CLOUD COMPUTING

Distributed Processing: Hadoop and MapReduce

PAUL TOWNEND

ASSOCIATE PROFESSOR, UMEÅ

Hadoop

WHAT IS HADOOP?

Originally conceived in 2006 by
Doug Cutting and **Mike Cafarella**
(Yahoo! and the University of Washington)

Hadoop is an open-source framework for **processing, storing and analysing data**

The fundamental principle is that it's more efficient to store and process data by **breaking it up and distributing it into many parts**

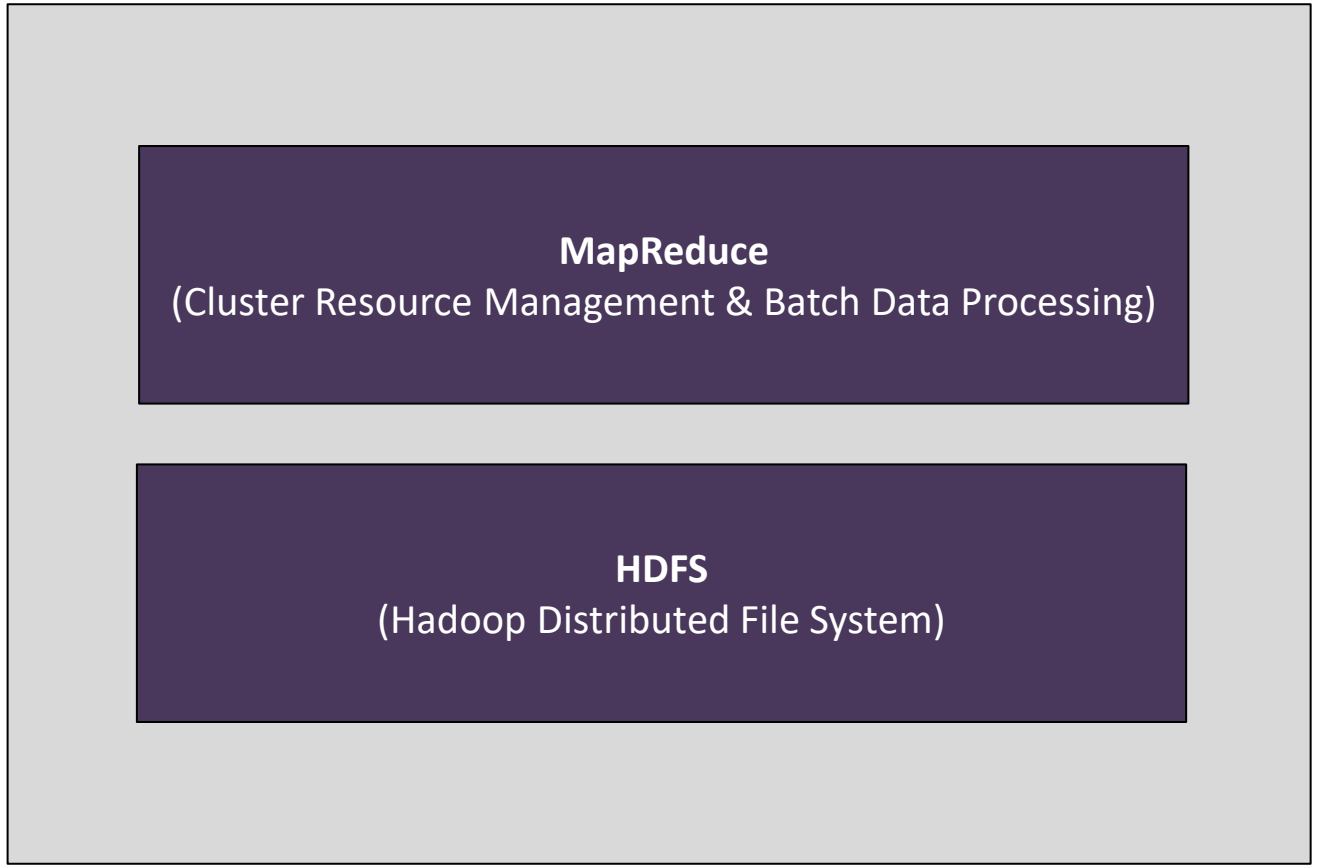
Hadoop is a whole ecosystem of different products, largely presided over by the **Apache Software Foundation**.

HADOOP VERSION 1

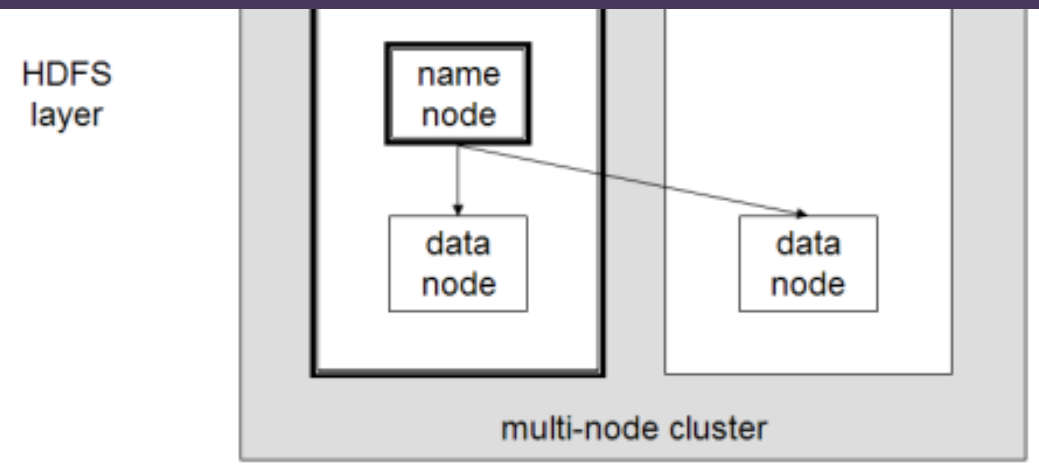
The first version of Hadoop consisted of two core components:

HDFS and **MapReduce**

Both heavily inspired by Google



HIGH LEVEL ARCHITECTURE



Typical workflow

Load data into the cluster
(HDFS writes)

Store results in the cluster
(HDFS writes)

Read the results from the cluster
(HDFS reads)

HDFS

HADOOP DISTRIBUTED FILE SYSTEM

Inspired by the **Google File System** (GFS – 2003)

In addition, there are a number of **DataNodes**, usually one per node in the cluster, which manage storage attached to the nodes that they run on.

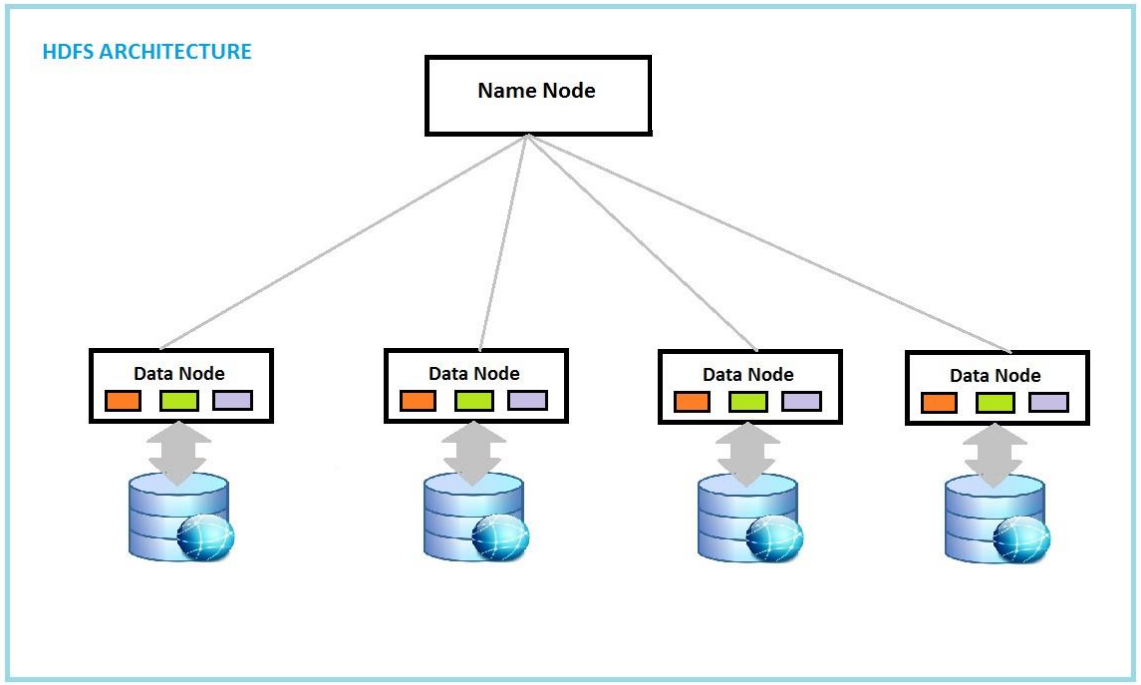
HDFS has a **primary** / **secondary** architecture

HDFS exposes a file system namespace and allows user data to be stored in files.

An HDFS cluster consists of a single **NameNode** - the primary server that manages the file system namespace and regulates access to files by clients.

Internally, a file is split into one or more blocks and these blocks are stored in a set of DataNodes.

HDFS V1 ARCHITECTURE



The **NameNode** executes file system operations like opening, closing, and renaming files/directories. It determines the mapping of blocks to **DataNodes**

The **DataNodes** serve read and write requests from the file system's clients. They perform block creation, deletion, and replication on instruction from **NameNode**.

Blocks in **DataNodes** are copied across various nodes as specified by the system preferences, so that if one **DataNode** fails, another **DataNode** containing the required data automatically comes online.

HDFS is built using **Java**; any machine that supports Java can run the **NameNode** or the **DataNode** software.

HDFS FILE WRITES

Client consults NameNode

↓

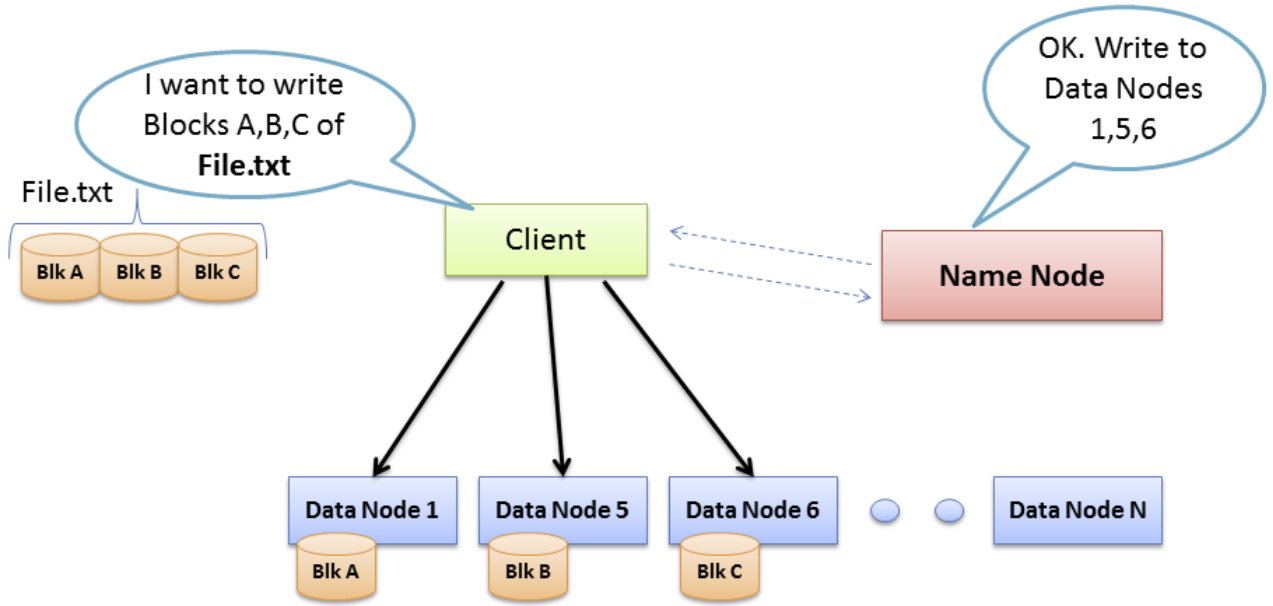
Client writes block directly to one DataNode

↓

DataNode replicates block

↓

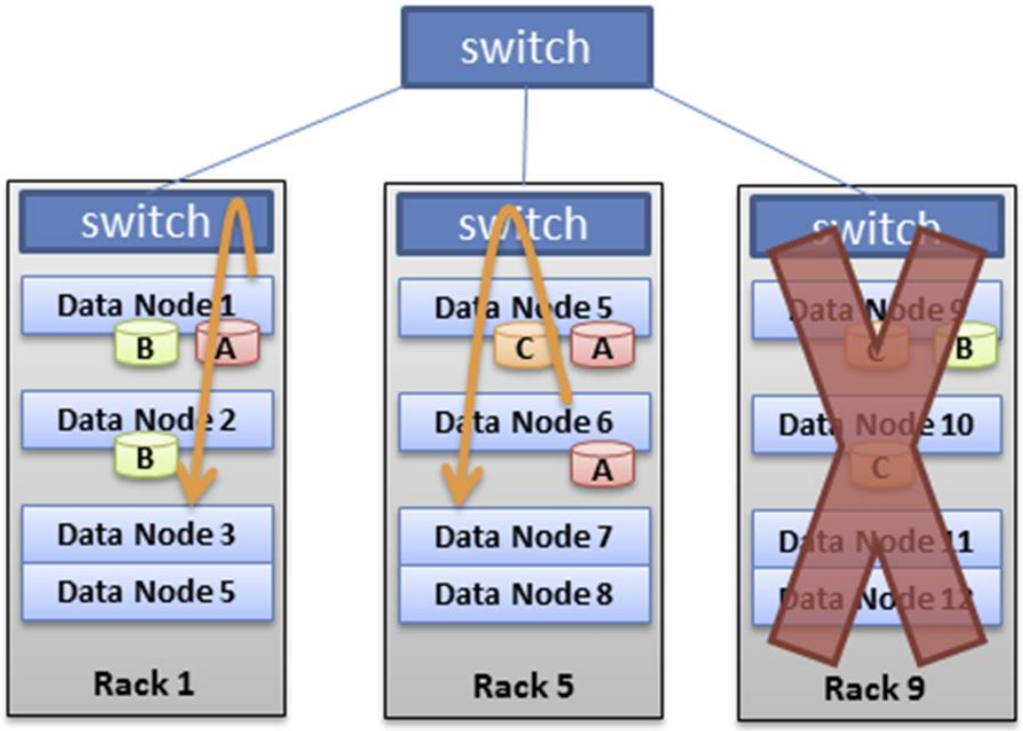
Cycle repeats for next block



The more blocks you have, the more machines will be able to work on the data in parallel

For fault-tolerance, each block is replicated as it is loaded. By default 3 copies of each block in the cluster. But you can configure this.

HADOOP RACK AWARENESS



With rack awareness, we can ensure data is not lost upon the failure of an entire rack

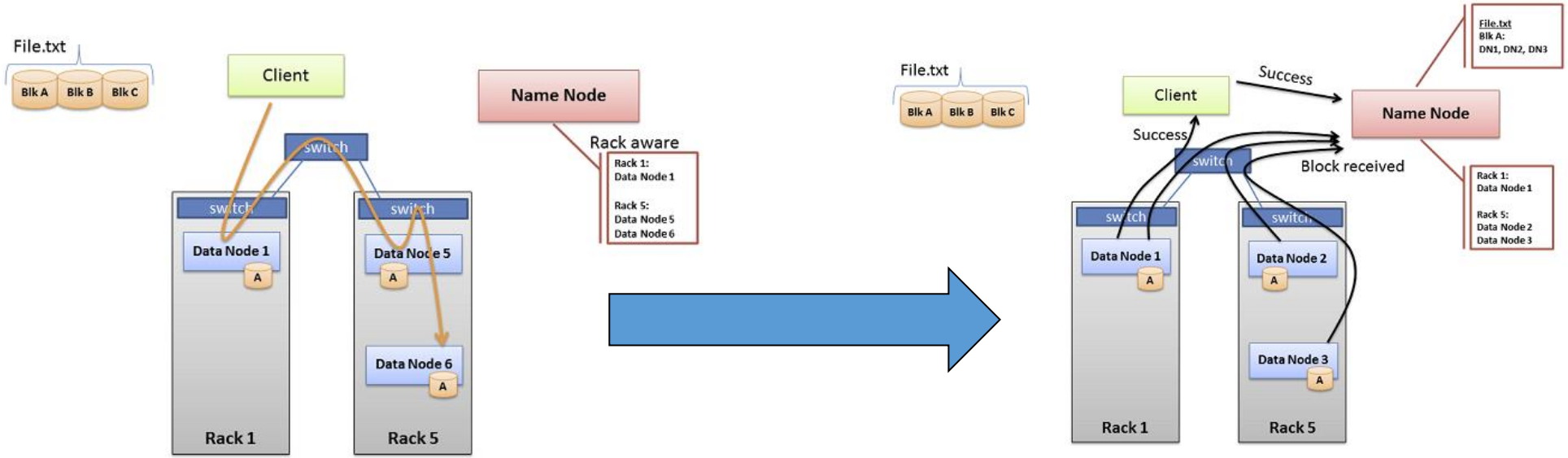
Can keep bulky data flows in-rack whenever possible

We assume that in-rack has higher bandwidth and lower latency

Rack numbers for each secondary Data Node need to be manually defined

We assume that in-rack has higher bandwidth and lower latency

HDFS PIPELINED WRITE



Before a client writes a block to a cluster, it checks to see if all DataNodes which are expected to have a copy are ready.

Above, Client connects to DataNode 1, which then queries DataNode5, which then queries DataNode6.

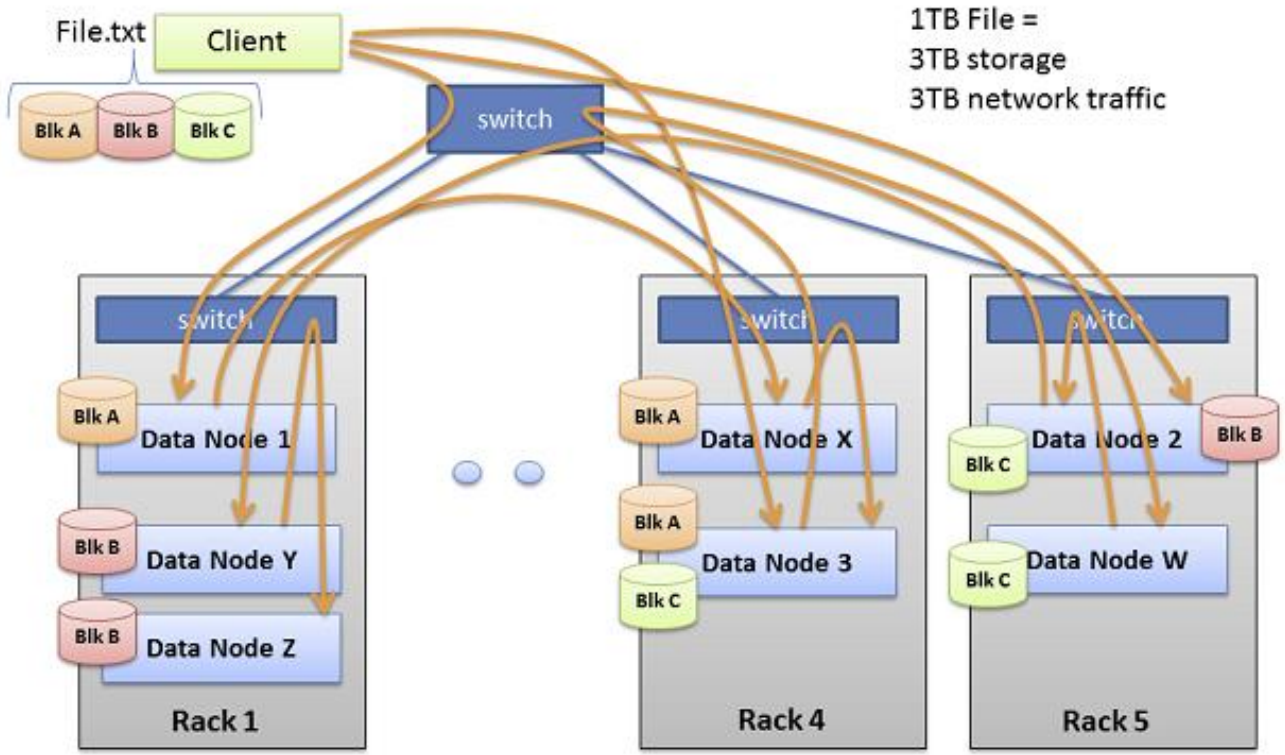
If all are ready, then DataNode 1 sends a “Ready” message back to the Client

As data is written for each block in the cluster, a **replication pipeline** is created.

This means that as a DataNode receives block data, it will at the same time push a copy to the next node in the pipeline.

Rack Awareness can again play a part in this (intra-rack comms are faster).

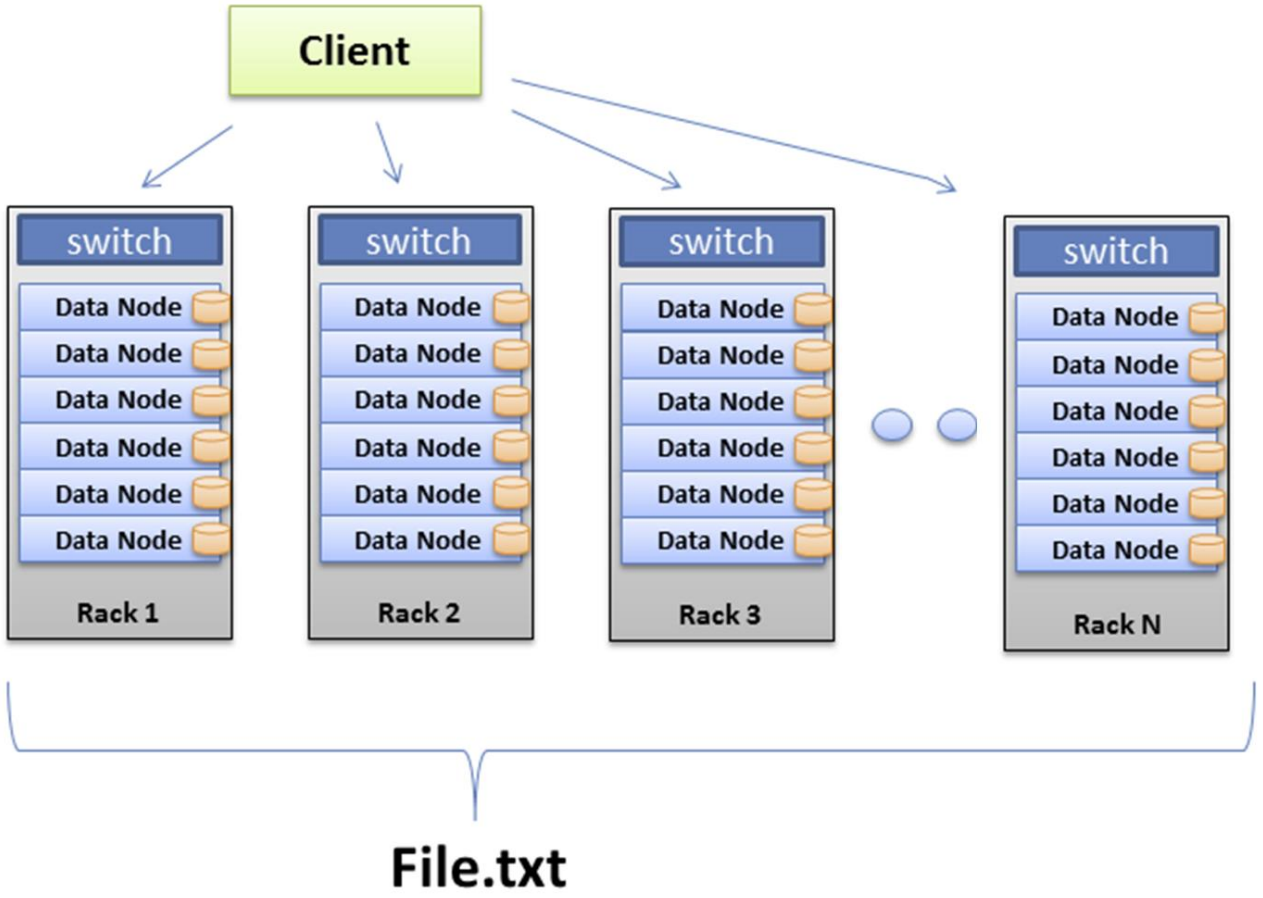
HDFS MULTI-BLOCK REPLICATION PIPELINE



As each block in a file is written, the initial node in the pipeline will vary. This spreads around the traffic.

Hadoop uses a lot of network bandwidth and storage, depending on the replication factor configured.

HDFS FILE SPREAD



Ideally, a file should eventually be spread across an entire cluster of machines

The more blocks that make up a file, the more machines the data can be spread on, and the more parallel processing power

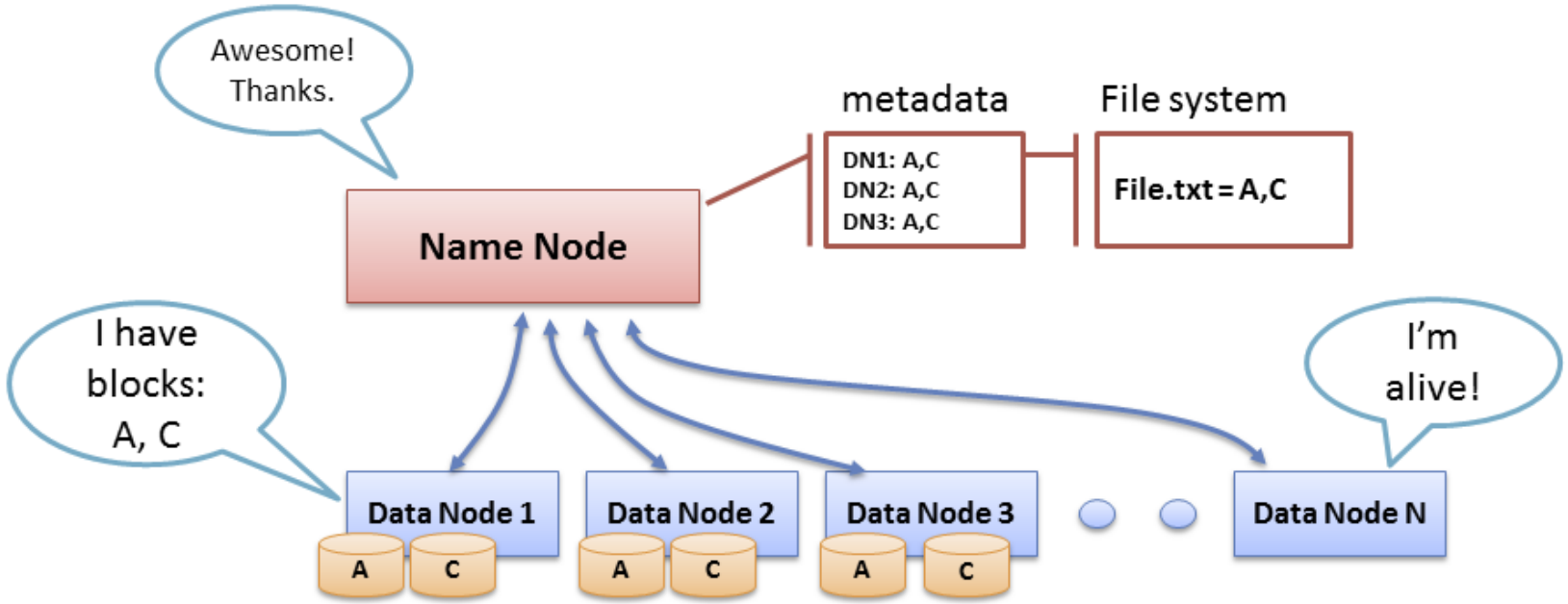
When the cluster goes wide, our network needs to scale appropriately.

HEARTBEATS

In addition to holding all file system metadata, the NameNode oversees the health of DataNodes and coordinates data access.

DataNodes send “heartbeat” messages to the NameNode every 3 seconds by TCP.

Every 10th heartbeat is a **block report**. This allows a NameNode to build its metadata and ensure the required number of replicated blocks is met.

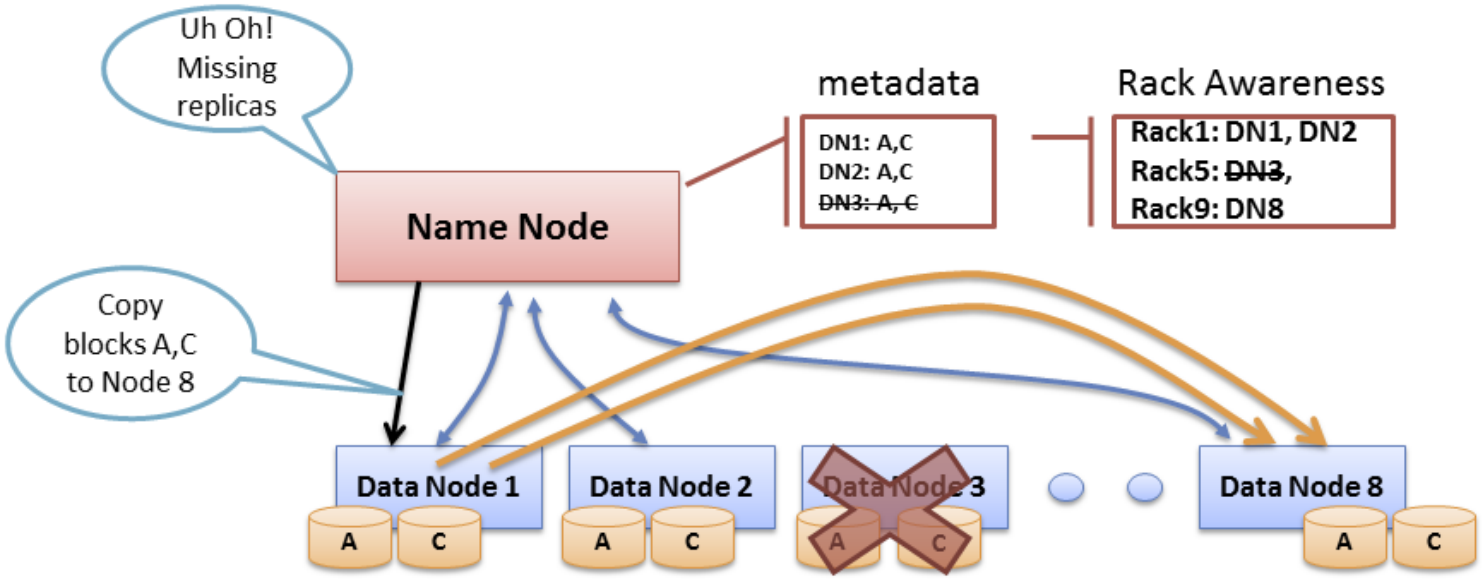


RE-REPLICATING MISSING REPLICAS

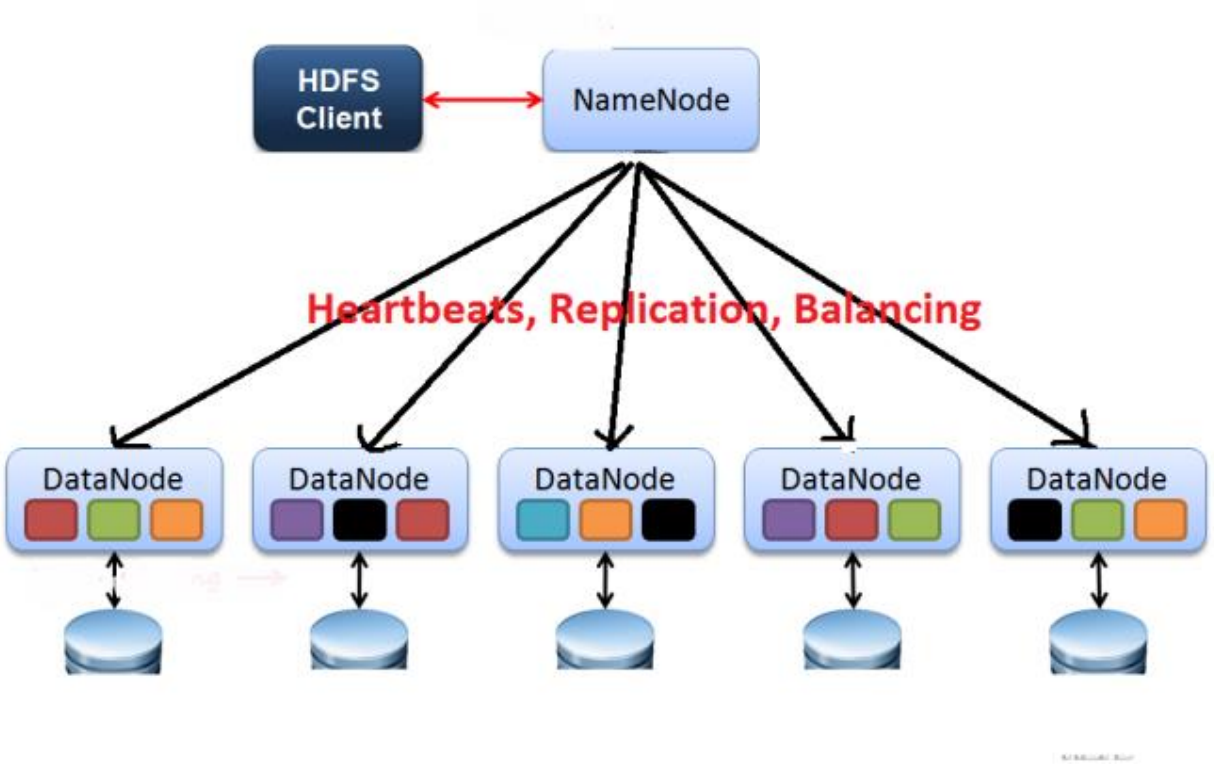
If a NameNode stops receiving heartbeats from a DataNode, it presumes it to have failed.

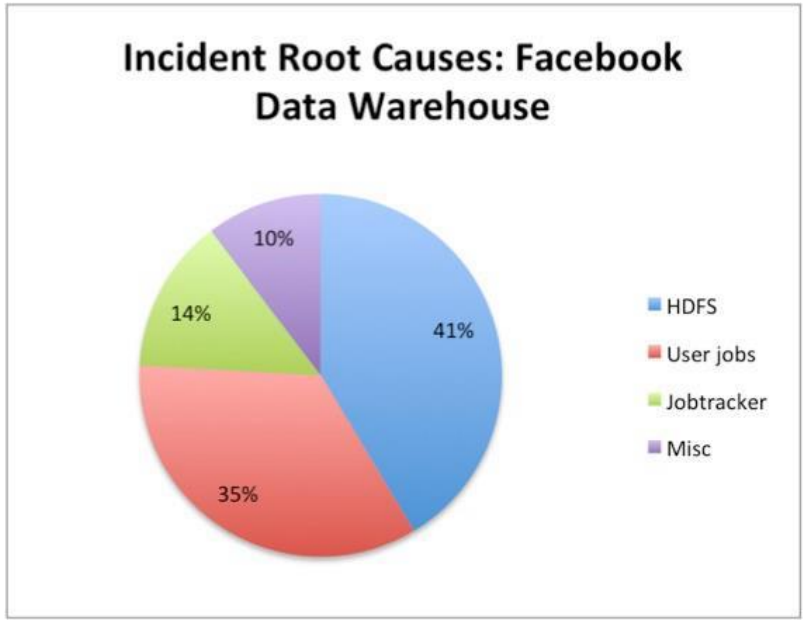
Based on the block reports it received, it knows which copies of blocks the failed node has, and can re-replicate those blocks to other DataNodes.

It can also – once again – consider rack awareness (multi-rack copies).



HADOOP DISTRIBUTED FILE SYSTEM





What is the weakness in this architecture?

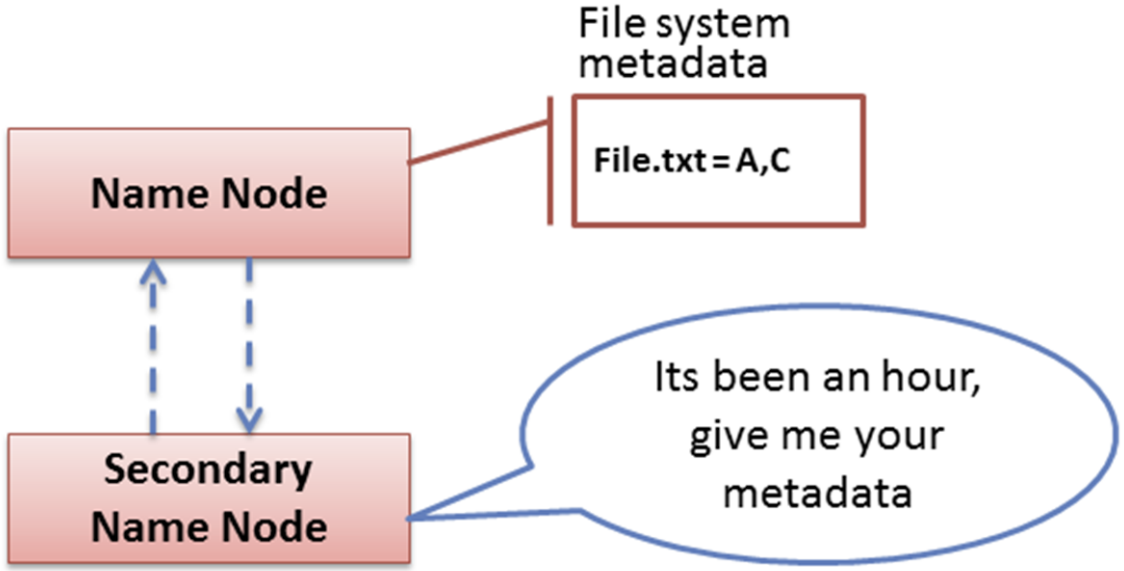
HDFS SECONDARY NAMENODE

Hadoop has a specific server role called the **Secondary NameNode**.

This node connects to the NameNode at regular intervals (default is one hour)
and copies its memory- and file- based metadata.

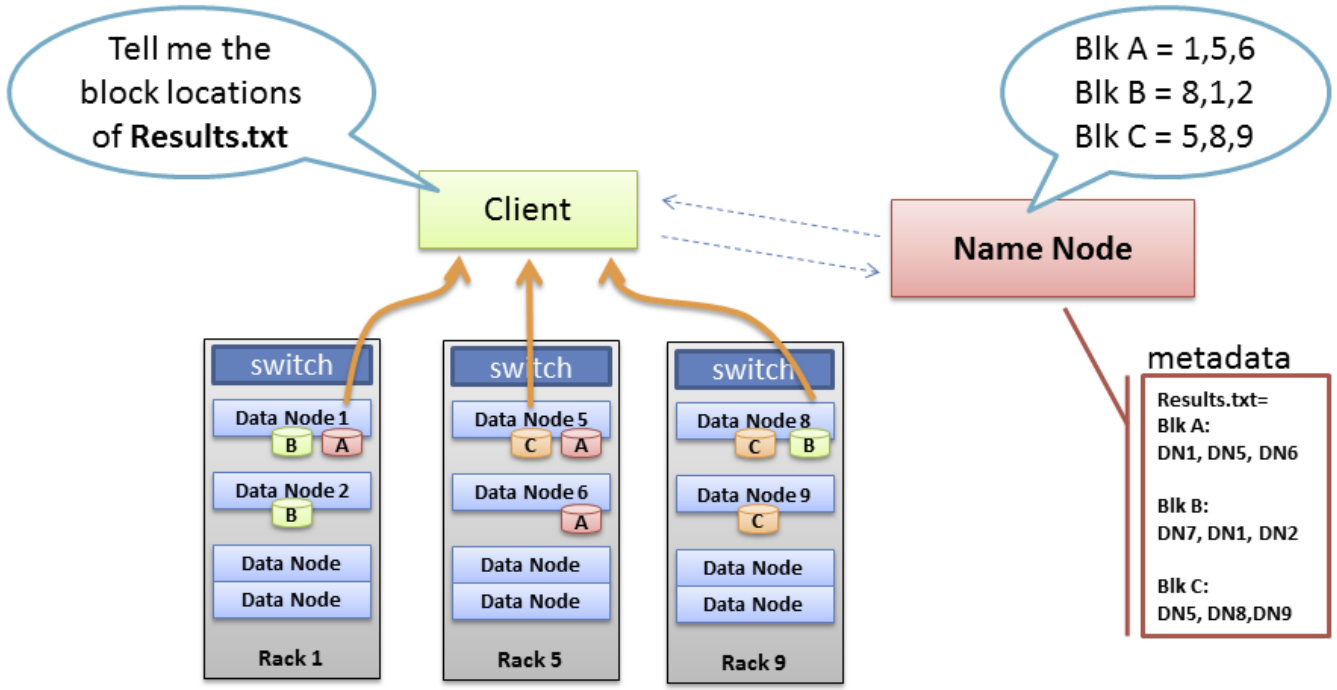
Secondary NameNode combines (checkpoints) the above information and delivers it back to the NameNode whilst maintaining a copy for itself

If the NameNode dies, the Secondary NameNode can be used to recover.



In a busy cluster, an admin may set the housekeeping to be much more frequent.

CLIENT READING FILES FROM HDFS



To read files, a Client consults the NameNode and asks for the block locations of a file.

The NameNode returns a list of each DataNode holding a block, for each block.

The Client picks a DataNode from each block list, and reads one block at a time with TCP.

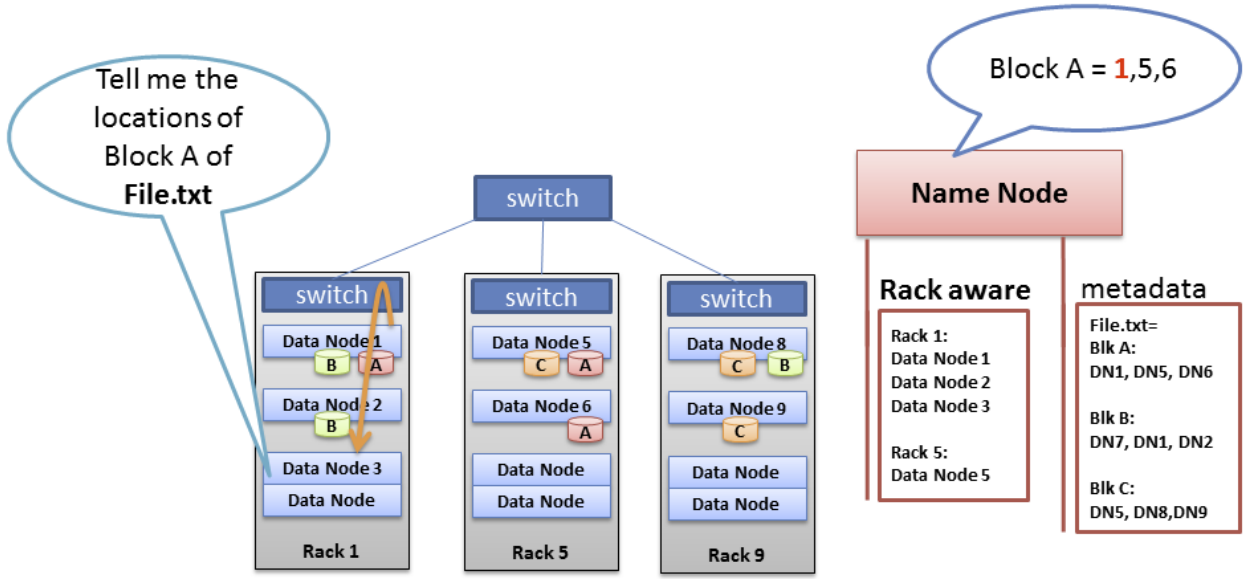
It does not progress to the next block until the previous block completes.

DATANODE READING FILES FROM HDFS

In some cases, a DataNode itself may need to read a block of data from HDFS.

This could be when it has been asked to process data that it does not have locally.

The NameNode's Rack Awareness can again provide optimal network behaviour in this case (encouraging intra-rack network traffic)



IN SUMMARY

Hardware Failure
Detection of faults and quick automatic recover is a core architectural goal of HDFS.

Portability
HDFS has been designed to be easily portable from one platform to another.

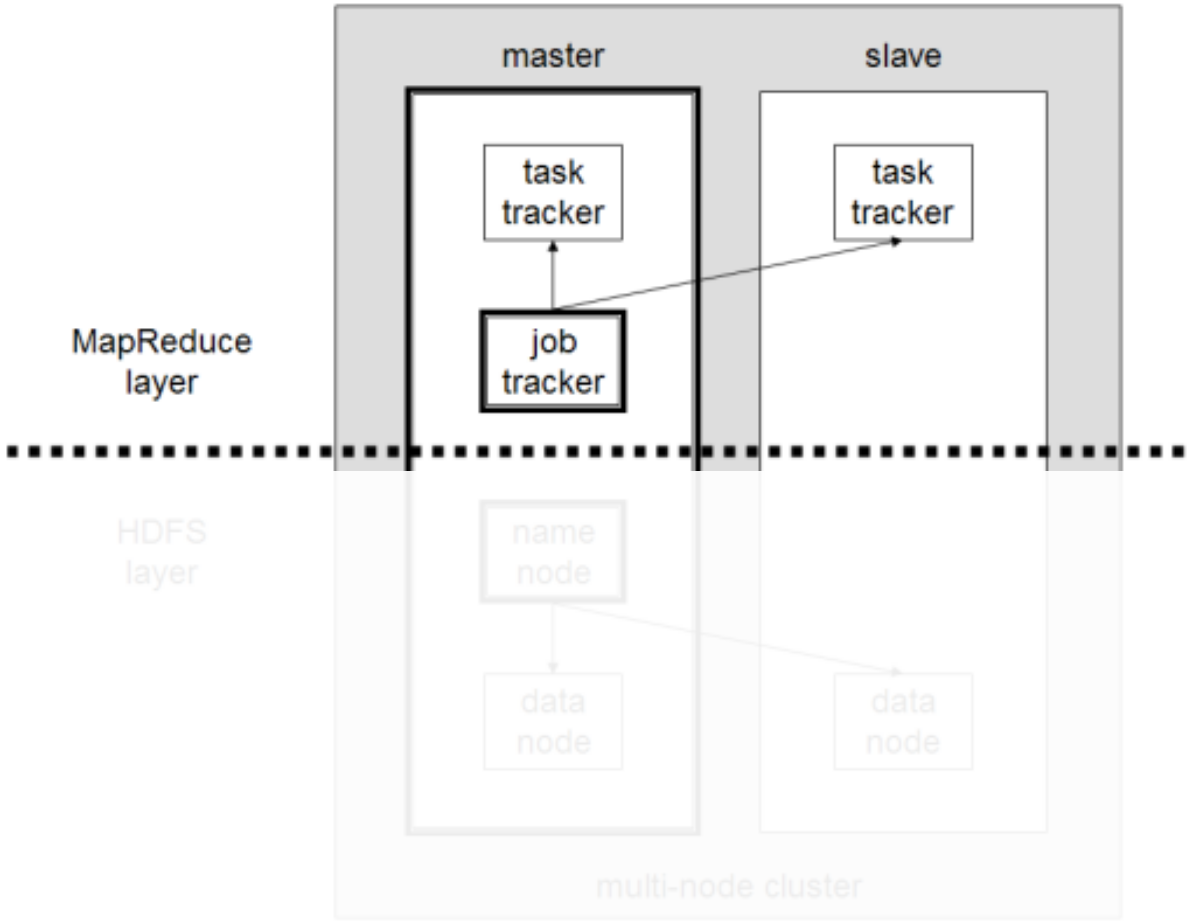
Large Data Sets
Typical files should be gigabytes, terabytes, or petabytes in size, and so HDFS is tuned to support large files.
It can support tens of millions of files in a single instance.

Streaming Data Access
HDFS is designed more for batch processing than interactive use by users.
Emphasis is on high throughput of data rather than low latency of data access.

Simple Coherency Model
Files are typically created and written once, and need not be changed.
This assumption simplifies data coherency issues. There is a plan to support appending-writes to files in the future.

Compute should move to Data
Computation is much more efficient if executed near the data it operates on, especially when the data size is huge.
HDFS provides interfaces for apps to move themselves closer to the data.

RECAP: HADOOP



Typical workflow

Load data into the cluster
(HDFS writes)

Analyse the data
(Map Reduce)

Store results in the cluster
(HDFS writes)

Read the results from the cluster
(HDFS reads)

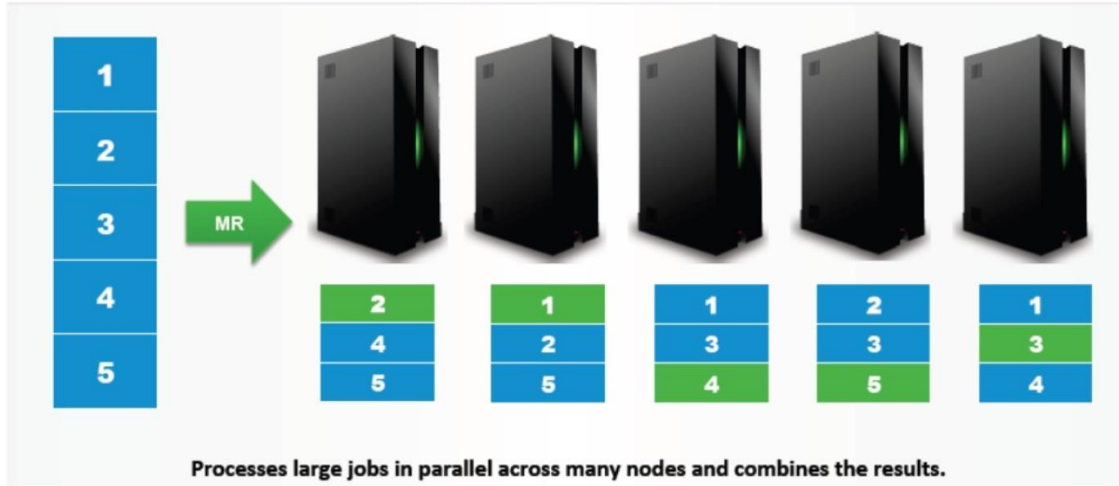
MapReduce

HDFS AND MAPREDUCE

HDFS

HDFS breaks incoming files into blocks and stores them redundantly across the cluster.

MapReduce



Processes large jobs in parallel across many nodes and combines the results.

WHAT IS MAPREDUCE?

Programming model for processing large data sets

running on commodity hardware in a reliable, fault-tolerant manner.

Originally announced by Google in Dec. 2004.

Hadoop contains a very popular open-source MapReduce implementation.

Basic concepts of MapReduce

Divide a big problem into sub-problems

Perform the same function on all sub-problems

Combine the output from all the sub-problems

MAPREDUCE WORKFLOW IN PRACTICE

Input splits

Input is divided into fixed-size pieces. These are each consumed by a single map.

Mapping

Data from each split is passed to a function to produce output <key, value> pairs.

(Sorting and Partitioning and) **Shuffling**

Consolidates the Mapping phase outputs – the same keys are grouped together.

Reduce

Outputs from the shuffle are aggregated (same key) and a single value returned.

MAP AND REDUCE

Map

Map is a **stateless** function which is used on a set of input values to calculate a set of key/value pairs

Reduce

Reduce is a **stateless** function which takes the set of these results and applies another function to them

In other words....

Map

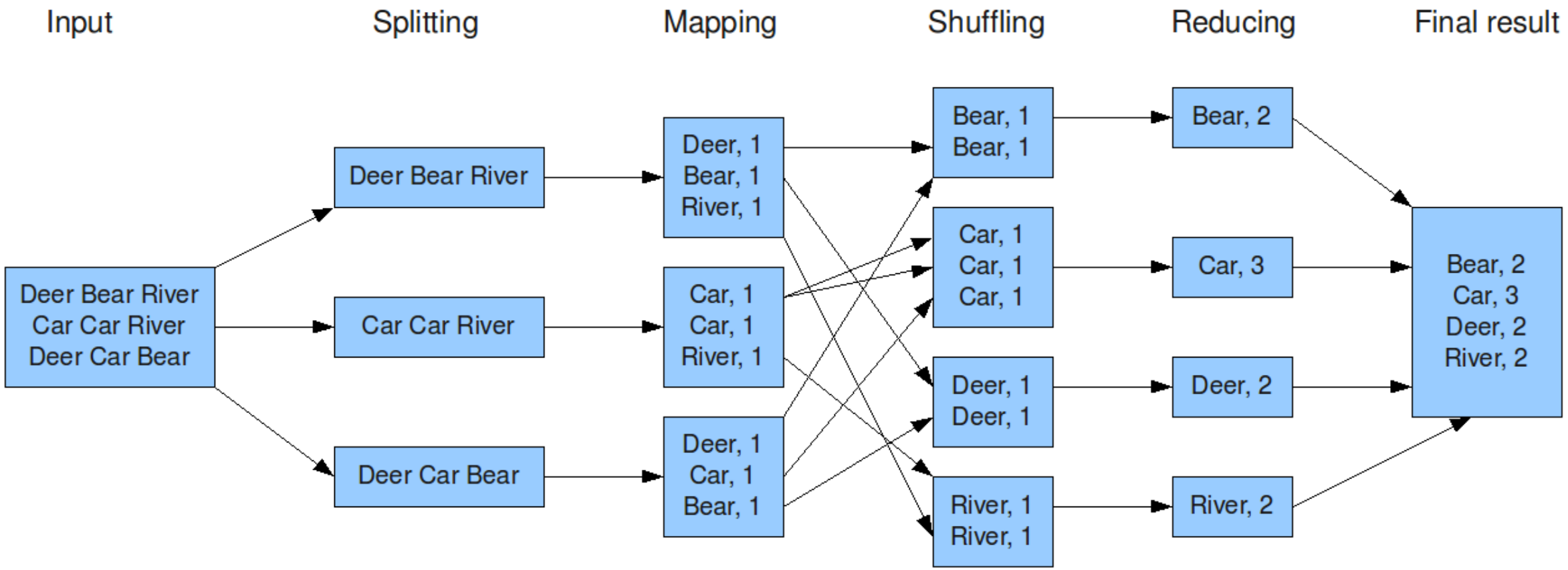
Transforms a set of data into key value pairs

Reduce

Aggregates the data into a scalar (**variable**). A reducer receives all the data for a individual "key" from all the mappers

This approach assumes no dependencies between the input data

SIMPLE WORD COUNTING EXAMPLE



MAPREDUCE CHARACTERISTICS

The outputs of Map and inputs/outputs of Reduce **are always <key, value> pairs**

Great for problems where the "sub problems" are NOT interdependent (e.g. output of one mapper should NOT depend on another)

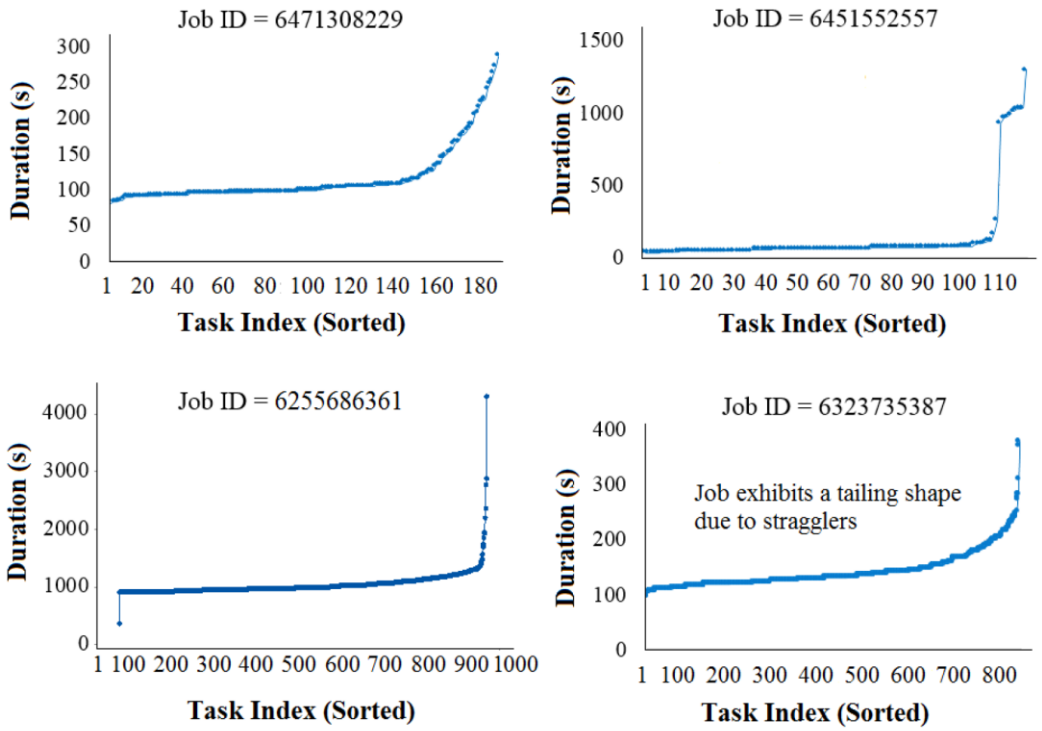
The reduce phase does not begin execution until ALL mappers have finished

MapReduce takes care of **scheduling tasks**, **monitoring them and re-executing failed tasks**.

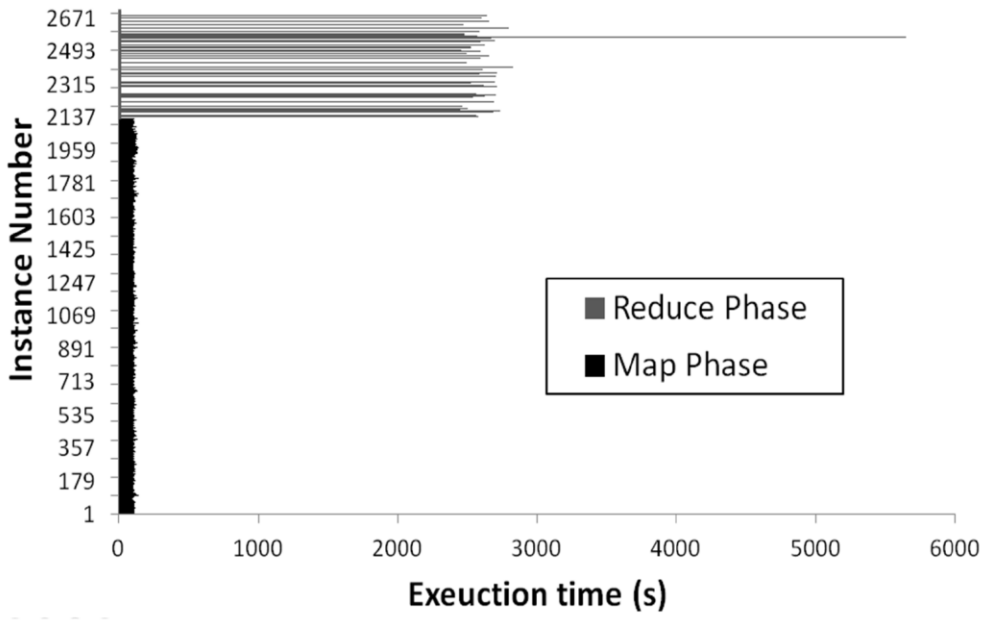
All data elements in MapReduce are **immutable** – they cannot be updated.

In Hadoop, MapReduce is Rack and HDFS aware

THE LONG-TAIL PROBLEM



Task duration pattern for jobs exhibiting stragglers in the Google cluster



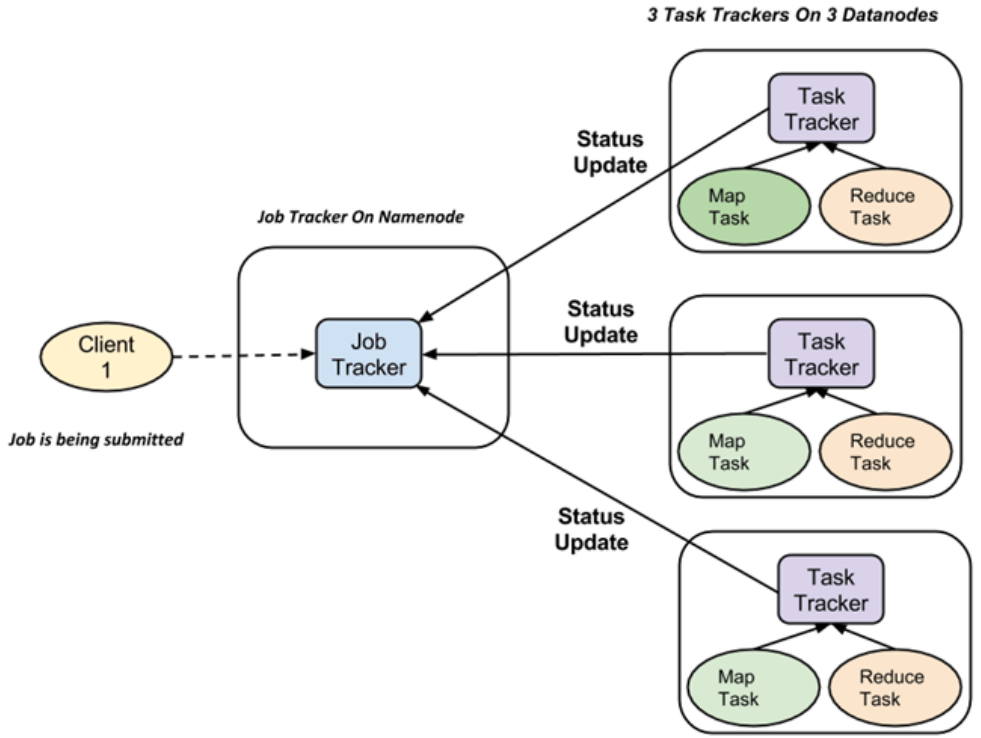
Task duration pattern for an example MapReduce job with a Reduce straggler

MAPREDUCE IN HADOOP (1)

Client submits MapReduce job to the **Job Tracker**

Job Tracker checks **NameNode** to learn which **DataNodes** have the file blocks (HDFS)

Job Tracker then provides the **Task Tracker** running on those nodes with the **Java code** required to execute the Map computation on their local data



Task Tracker starts a Map task and monitors its progress. Provides heartbeats & task status back to **Job Tracker**

As each Map task completes, its node stores the result in temporary storage (**intermediate data**). When all Maps complete, this is sent over the network to nodes running reduce tasks.

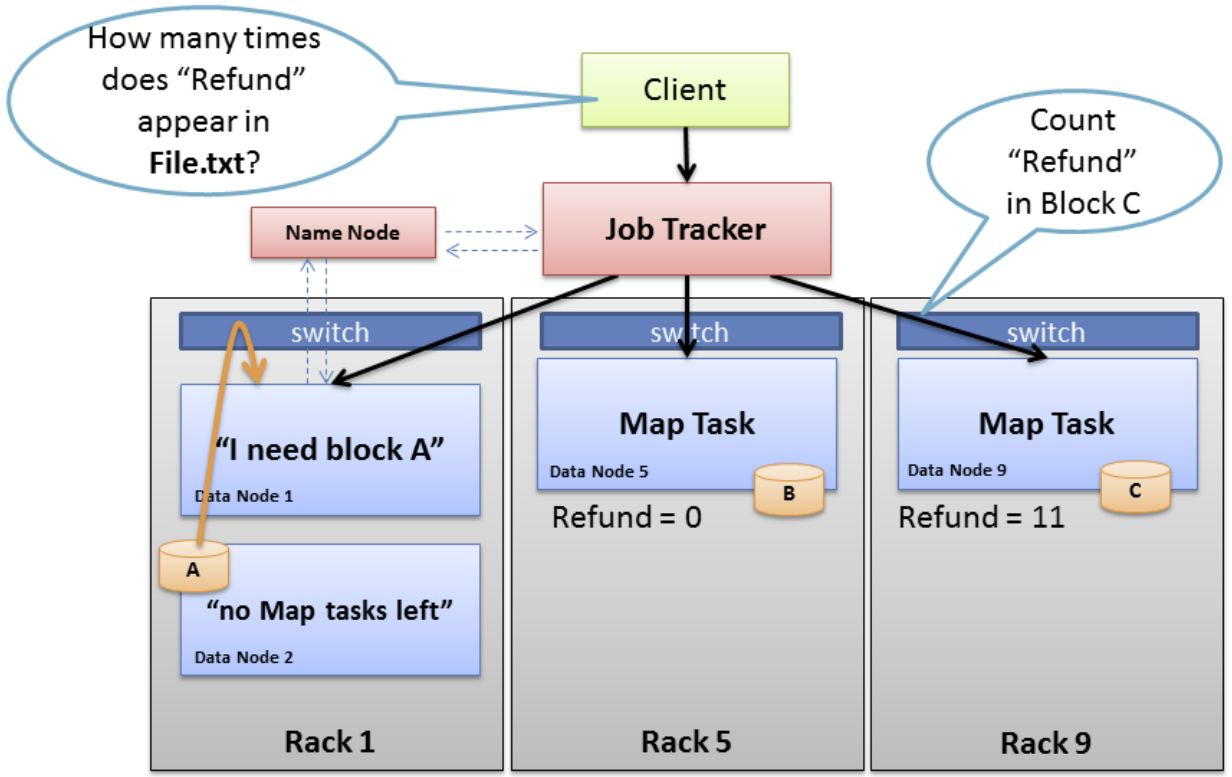
MAPREDUCE IN HADOOP (2)

The **Job Tracker** will always try to pick nodes with local data for a Map task.

This might not be possible (nodes with local data might be running too many other tasks, etc.)

In this situation, the **Job Tracker** will try to assign the task to a node in the same rack as the data (using Rack Awareness) wherever possible.

The **NameNode** will instruct the node to copy the data from the relevant **DataNode**.

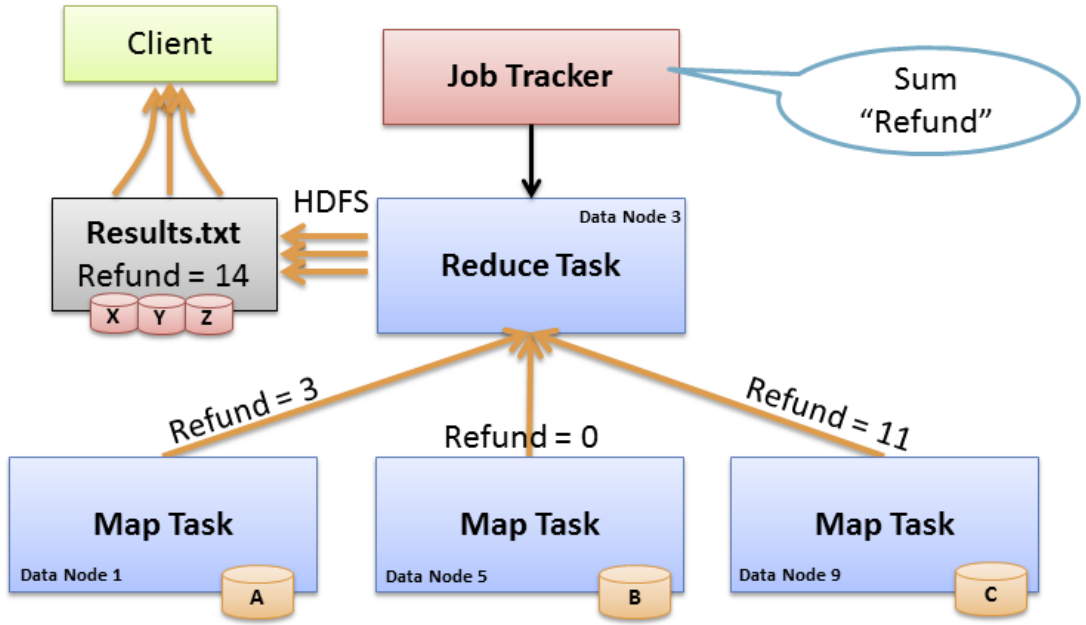


MAPREDUCE IN HADOOP (3)

Job Tracker starts a *Reduce* task on any of the nodes

It instructs the task to copy **intermediate data** from every completed *Map* task needed

If *Map* tasks respond quickly and flood the *Reduce* task with TCP data, "**Incast**" or "**fan-in**" occurs



To handle this, the cluster network switches need to have **good internal traffic management capabilities and buffers**

The outputs from the Reducer tasks are written to HDFS (split into blocks, pipeline replication, etc.)

MAPREDUCE: AN EXAMPLE THAT ISN'T WORD COUNT

Facebook has a "friends in common" feature.

This list doesn't change frequently, so to improve performance we can use MapReduce to **calculate everyone's friends in common once per day and store that result**.

When someone visits a page, this stored result can then be displayed.

Assume friends are stored as: **Person -> [List of Friends]**

FINDING MUTUAL FRIENDS EXAMPLE (2)

```
Person -> [List of Friends]
```

```
A -> B C D
B -> A C D E
C -> A B D E
D -> A B C E
E -> B C D
```

Assume each line will be an input split to a mapper.

The mapper will output **<Friend Person, List of Friends>**
As an example, the first two lines will have the result:

```
(A B) -> B C D
(A C) -> B C D
(A D) -> B C D
```

```
(A B) -> A C D E
(B C) -> A C D E
(B D) -> A C D E
(B E) -> A C D E
```

FINDING MUTUAL FRIENDS EXAMPLE (3)

Person -> [List of Friends]

```
A -> B C D
B -> A C D E
C -> A B D E
D -> A B C E
E -> B C D
```

Map and Sort

```
(A B) -> (A C D E)
(A B) -> (B C D)
(A C) -> (A B D E)
(A C) -> (B C D)
(A D) -> (A B C E)
(A D) -> (B C D)
(B C) -> (A B D E)
(B C) -> (A C D E)
(B D) -> (A B C E)
(B D) -> (A C D E)
(B E) -> (A C D E)
(B E) -> (B C D)
(C D) -> (A B C E)
(C D) -> (A B D E)
(C E) -> (A B D E)
(C E) -> (B C D)
(D E) -> (A B C E)
(D E) -> (B C D)
```

We are assuming that the list on the right has already been **sorted** by the MapReduce framework, ready for **partitioning** and **shuffling**.

The reducer function will intersect lists of values and output the same key with this result.

FINDING MUTUAL FRIENDS EXAMPLE (4)

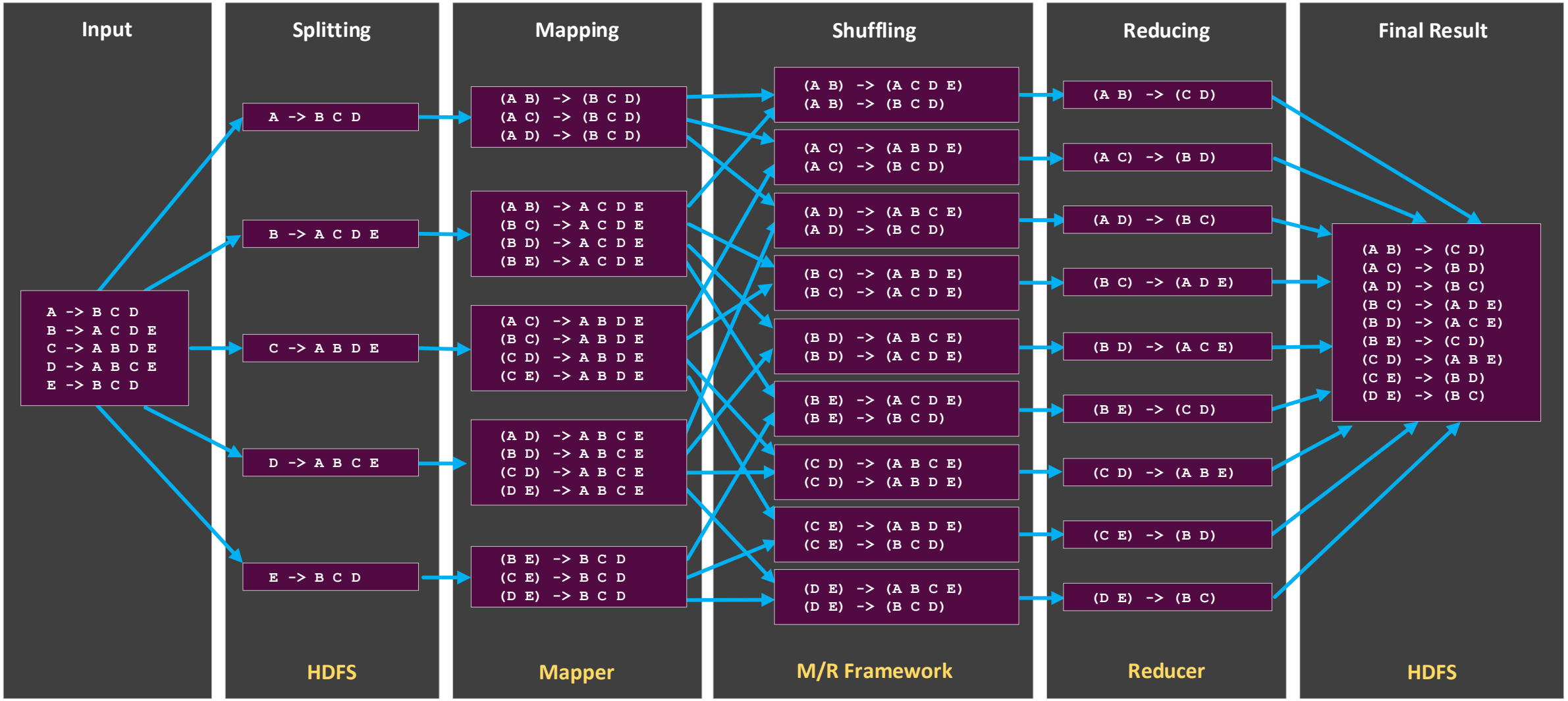
```
(A B) -> (A C D E)
(A B) -> (B C D)
(A C) -> (A B D E)
(A C) -> (B C D)
(A D) -> (A B C E)
(A D) -> (B C D)
(B C) -> (A B D E)
(B C) -> (A C D E)
(B D) -> (A B C E)
(B D) -> (A C D E)
(B E) -> (A C D E)
(B E) -> (B C D)
(C D) -> (A B C E)
(C D) -> (A B D E)
(C E) -> (A B D E)
(C E) -> (B C D)
(D E) -> (A B C E)
(D E) -> (B C D)
```

Partition, Shuffle, Reduce

```
(A B) -> (C D)
(A C) -> (B D)
(A D) -> (B C)
(B C) -> (A D E)
(B D) -> (A C E)
(B E) -> (C D)
(C D) -> (A B E)
(C E) -> (B D)
(D E) -> (B C)
```

Now - for example - when A visits B's profile, we can quickly look up **`(A B)`** and see that they have two friends in common, **`(C D)`**

FINDING MUTUAL FRIENDS EXAMPLE (5)

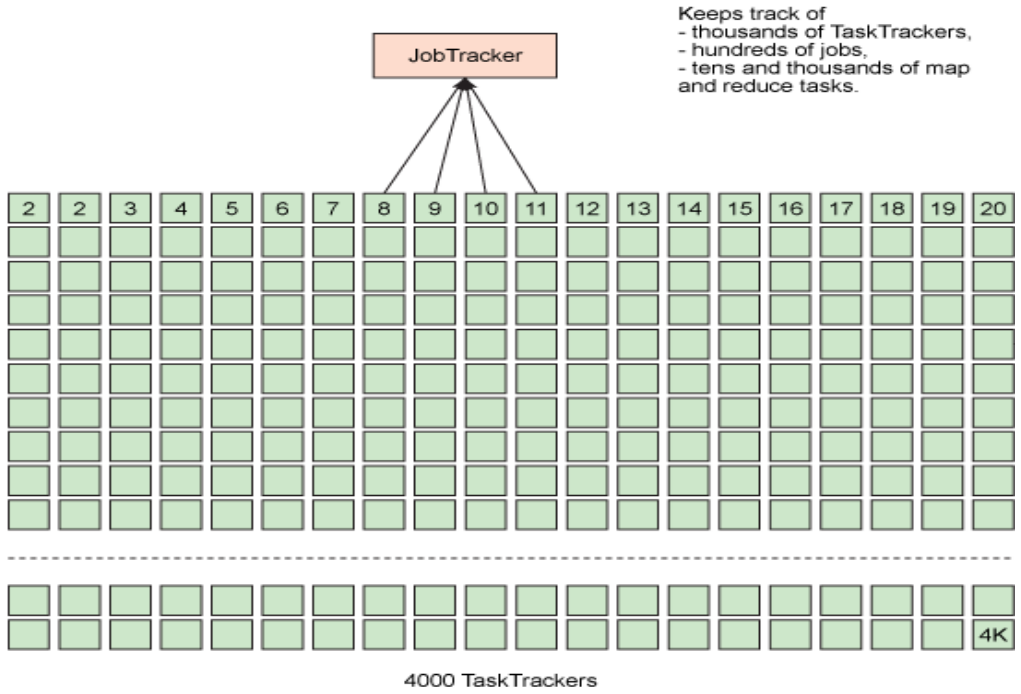


LIMITATIONS OF HADOOP V1

JobTracker is a single point of availability.
If JobTracker fails, all jobs must restart.

In Hadoop 1.0, there is tight coupling between Cluster Resource Management and MapReduce.

Yahoo estimated the limits of this design to be 5000 nodes and 40,000 concurrent tasks



There are also problems with resource utilization

Hadoop 1.0 has a pre-defined number of map and reduce slots.
Map slots might be 'full' while Reduce slots are 'empty' (and vice versa)

NameNodes hold all metadata in main memory, so typically limited to 50-100 million files per cluster.

LIMITATIONS OF HADOOP V1 (2)

There are also limitations in running non-MapReduce applications in Hadoop 1.0.

MapReduce works on batch-driven analysis.

But it is often desirable to run other computation paradigms in a Hadoop Cluster

Why run non-MapReduce applications?

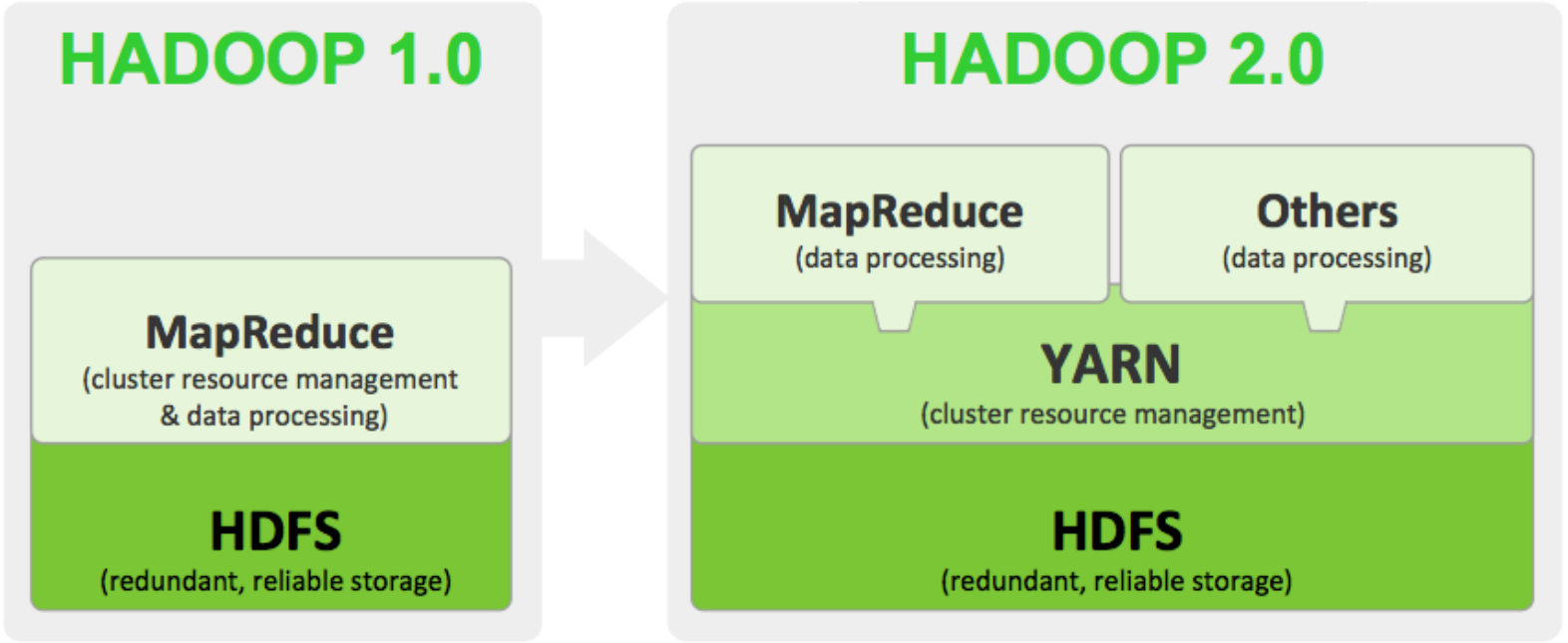
MapReduce is batch driven; other engines work much better when results are needed in real-time.

Message-passing approaches are not possible in MapReduce (no interdependencies).

These problems (and more) are addressed in Hadoop v2.0

Hadoop V2

HADOOP V2



Hadoop 2 moves from a restricted batch-oriented model to **more interactive and specialized processing models**

The biggest changes in Hadoop 2 are **HDFS Federation**, **YARN,** a highly available **NameNode**, and the concept of **Containers**

YARN

YET ANOTHER RESOURCE NEGOTIATOR (YARN)

The fundamental idea of **YARN** is to **split Hadoop resource management and job scheduling into separate processes (daemons).**

Different types of app can be submitted to YARN (MapReduce, Giraph, etc.) **An app is either single job or a Directed Acyclic Graph (DAG) of jobs.**

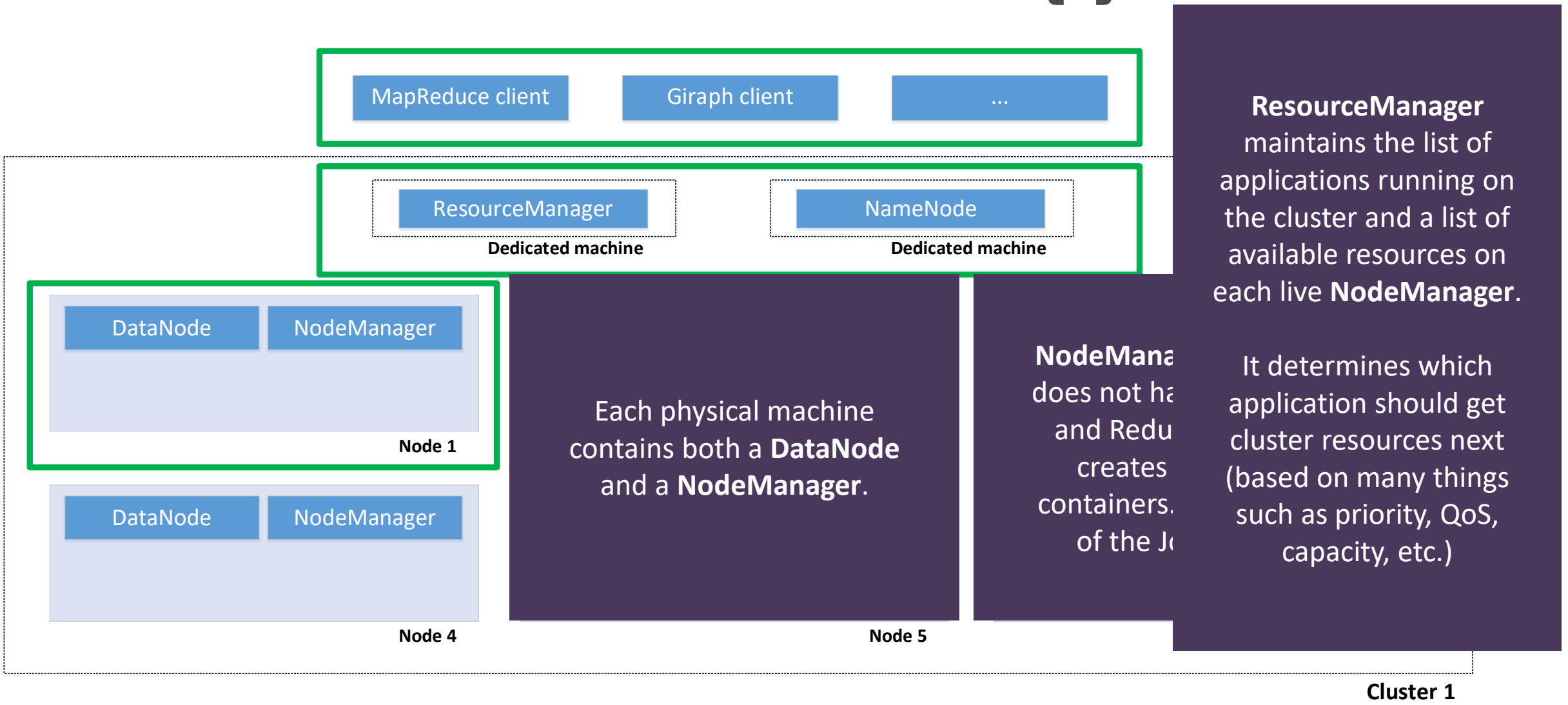
ResourceManager is the authority that arbitrates resources among all applications.
Replaces the **JobTracker**.

NodeManager is a per-machine framework responsible for **containers**, monitoring resource usage, and reporting to the **ResourceManager.** Each machine in a cluster is a NodeManager and a DataNode.

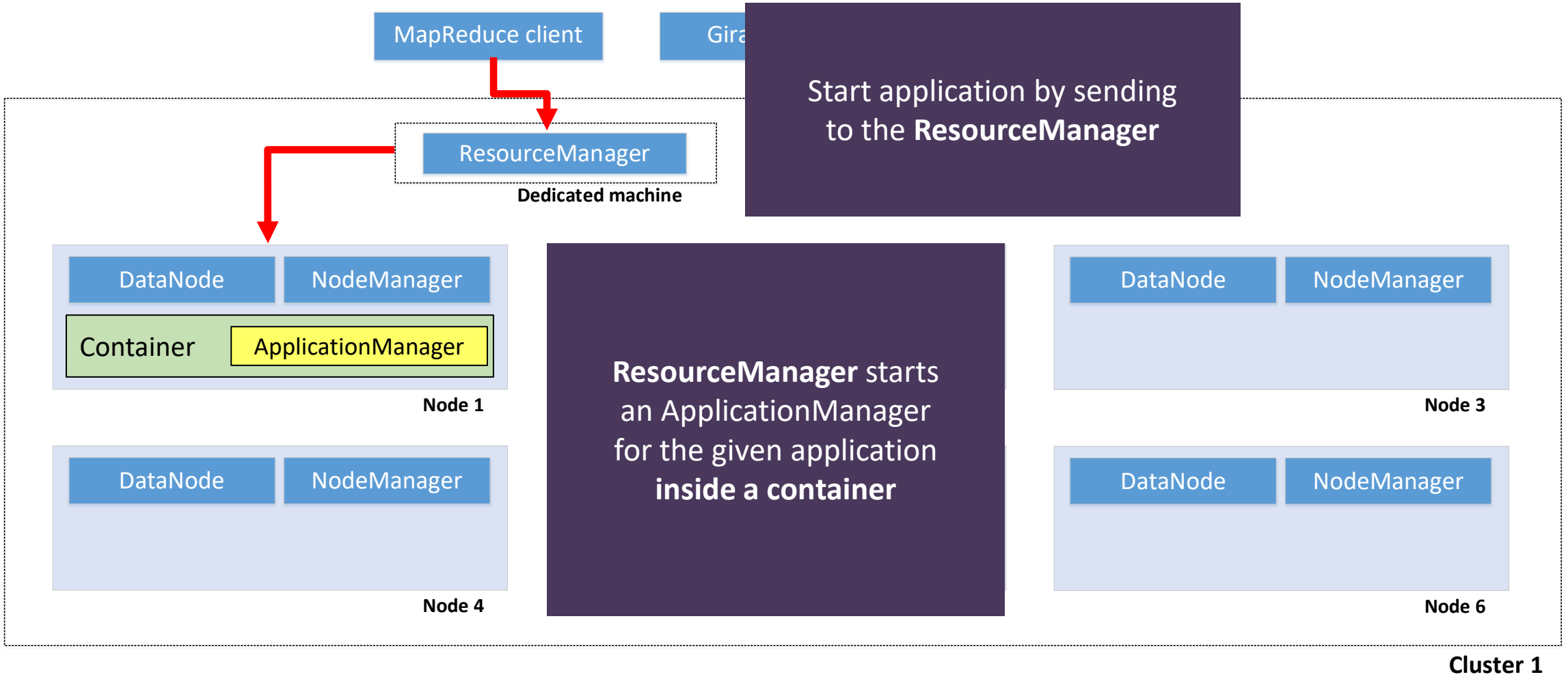
ApplicationMaster is tasked with negotiating resources from the **ResourceManager** and working with **NodeManager**s to execute & monitor tasks.

This allows more jobs to be run in parallel, and scalability is dramatically increased

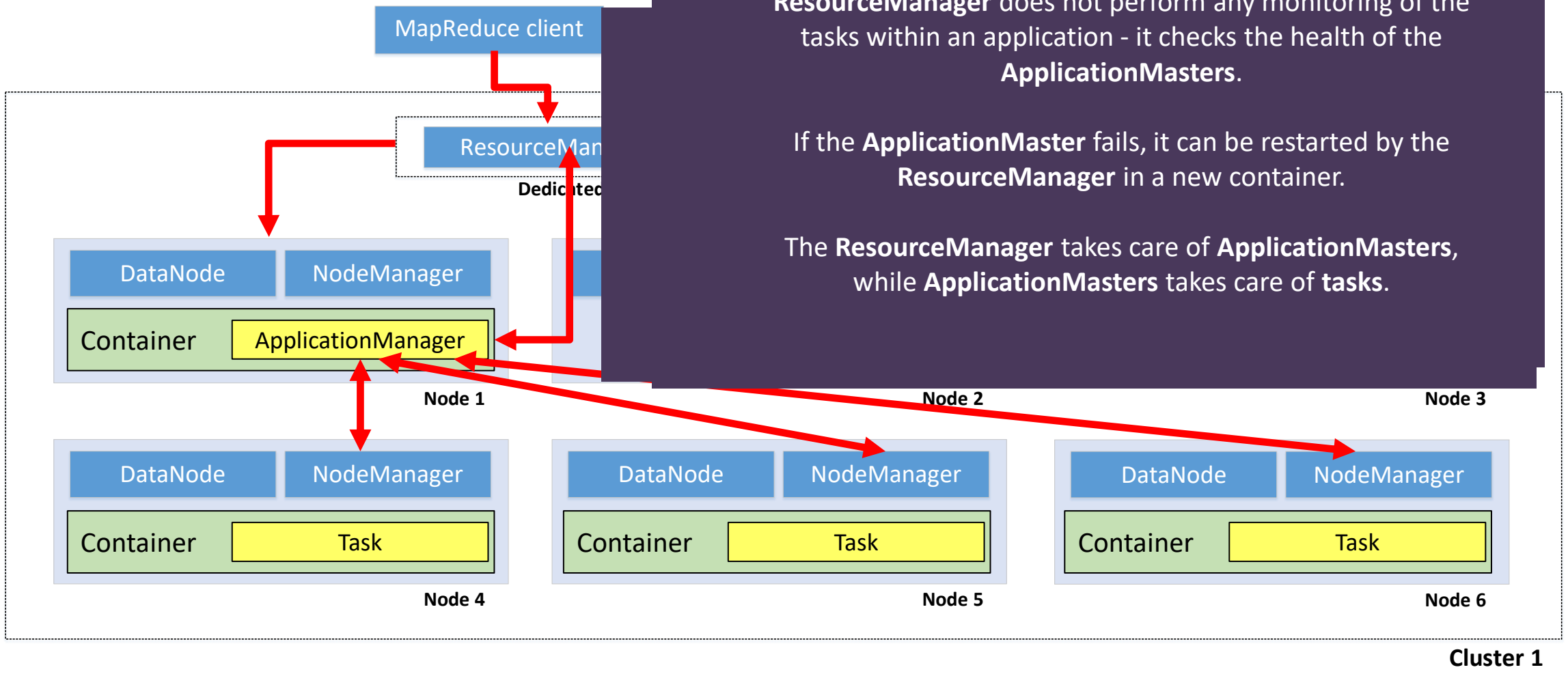
APPLICATION SUBMISSION IN YARN (1)



APPLICATION SUBMISSION IN YARN (2)



APPLICATION SUBMISSION IN YARN (3)



ResourceManager does not perform any monitoring of the tasks within an application - it checks the health of the **ApplicationMasters**.

If the **ApplicationMaster** fails, it can be restarted by the **ResourceManager** in a new container.

The **ResourceManager** takes care of **ApplicationMasters**, while **ApplicationMasters** takes care of **tasks**.

YARN CAN RUN ANY DISTRIBUTED APPLICATION

The **ResourceManager**, **NodeManager**, and **Container** are not concerned about the type of task or application they are to run.

Any application can run as long as an appropriate ApplicationMaster is implemented for it.

This has several obvious benefits

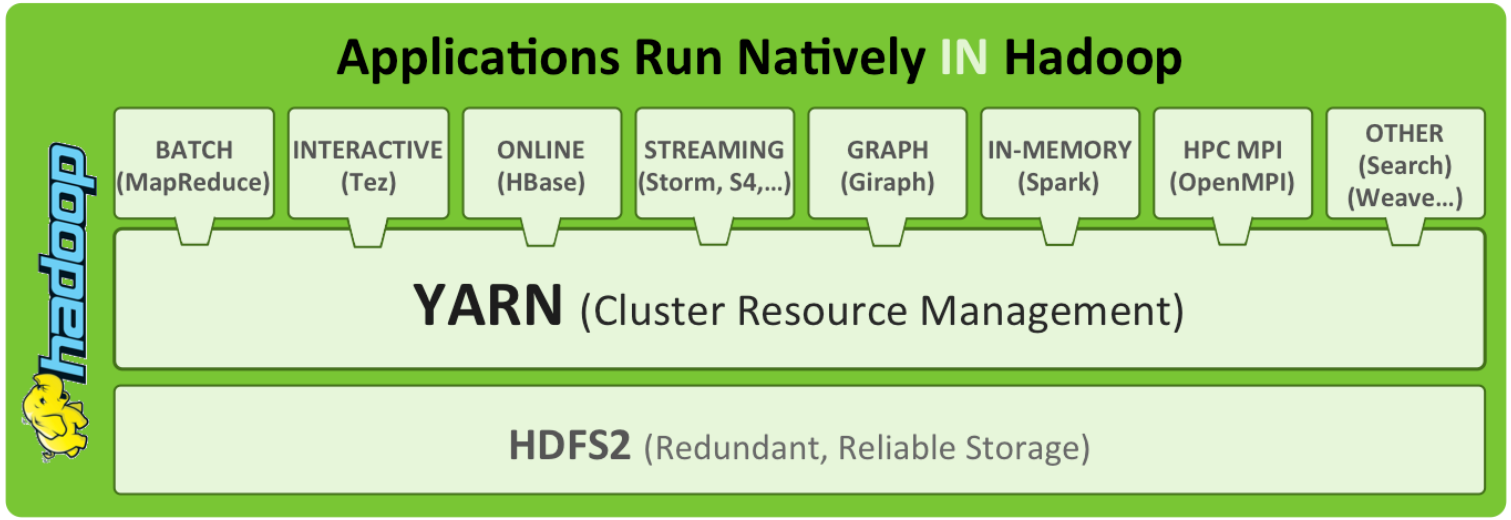
Higher cluster utilisation (resources not used by one framework can be used by another)

Lower operational costs (only one type of cluster needs to be managed and tuned)

Reduced data motion (no need to move data between YARN and other systems)

Managing a single cluster should also result in less data centre space used, which in turn leads to less operational cost, power spend, less cooling, etc.

HADOOP V2 IN SUMMARY



HDFS2 – High-availability and federated **NameNodes** for horizontal scalability

YARN – Moves beyond the batch processing of Hadoop 1 and improved efficiency

Whole eco-system of tools and applications

WHERE TO FIND THIS IN THE CLOUD?