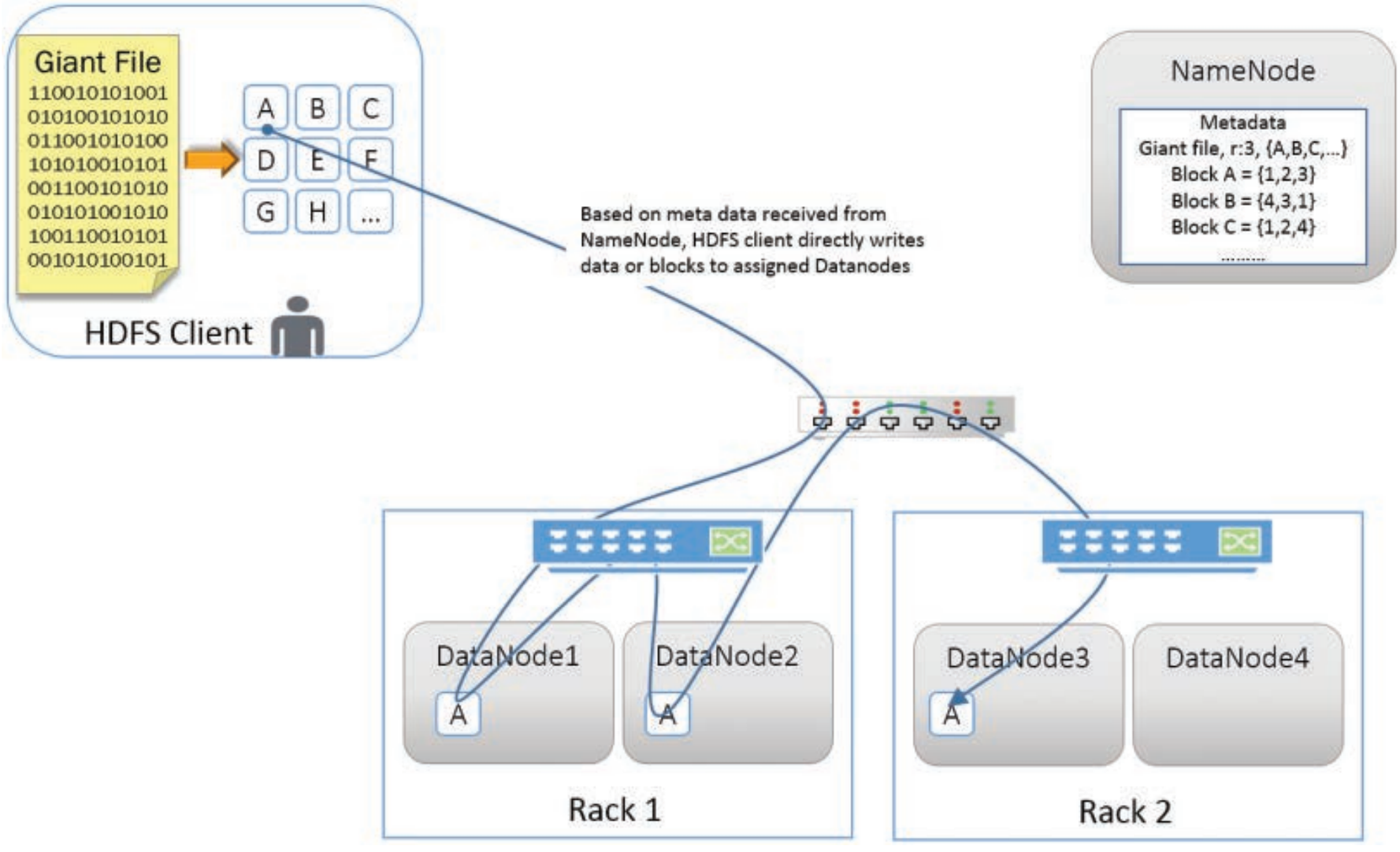
- This can happen, for instance, by command-line or by means of a client program requesting the writing operation.

- First step:



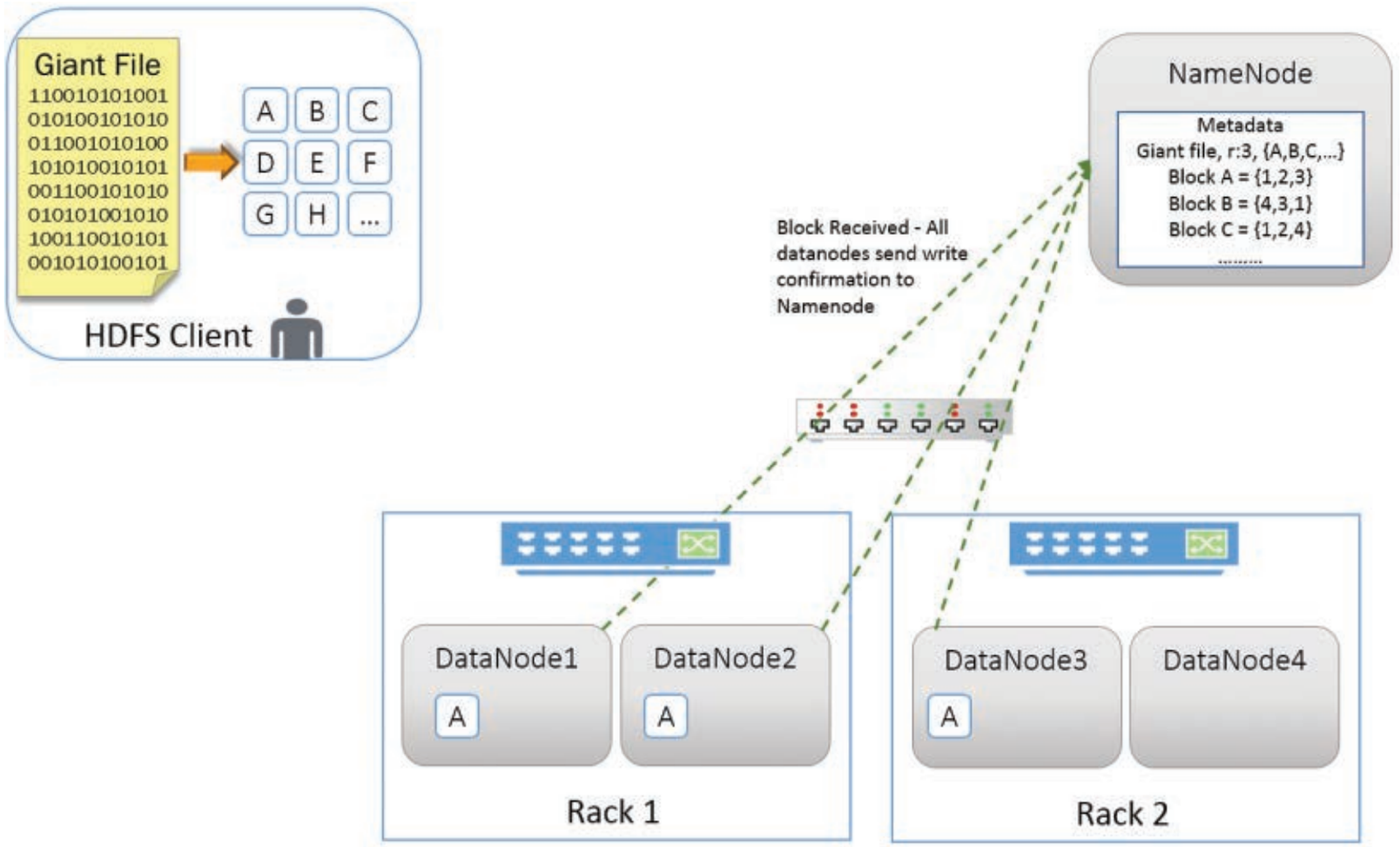
Writing a file to HDFS

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



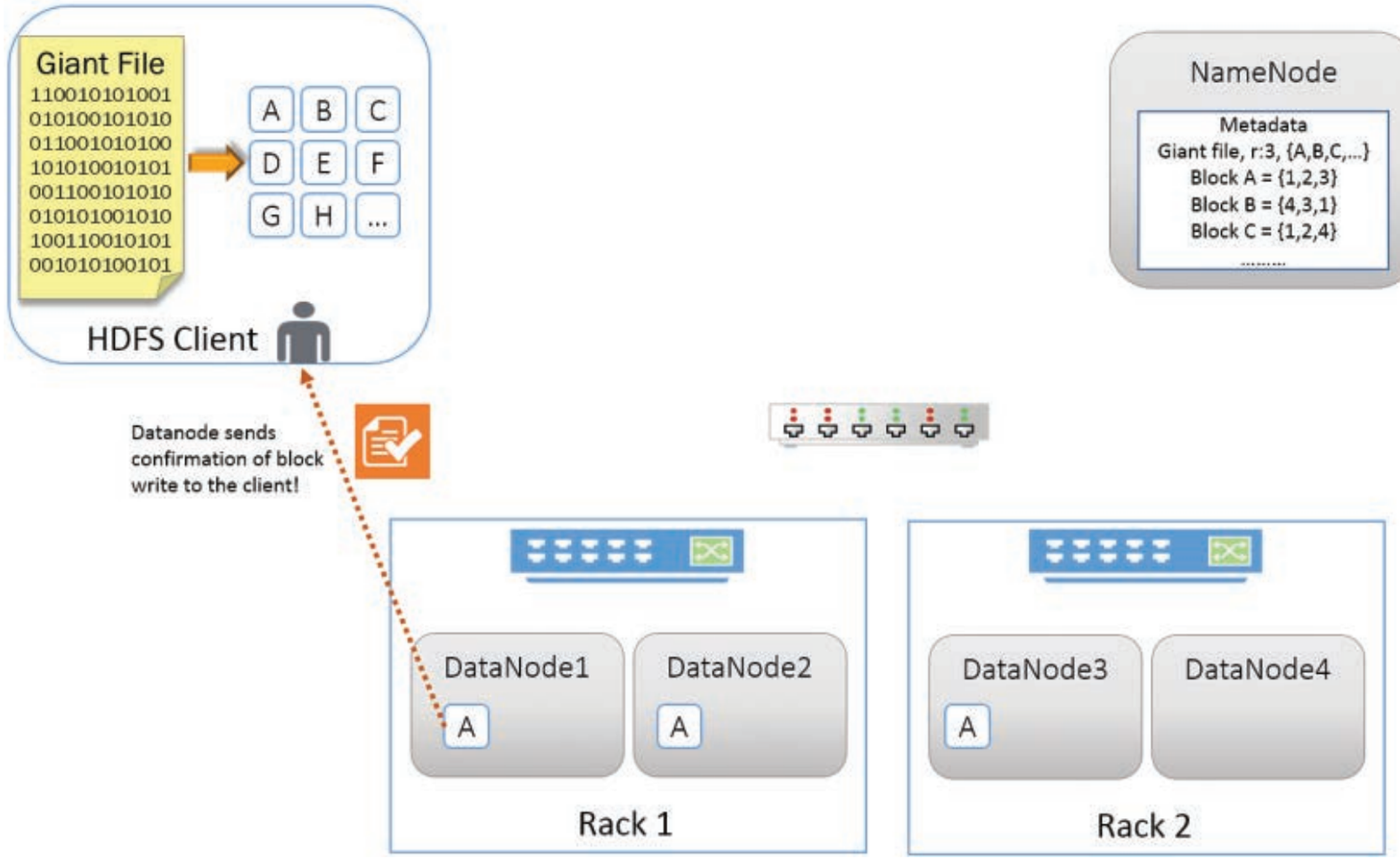
Writing a file to HDFS

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



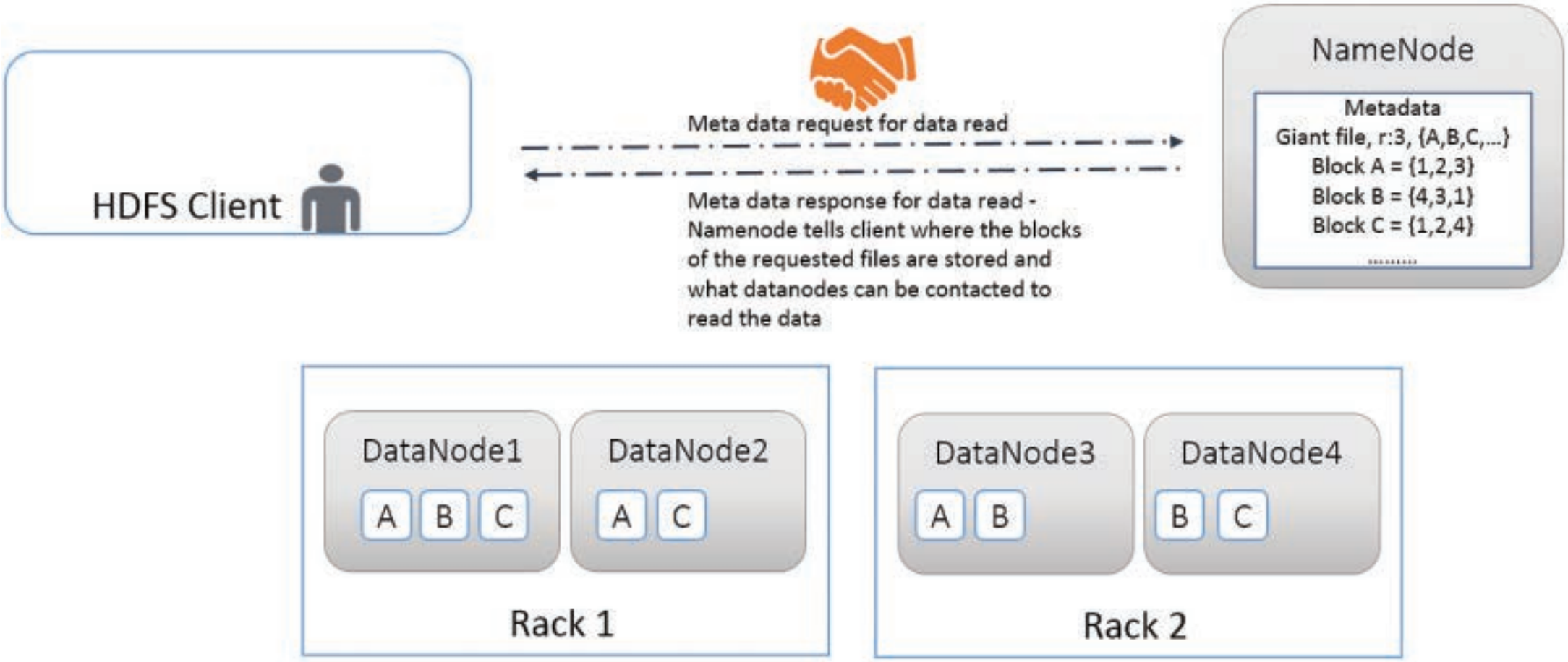
Writing a file to HDFS

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



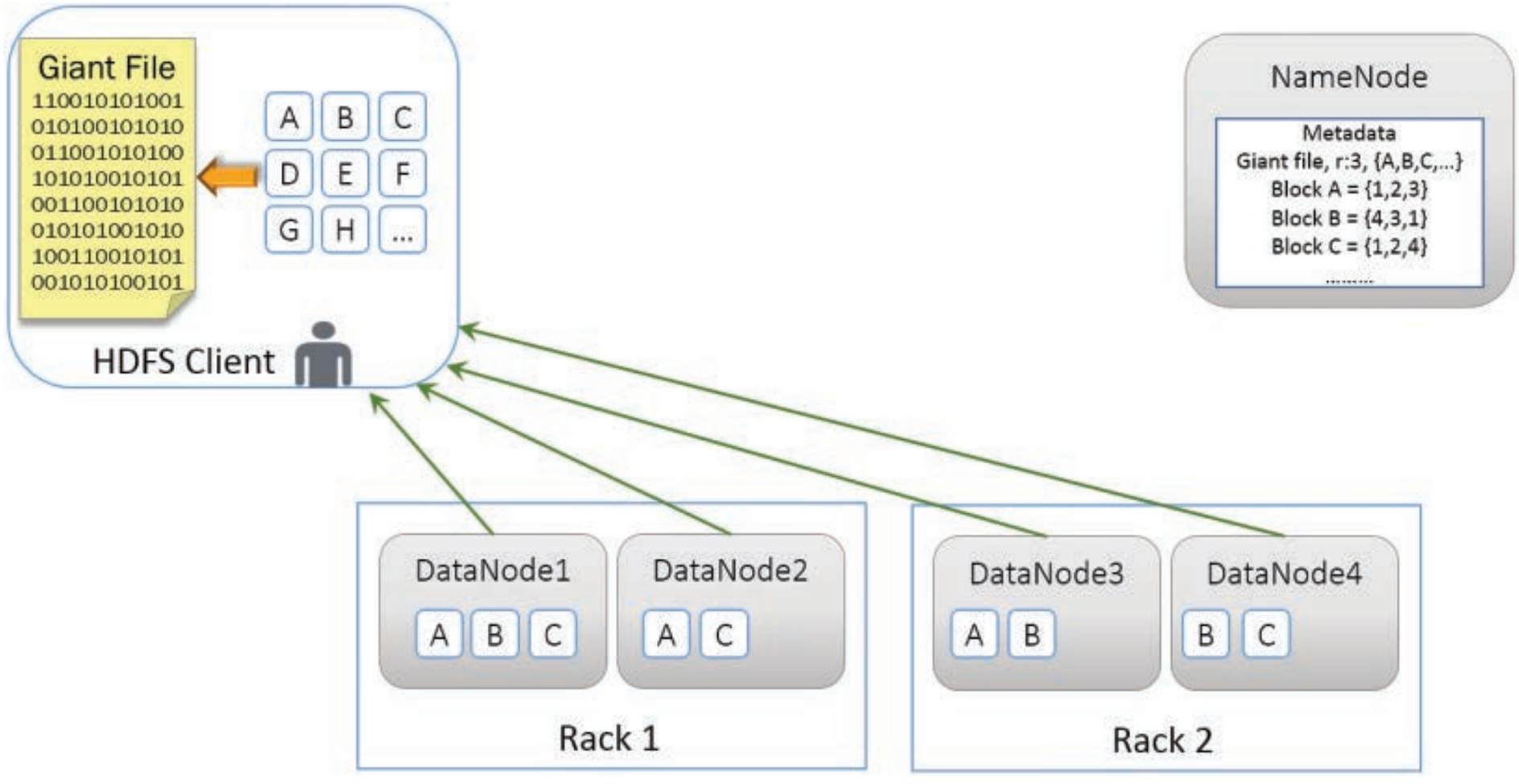
Reading a file from HDFS

- 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 fs -mkdir /example/sampledata
```

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

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

- Listing content of a directory

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

- Copying a file to FS

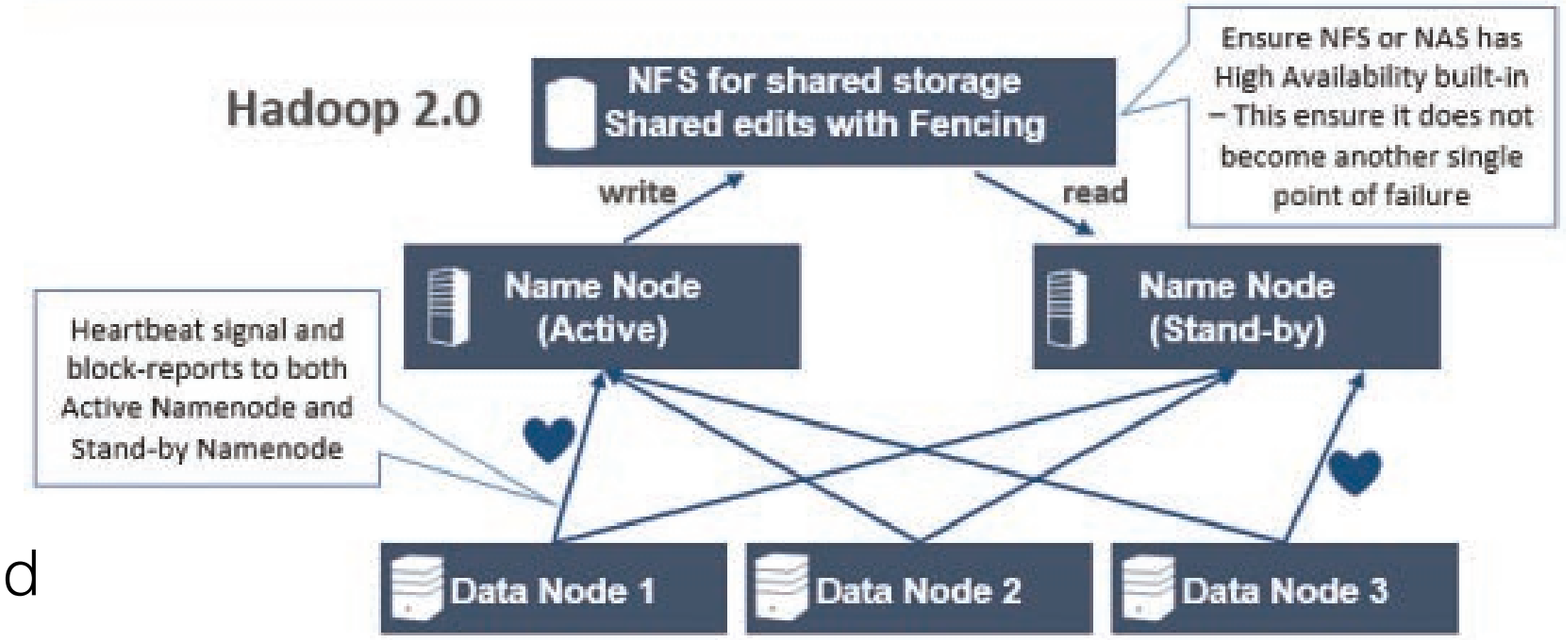
```
> hdfs fs -copyToLocal  \
/user/hadoop/filename \
/apps/dist/examples/data/gutenberg
```

HDFS high availability

- The presence of a single name node in Hadoop 1.0 caused many problems, as files could become corrupted, stopping the cluster working.
- To overcome this limitation, Hadoop 2.x introduced a Secondary name node, with the following tasks
    - storing periodically a checkpoint of Name node information
    - logging changes after the check point
    - taking over in case of failures of the Name node (this requires merging the log and the check point)
- Unfortunately this configuration has not been successful in production, so currently the HDFS High Availability (HA) configuration is normally adopted
    - An active Name Node plus a passive (stand-by) Name Node
    - They stay synchronised : if the active Name Node fails the passive Name Node takes over
    - See next possible configurations.

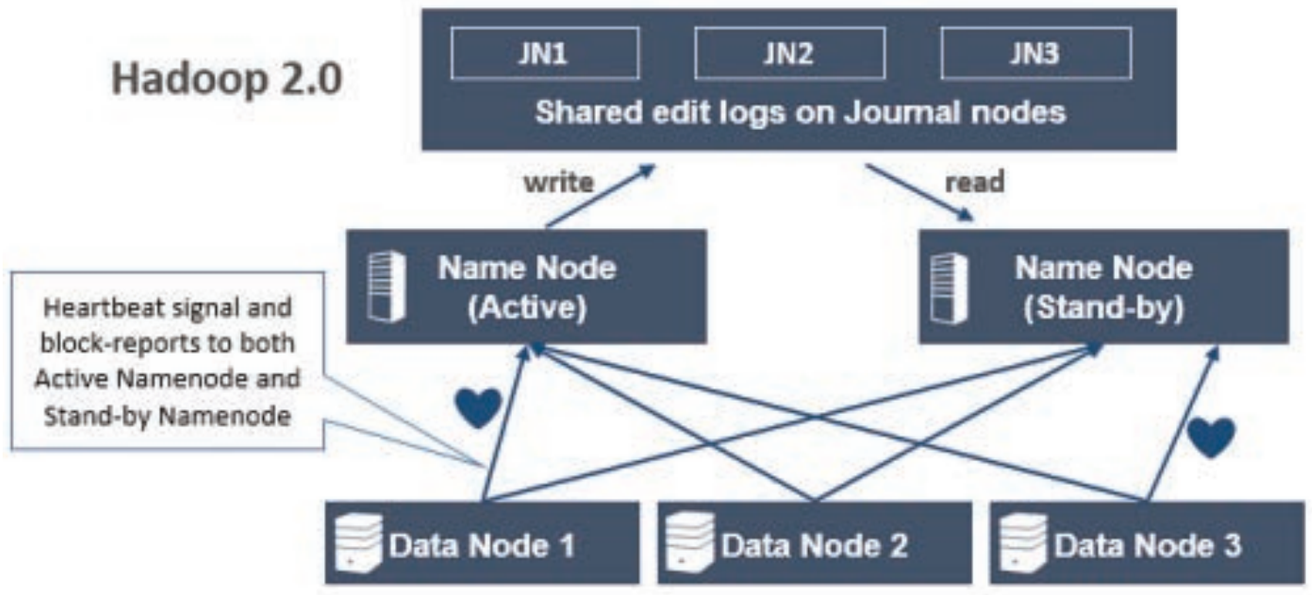
HDFS HA with shared storage

- Separate server for storing the Name node image
- Fencing: active NN writes, passive NN reads
- Data nodes send heartbeats to both name nodes
- Passive NN is constantly updated
- It is important that the server for passive NN is as powerful as that for the active NN
- For this HA option configurations are needed.



HA with quorum based storage

- Frequently adopted.
- It can be deployed on the Hadoop cluster itself
- Multiple daemons called Journal nodes are accessed by both NN, for write and read as before
- Each change to the active NN file system is logged to the majority of the JNs.
- Only when a quorum of the JNs validate the writing the active NN validate the modification to the file system.
- As JNs performs light operations (record append), they can run on the cluster nodes.
- At least 3 JNs must run, preferably on different machines, in case of HW failures.



MapReduce & YARN

Plan

- Why MapReduce
- Basic and advanced MapReduce
- YARN - Yet Another Resource Negotiator

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 data analytics
- Several programming languages can be used to implement MapReduce algorithms (Java, Python, C++, etc.)
- Its main runtime support is Hadoop, which was *initially* designed around MapReduce
- 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

    - A MapReduce algorithm takes as input a collection of key-value pairs and returns a collection of the same kind

- The Map function is intended to be applied to each input pair (k,v) and for this pair returns a, possibly empty, list of pairs [(k1,v1), …. , (kn,vn)]

- The Reduce function is applied on pairs of the form (k', [v1', …. , vn']) and for each such pair returns a list of key-value pairs that takes part of the final result.

- The behaviour of (how the output is determined for) the Map and Reduce function is determined by the user

Main principles

- Processing model: given a MapReduce algorithm over an input I

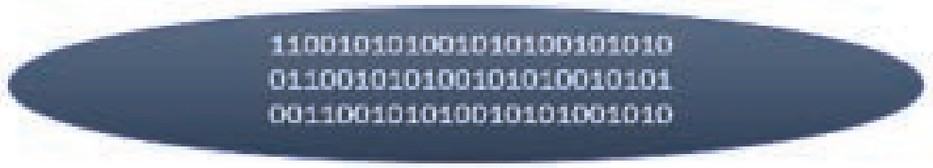
  - the Map phase consist of applying the Map function to each pair in I

  - the collection M of pairs resulted by the Map phase are submitted to a group-by process called *shuffle-and-sort* aiming at grouping by key the pairs in M, resulting into a collection of pairs of the form (k', [v1', …. , vn'])

  - the Reduce phase consists of iterating the Reduce function application on each pair produced by shuffle-and-sort

Map

Take a large problem and divide it into sub-problems in the form of Key-value pairs

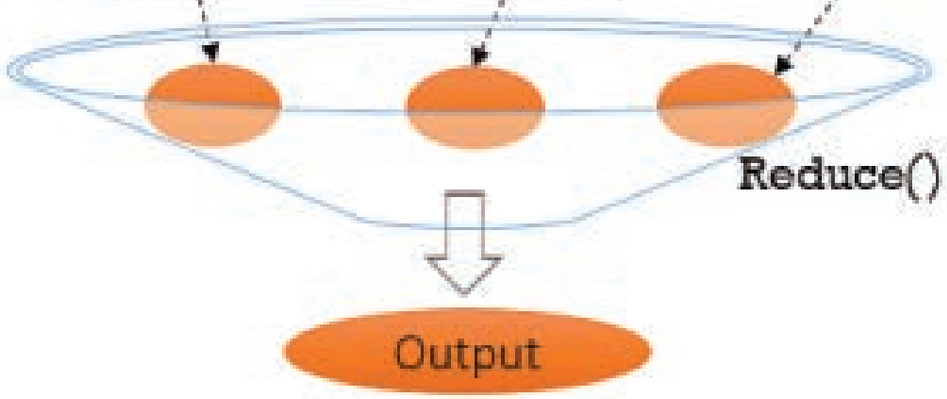


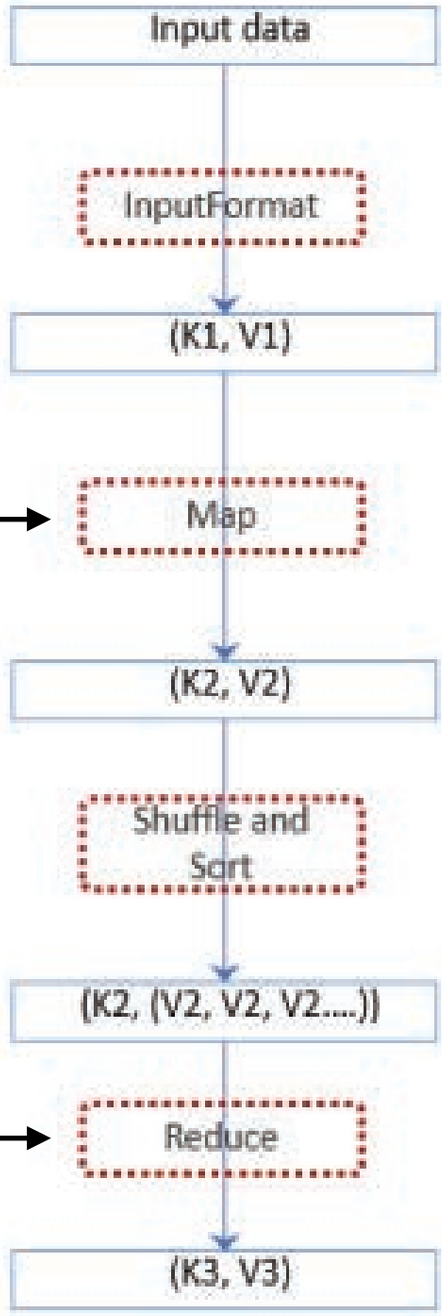
Split the input data – Based on sub-divide & conquer approach

Apply the same Map function on each of the key-value pair

Reduce

Shuffle and sort the Map-outputs and then apply Reduce function to combine the output to produce aggregated value for each key





Benefits

- Simplicity in parallelism
  - Parallelism is implicit in the model: Map and Reduce functions can be applied in a *shared-nothing* parallelism fashion
  - MapReduce programs can be automatically parallelized over an Hadoop cluster
- Fault tolerance
  - If a slave fails in computing a part of the Map or Reduce phase, the system detects this and re-assign the missing work to another slave in a transparent manner
  - HDFS replication is crucial for this
- Scalability
  - As the data load increases, parallelism level increases by adding slaves to the cluster
- Data locality
  - Hadoop maximize proximity between the slave where data is stored and the slave where processing happens.
  - In the Map phase, most of Mappers are executed on the nodes storing input blocks

A practical 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: 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.
- Map: takes as input a couple (k,v) returns a pair (w,1) for each word w in v
- Shuffle&Sort: group 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)

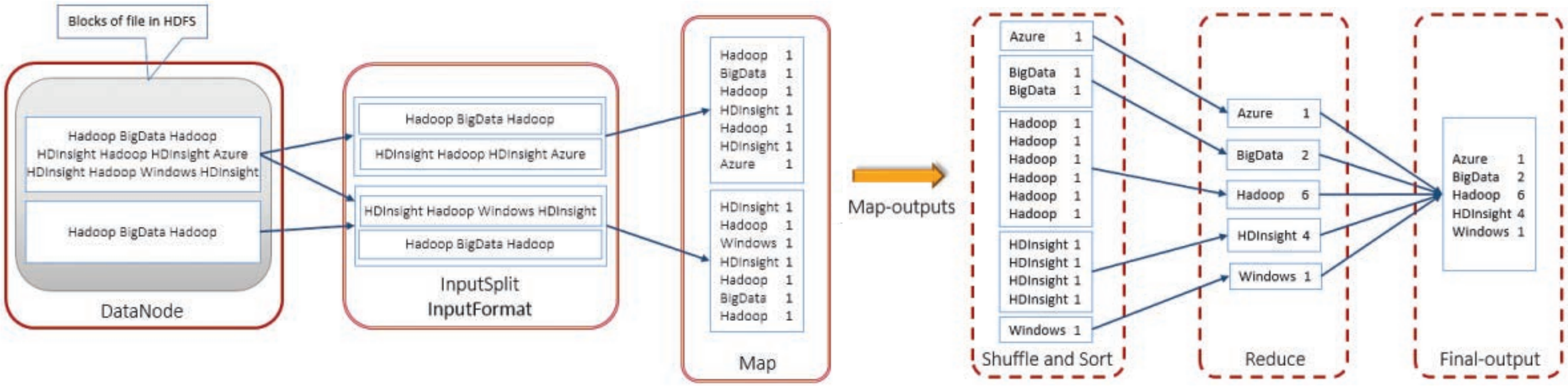
  c=0

  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

- 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 (*)
- 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 to shuffle and sort (and to sent trough the network)
- 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 se when Combiner is triggered.

Pseudo code with Combine

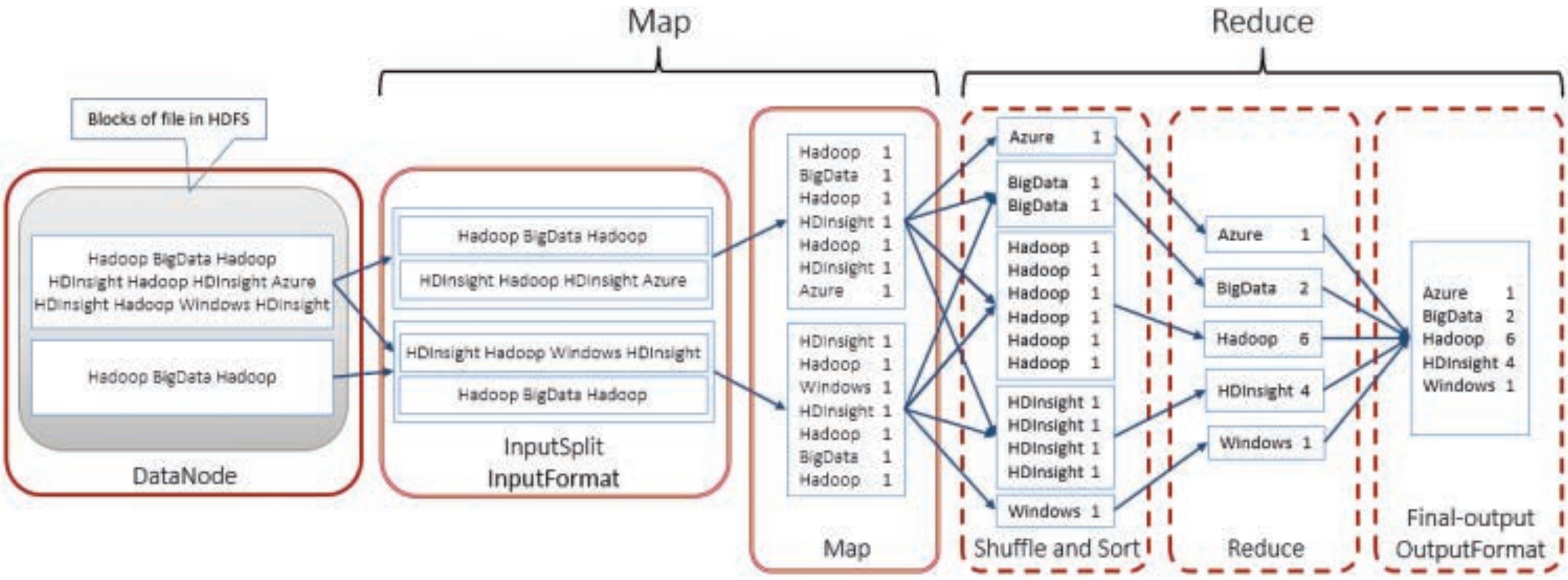
- Map(k, v)
  for each w in v
    emit(w, 1)

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

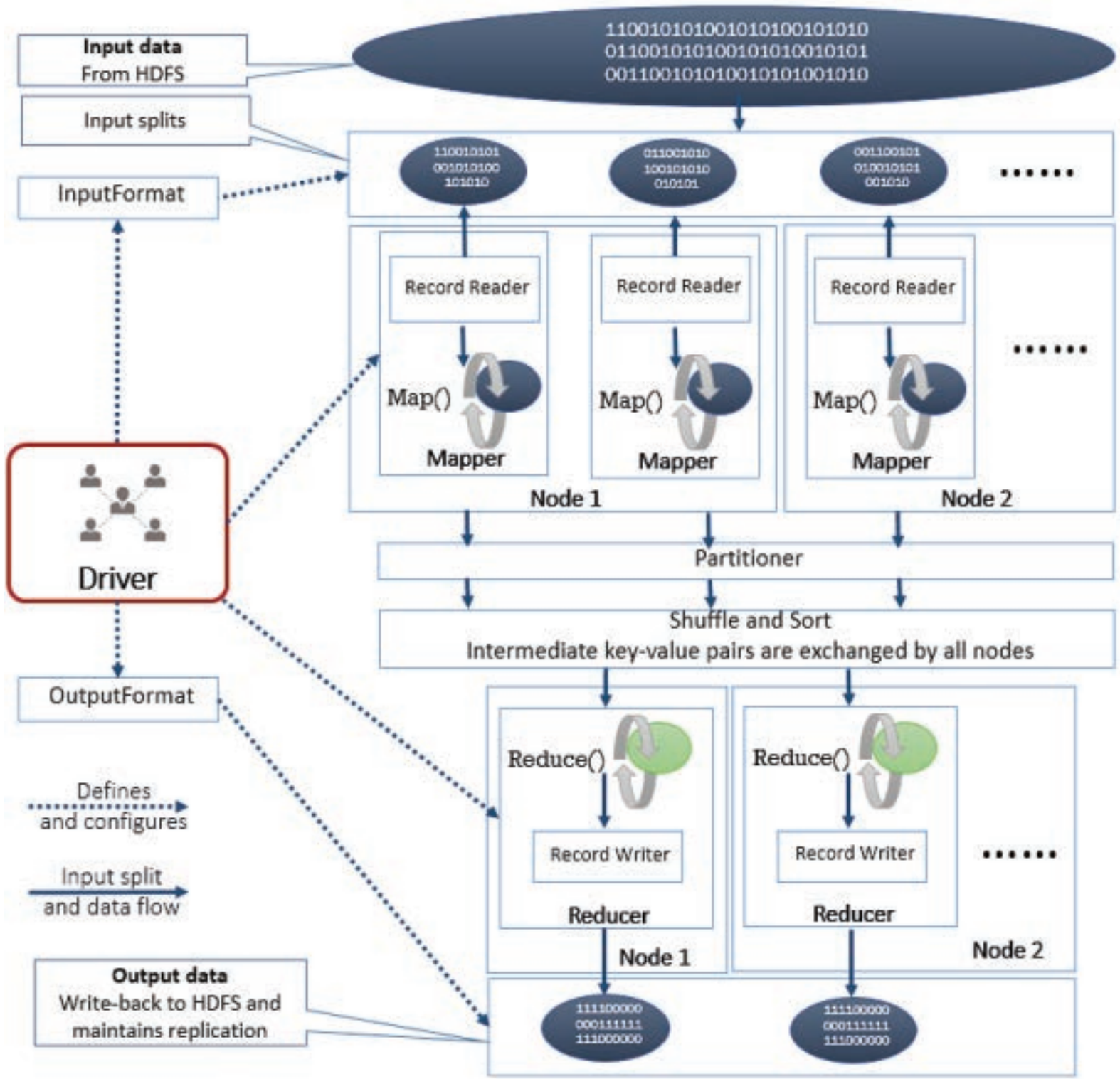
- Reduce(k, v)
  c=0
  for x in v
    c = c +x
  emit(k, c)

Note that, as usual, Combine is isomorphic to Reduce.
This is because the sum operation performed by WordCount is associative.
We will see different cases next.

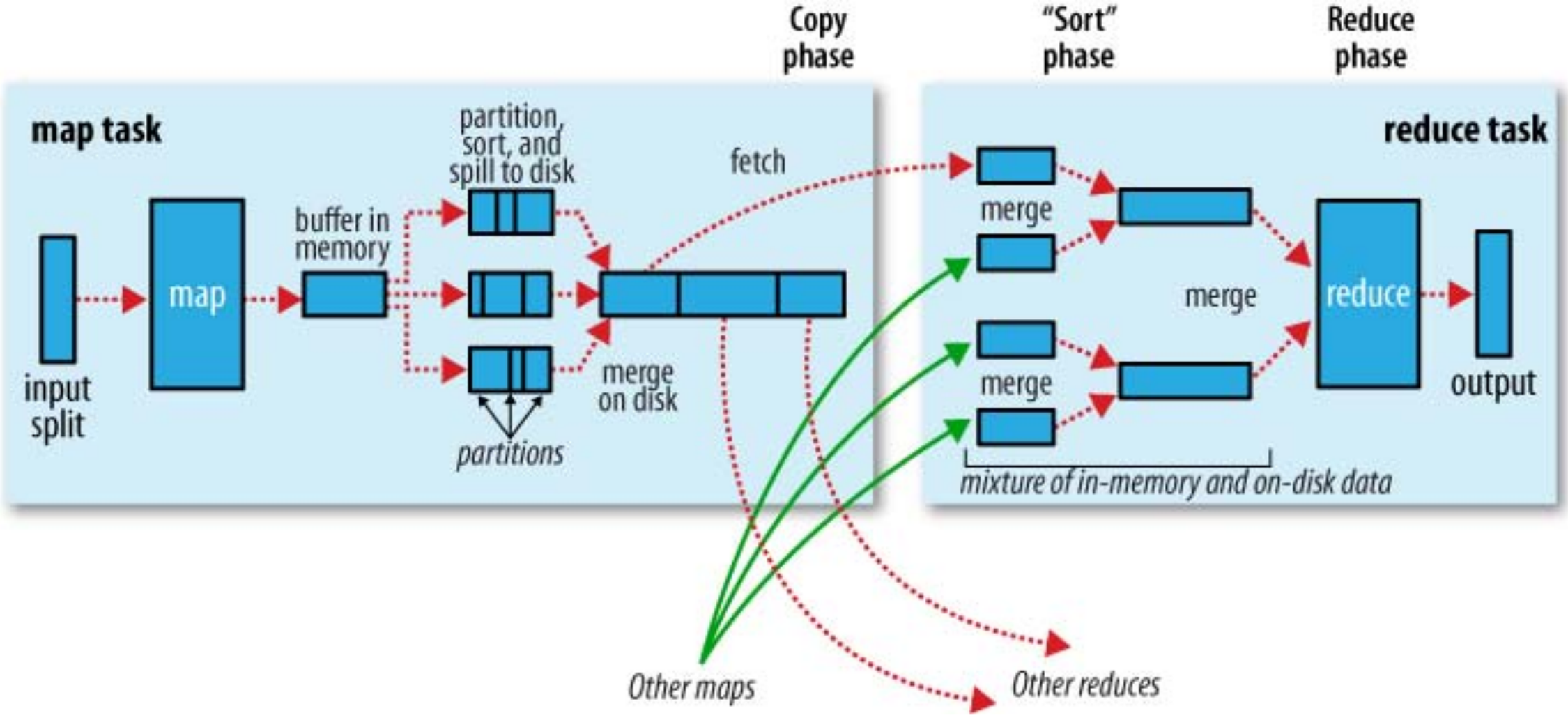
Example of data flow



General high-level execution flow of a Map-Reduce job



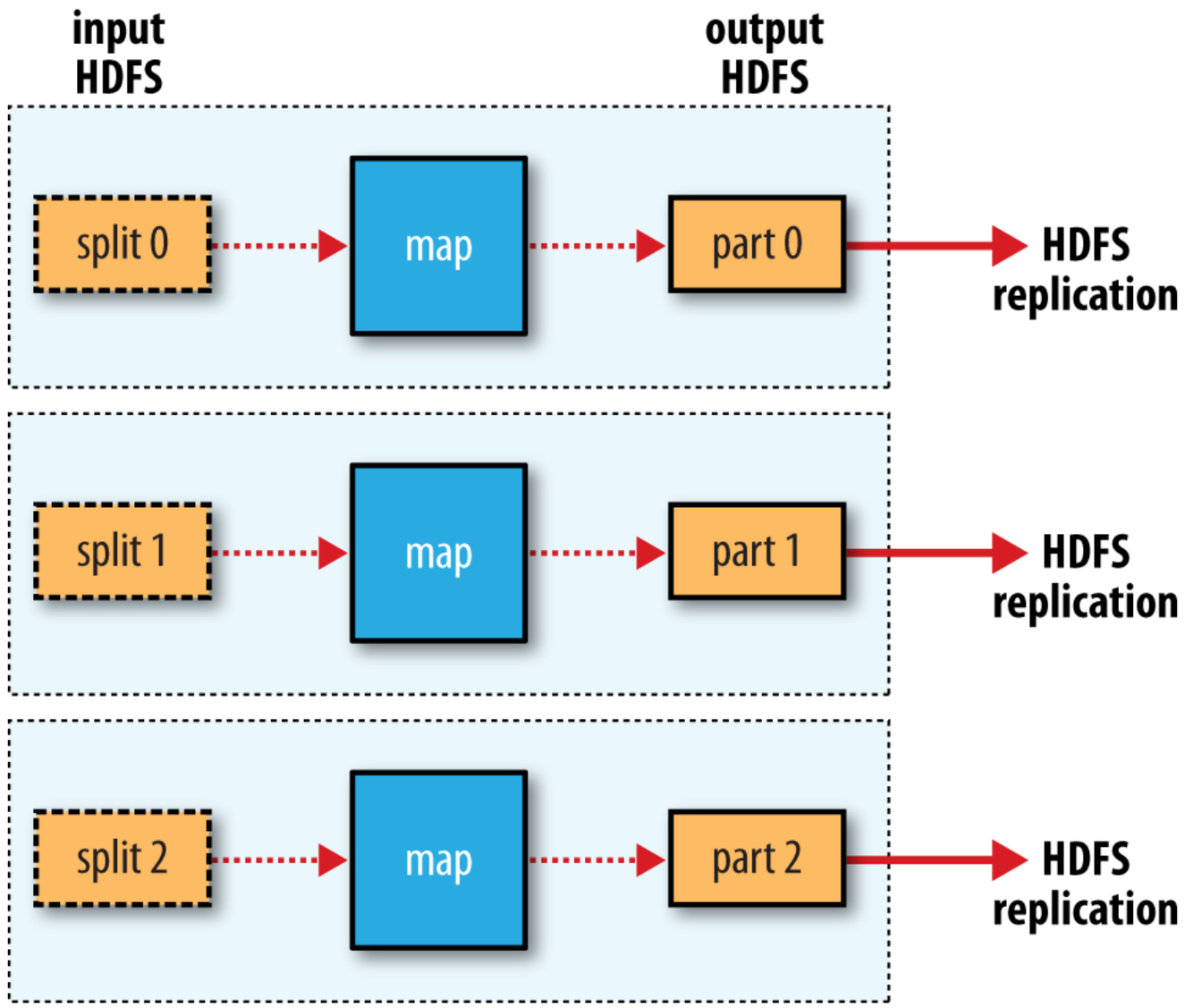
A deeper look at shuffle&sort



By default, a **spill** file is created each time the buffer memory is 80% full

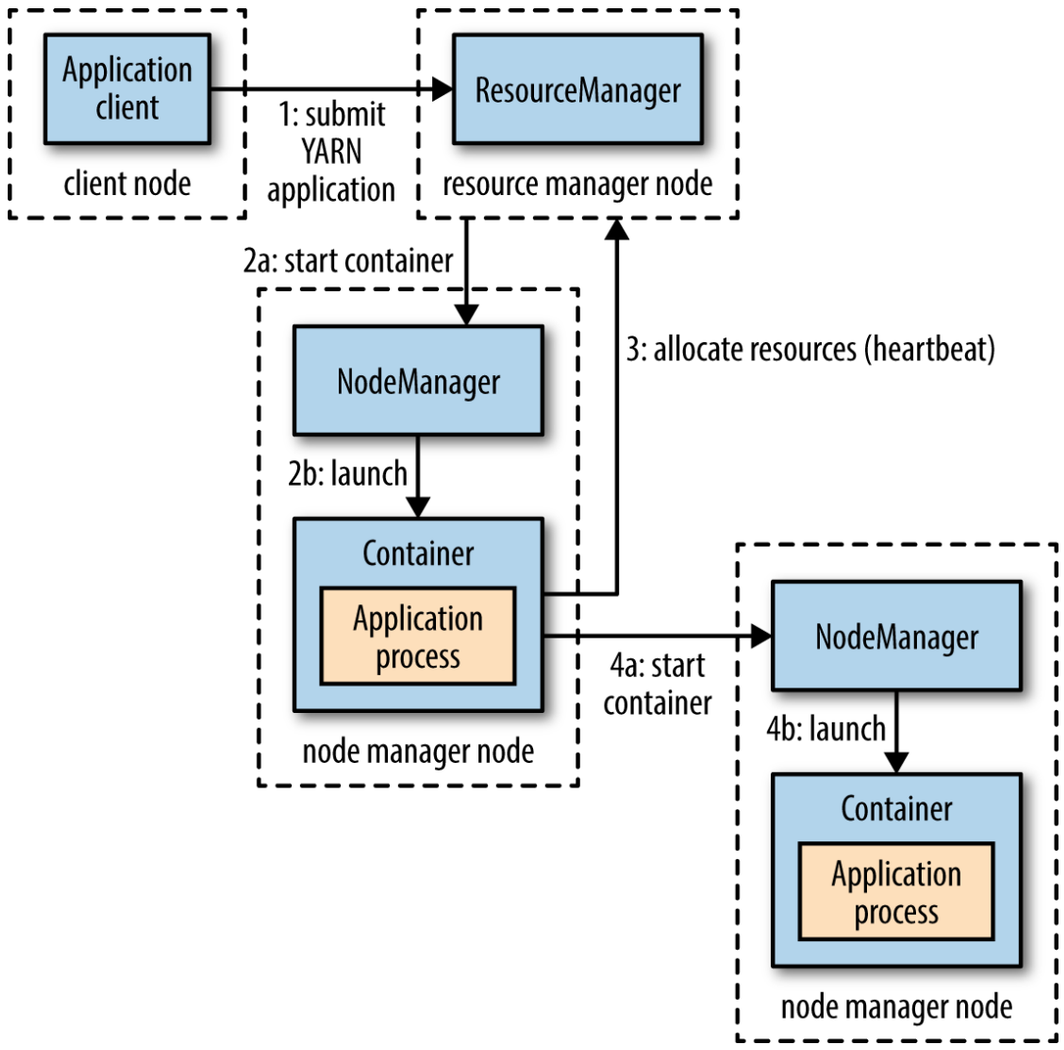
Combine is triggered if at least tree spill files are created.

Dataflow with no reduce



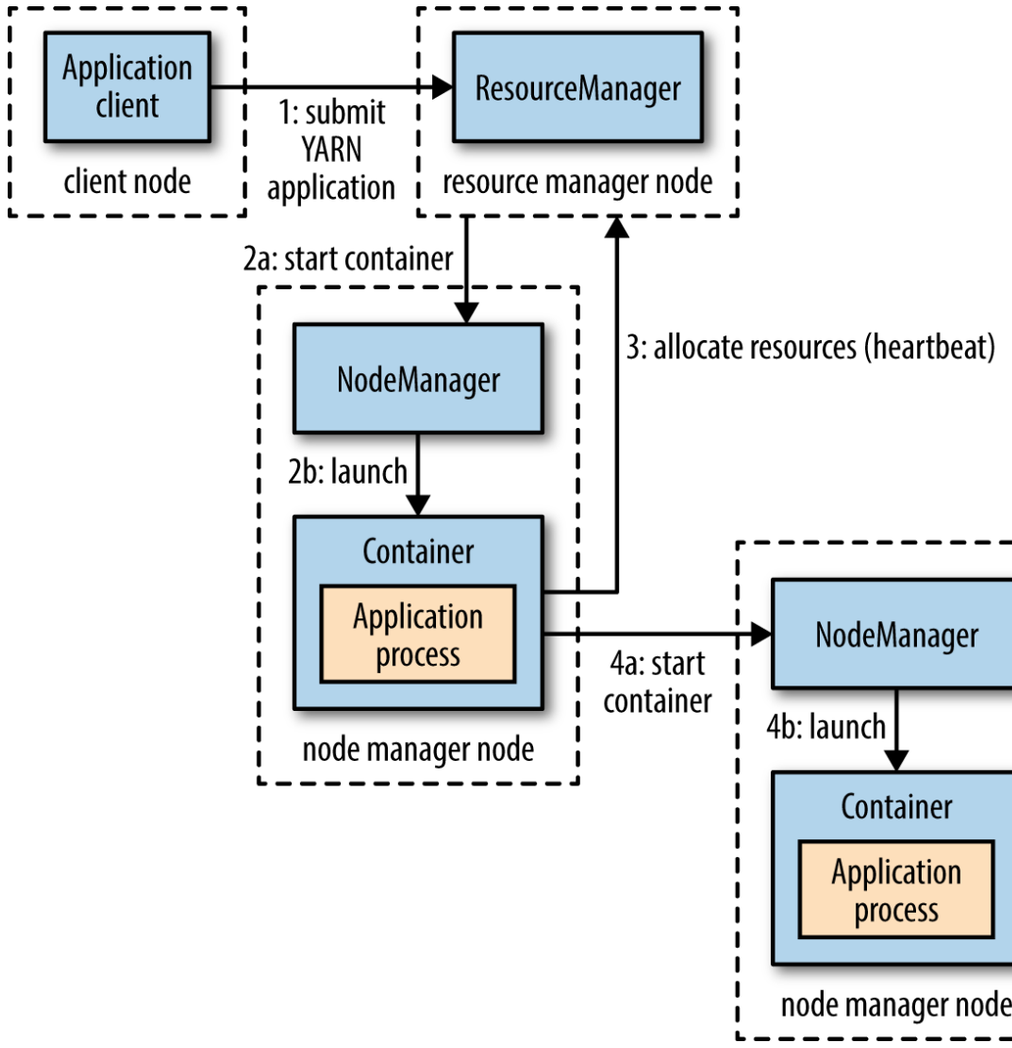
YARN

- 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 and 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

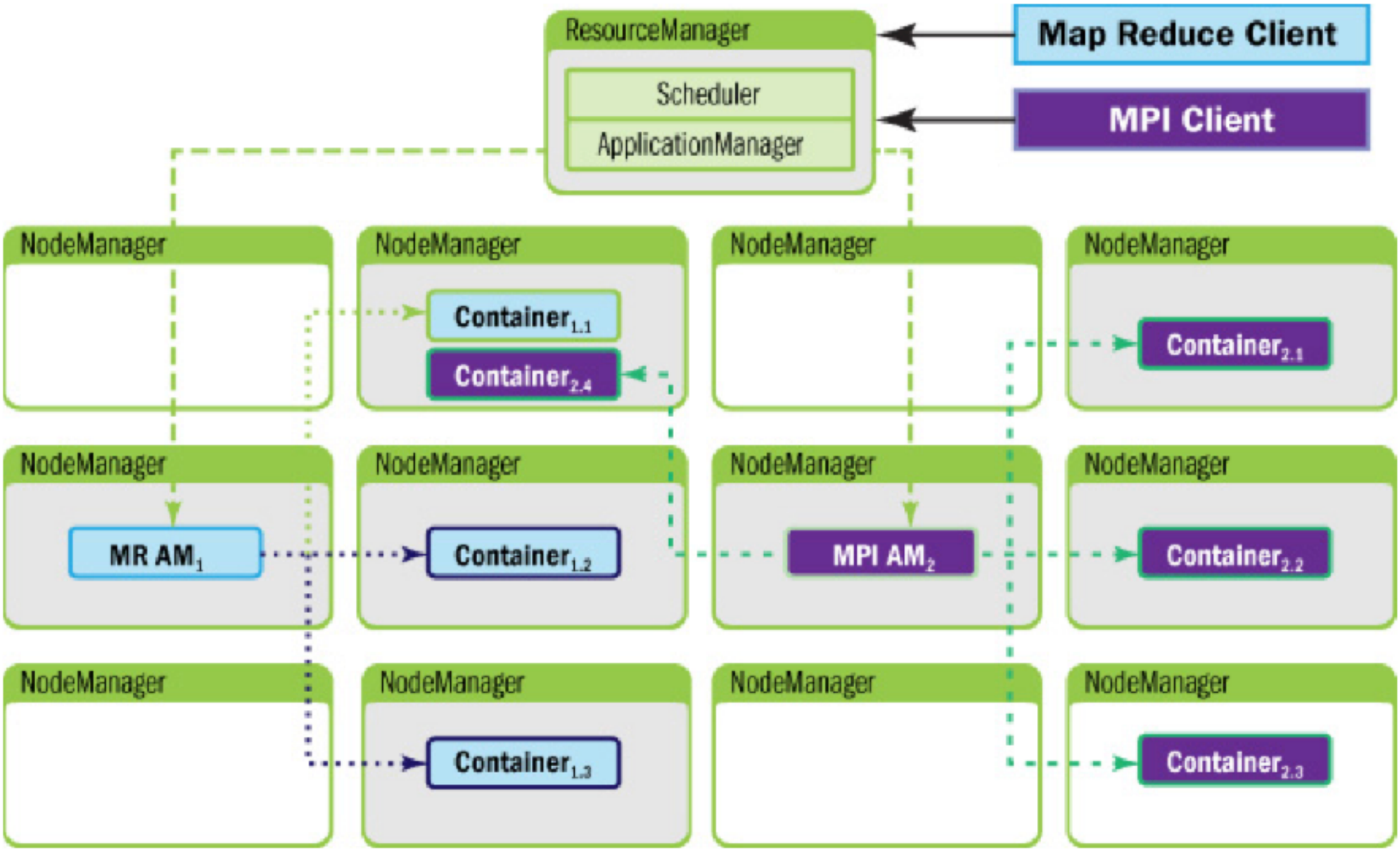


YARN

- 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 running of the application; this is the application manage.
- 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 like for Spark, or at run time (step 4)
- At the end of the process, the application managers informs the resource manager, which will kill the allocated containers.



YARN as a data OS



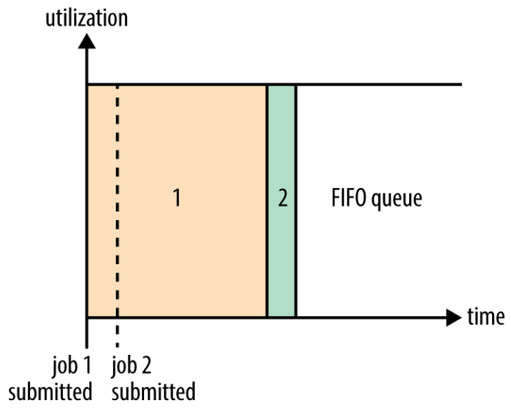
YARN scheduling policies

The kind of scheduling policy has to be set in the yarn-default.xml configuration file

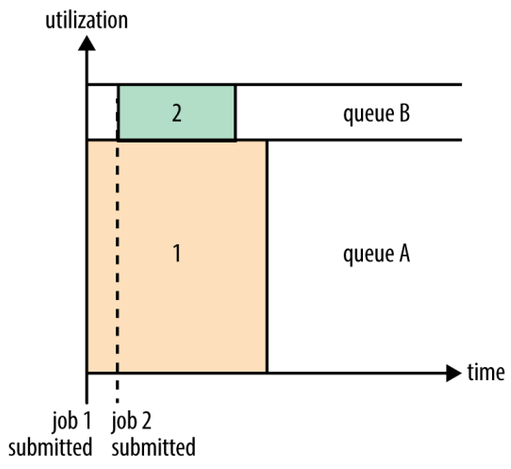
- FIFO, for clusters with ‘small’ workloads
- Capacity Scheduler (introduced by Yahoo! for large clusters)
    - the administrator configures several queues with a predetermined fraction of the total resources (this ensures minimal resources for each queue)
    - each queue is FIFO, and has strict Access Control Policies to determine what user can submit to what queue
    - configuration can be changed dynamically, when feedback from the Node managers is available, more starved queues can have more resources
    - work best when there is knowledge about the kind of workload, this allows for setting minima resources for queues
- Fair Scheduler
    - each application is bound to a queue
    - YARN containers are given with priority to queues consuming less resources
    - within a queue the application having the fewest resources is assigned the asked container
    - this can imply resource reduction for running jobs
    - attention: in the presence of a single application, that application can ask for the entire cluster.

YARN scheduling policies

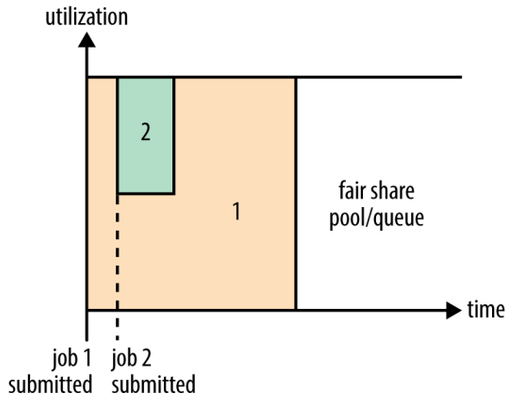
i. FIFO Scheduler



ii. Capacity Scheduler

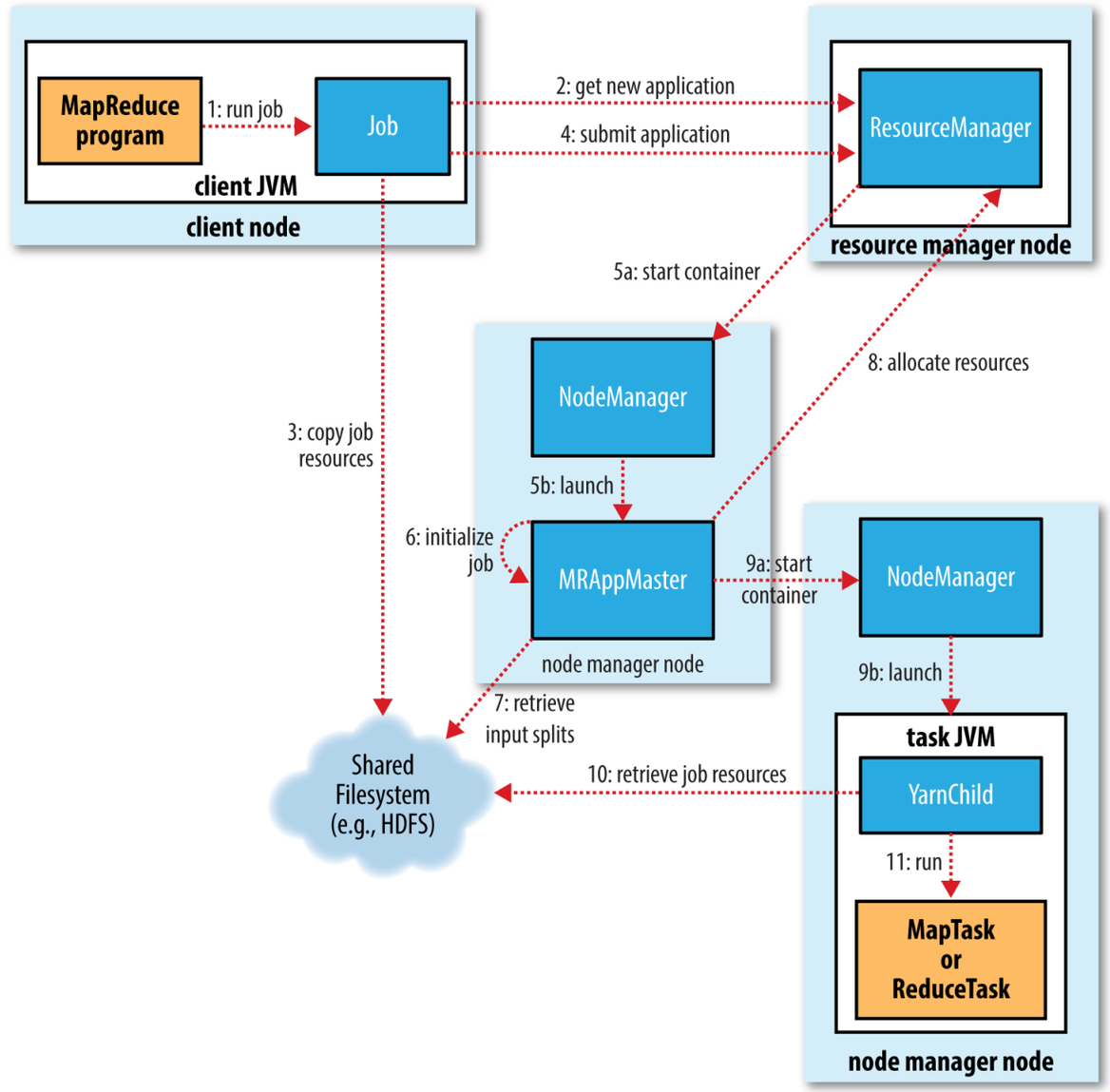


iii. Fair Scheduler



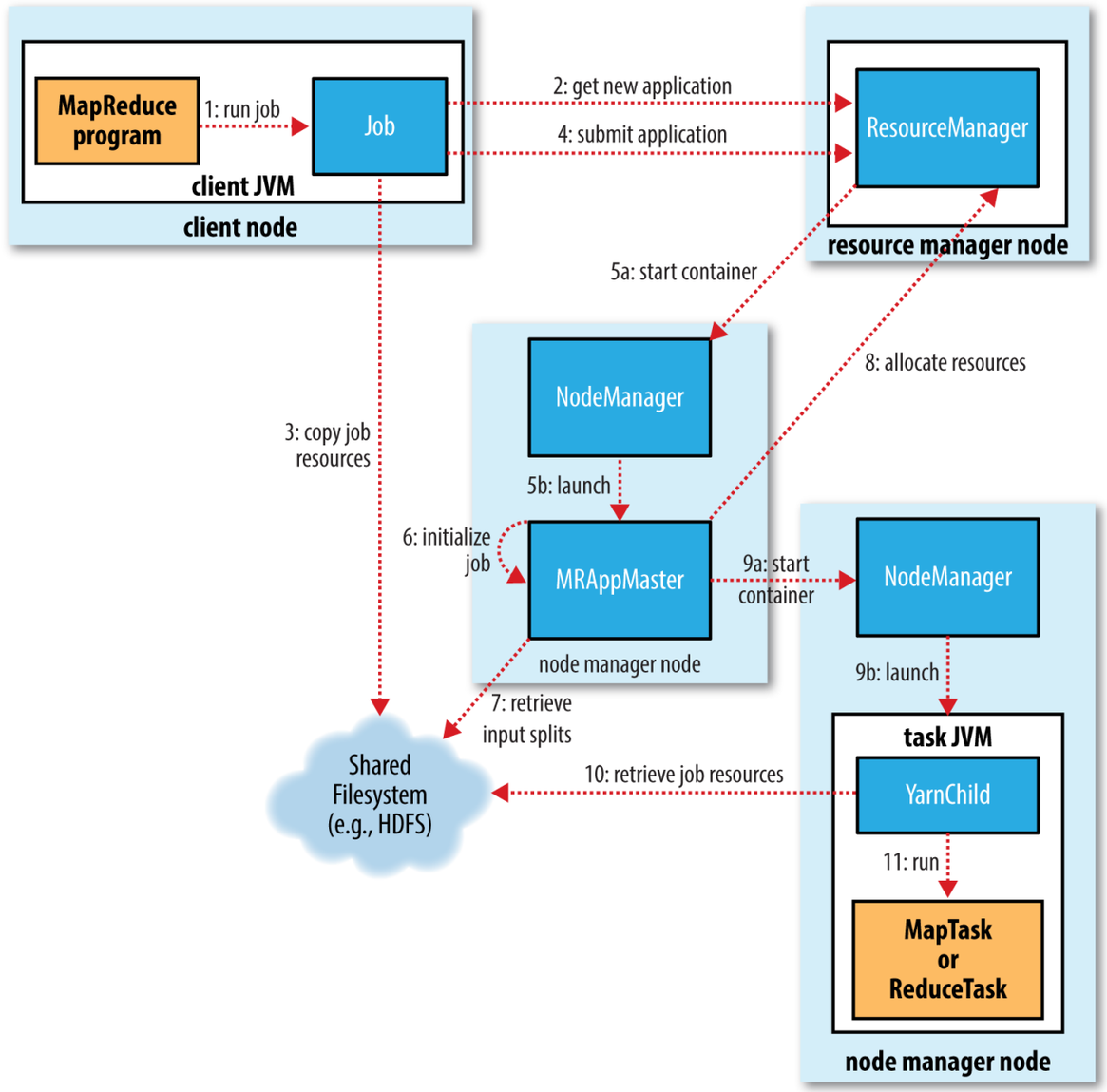
MapReduce on YARN

- As already said, MapReduce job execution relies on YARN
- In a first phase (steps 1-3) the client node asks the RM for a job ID and then sends all the information needed to start the JOB (….)
- The RM then starts a container (steps 5-6) for the the MR Application manager, which initialise the jobs and decide whether to run the job locally (uber execution, input size < size of a block) or on the cluster in parallel.
- In case of distributed execution, the AP asks for containers for Map and Reduce tasks (a container for each task)
- Only Map tasks container are allocated keeping into account locality when possible
- The number of running Map tasks depends on the number of input blocks
- The number of Reduce tasks is set by the user.



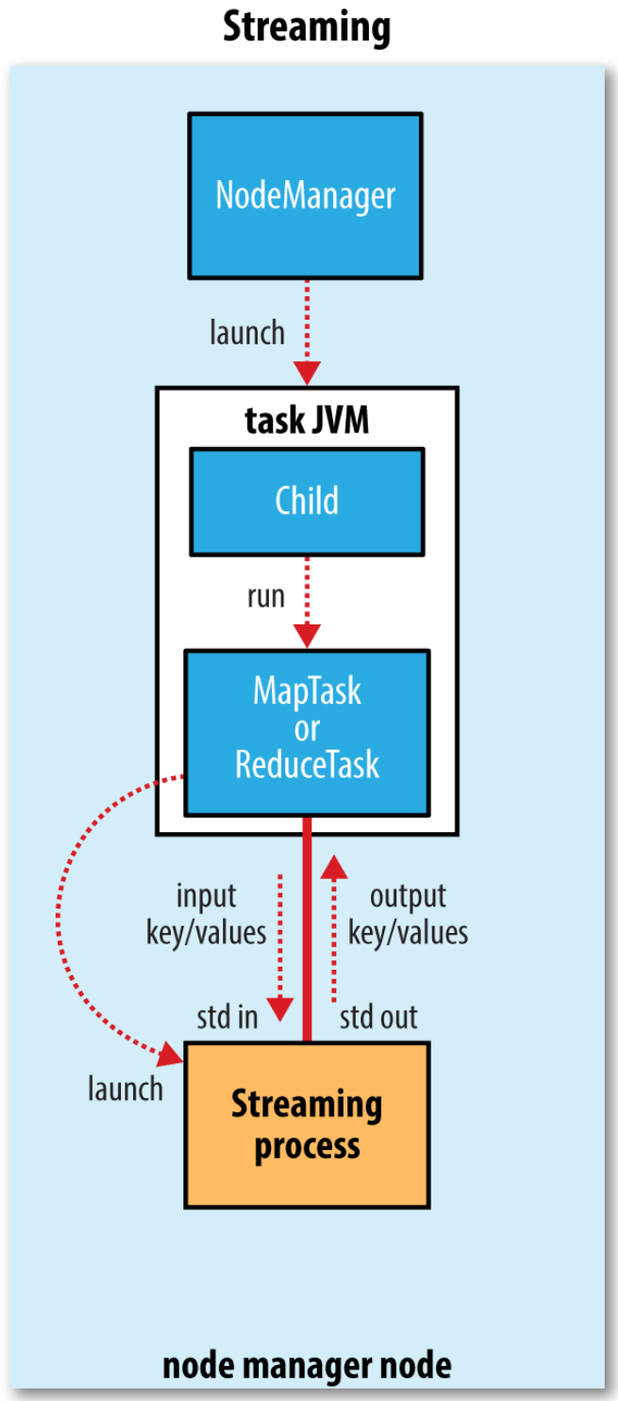
MapReduce on YARN

- Task execution is made within a dedicated Java Virtual Machine.
- If task succeeds it commits and inform the AM
- According to configuration, the same task can be run in parallel in order to augment robustness and efficiency (speculative task execution)
- Only one of speculative tasks commits, the other ones abort.
- The job succeeds if all the to be executed tasks succeed.
- Several statistics about execution are made available by Counters.



Hadoop streaming

- Although Hadoop is a Java-based system it can support execution of MapReduce jobs written in other languages.This is particularly important for us, as we are going to implement MapReduce jobs in Python.
- To this end we will rely on 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 node and can read records (pairs) on the Linux/ Unix **standard input stream (stdin) and 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



Now it is time to make practice with developing, implementing and running MapReduce algorithms.

One-job Join in MapReduce

- Consider the inner join query seen last time
- We saw that a general solution consists of using 3 jobs: the first twos to prepare data by adding table provenance information, the third one to perform the join
- We see now how to make data preparation directly in the map, thus avoiding the first two jobs.

One-job Join in MapReduce

- Key idea: when reading a line in the map phase recover the name of the file the line comes from
    - this gives us the provenance flag
        - we just need to extract the file name in path returned by

```
a = os.environ['mapreduce_map_input_file']
```

- Lets see the code.

The Map

```
#!/usr/bin/python

import os

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
    attributes = line.split(",")
    # read name of file from os environment
    a = os.environ['mapreduce_map_input_file']
    #a="/Users/dario/Dropbox/TEACHING-2016/X/BigData/Data/Tables/data/Customer.txt'
    l = a.split("/")
    namefile = l[len(l)- 1]

    if namefile !='Customer.txt' :
        print '%s\t%s' % ( attributes[0], namefile +","+ attributes[1])
    elif attributes[1][3:5]=="07" :
        print '%s\t%s' % (attributes[0], namefile )
```

The Reduce

```
#!/usr/bin/python

import sys

current_cid = None
current_table=None
cid = None
lvalues =[]

# input comes from STDIN
for line in sys.stdin:
    # remove leading and trailing whitespace
    pair = line.strip().split('\t')

    # parse the input we got from mapper.py
    cid = pair[0]
    rest = pair[1]

    if current_cid == cid:
        lvalues.append(rest)

    else:
        if current_cid:
            # write result to STDOUT
            if 'Customer.txt' in lvalues :
                for x in lvalues :
                    if x != 'Customer.txt' :
                        print '%s\t%s' % (current_cid, x.split(",")[1])

        current_cid = cid
        lvalues=[rest]

if current_cid == cid:
    if 'Customer.txt' in lvalues :
        for x in lvalues :
            if x != 'Customer.csv' :
                    print '%s\t%s' % (cid, x.split(",")[1])
```

Job execution on the cluster

- Please send me an email (dario.colazzo@dauphine.fr) with subject ‘PhD course BigData’, I’ll send you a private key and useXY name.
- To connect to the cluster:
  - $ chmod 600 <private key file>
  - $ ssh -i <path to your private key> -p 993 userXY@www.lamsade.dauphine.fr
- Once connected create a directory for the data and one for the code
- Copy in your data directory the following two simple tables
  - /home/dario.colazzo/data/smalltables/Customer.txt
  - /home/dario.colazzo/data/smalltables/Order.txt
- Copy in your code directory the mapper and reducer, do not forget chmod +x.
  - /home/dario.colazzo/code/python/join/mapper.py
  - /home/dario.colazzo/code/python/join/reduce.py
- Copy input data into your input data directory in HDFS
  - You need first to create the input data directory (in my case /user/dario.colazzo/data/smalltables)

Launching the job

hadoop jar \

/opt/hadoop/hadoop-2.7.3_master/share/hadoop/tools/lib/hadoop-streaming-2.7.3.jar \

-input path to your HDFS directory containing tables \

-output path to your HDFS directory containing output (for instance, /user/dario.colazzo/output)

-file local path to mapper.py \

-file local path to reducer.py \

-mapper local path to mapper.py \

-reducer local path to reducer.py