Hadoop

Understanding Hadoop and its components

NoSQL: Lecture 6

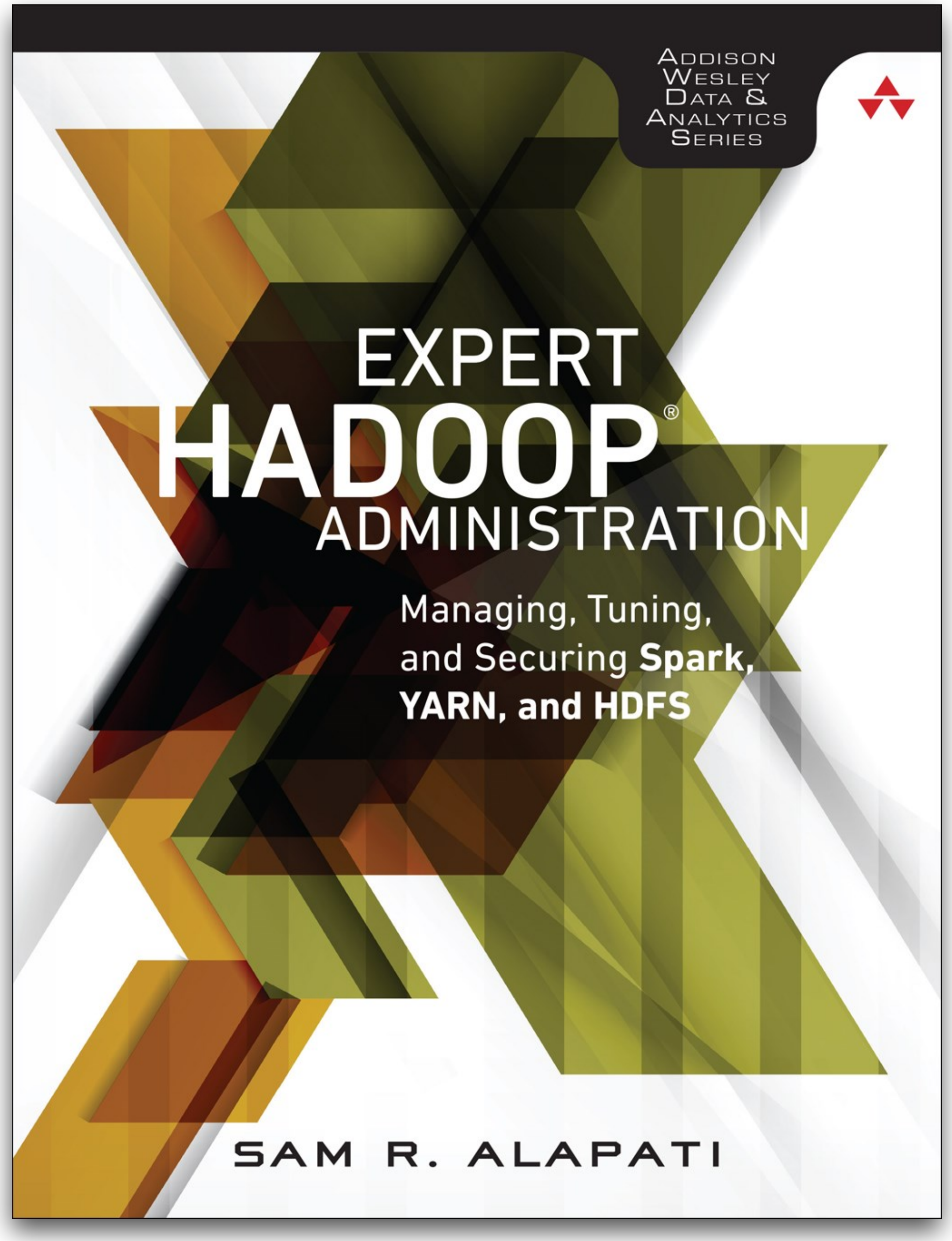
Piotr Fulmański

FACULTY OF MATHEMATICS
AND COMPUTER SCIENCE
University of Lodz

Expert Hadoop Administration: Managing, Tuning, and Securing Spark, YARN, and HDFS

by Sam R. Alapati

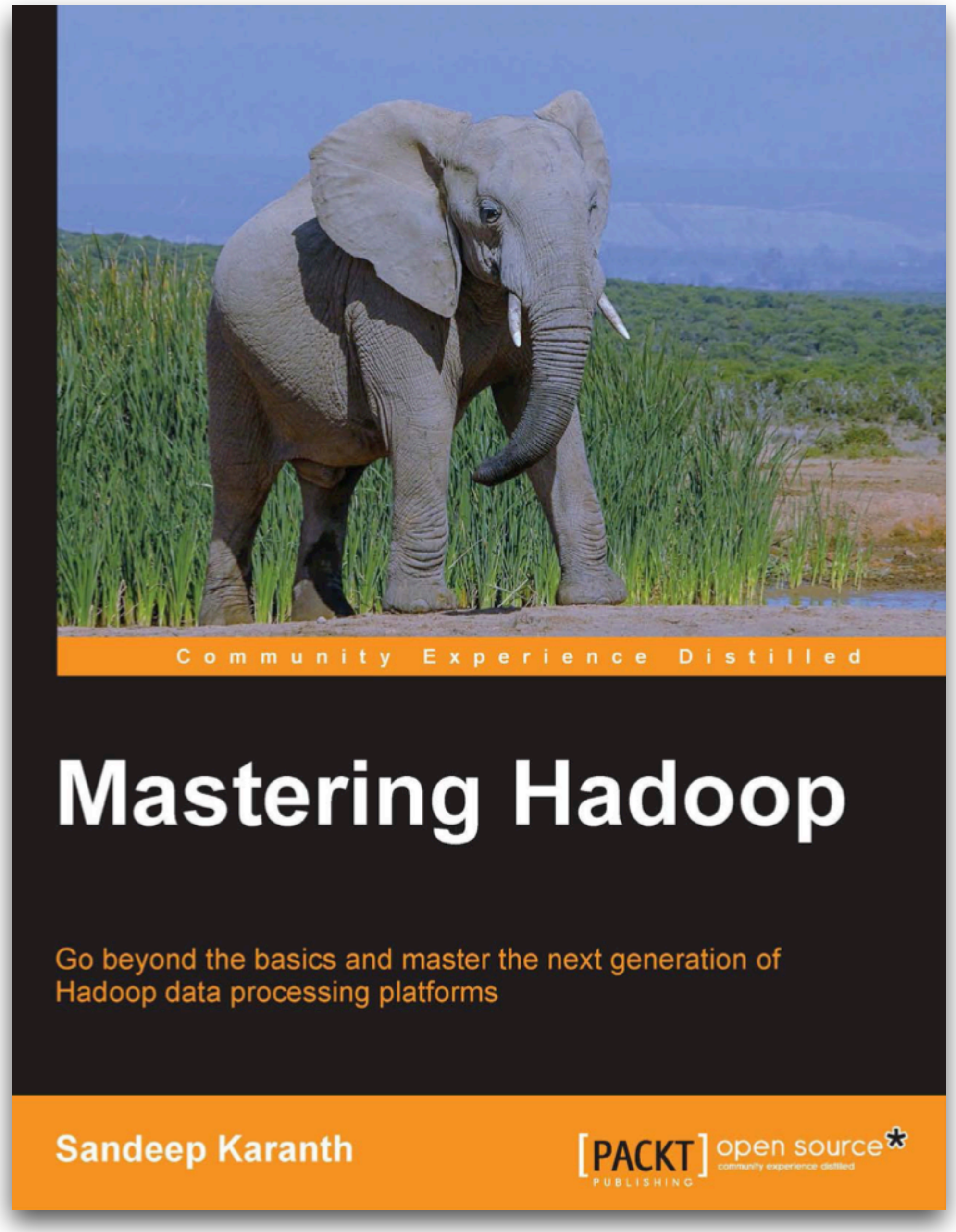
Addison-Wesley, 2016


Addison Wesley Data & Analytics Series
EXPERT HADOOP® ADMINISTRATION
Managing, Tuning, and Securing Spark, YARN, and HDFS
SAM R. ALAPATI

Mastering Hadoop

by Sandeep Karanth

Packt Publishing, 2014


Community Experience Distilled
Mastering Hadoop
Go beyond the basics and master the next generation of Hadoop data processing platforms
Sandeep Karanth
[PACKT] open source*
PUBLISHING
community experience distilled

Hadoop origins

Hadoop origins

The flood of Big Data

Hadoop origins

Big Data and its challenges

In short, Big Data refers to the massive amount of data that cannot be stored, processed, and analyzed using traditional ways.

The main elements of Big Data are:

- Volume - There is a massive amount of data generated every second;
- Velocity - The speed at which data is generated, collected, and analyzed;
- Variety - The different types of data: structured, semi-structured, unstructured.

Components of Hadoop

Hadoop is a framework that uses distributed storage and parallel processing to store and manage Big Data. There are three components of Hadoop.

- Hadoop HDFS - Hadoop Distributed File System (HDFS) is the storage component of Hadoop. It is used to store data across machines.
- Hadoop MapReduce - Hadoop MapReduce is the processing component of Hadoop. It is responsible for performing distributed processing.
- Hadoop YARN - Hadoop YARN is a resource management unit of Hadoop.
- Zookeeper is used to ensure synchronization across a cluster.

Hadoop HDFS

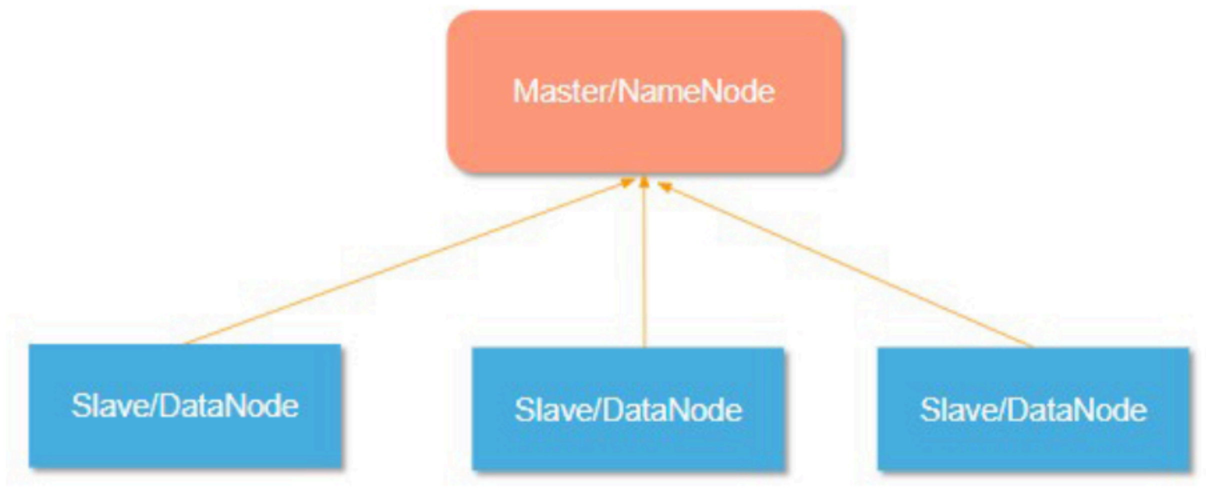
Hadoop HDFS

Features of HDFS

Features of HDFS

- Provides distributed storage
- Can be implemented on commodity hardware
- Provides data security
- Highly fault-tolerant - if one machine goes down, the data from that machine goes to the next machine

There are two components of HDFS - **NameNode** and **DataNode**. While usually there is only one name node at a time, there can be multiple data nodes. The name node is called the **master**, and the data nodes are called the **slaves**. The name node is responsible for the workings of the data nodes. It also stores the metadata.



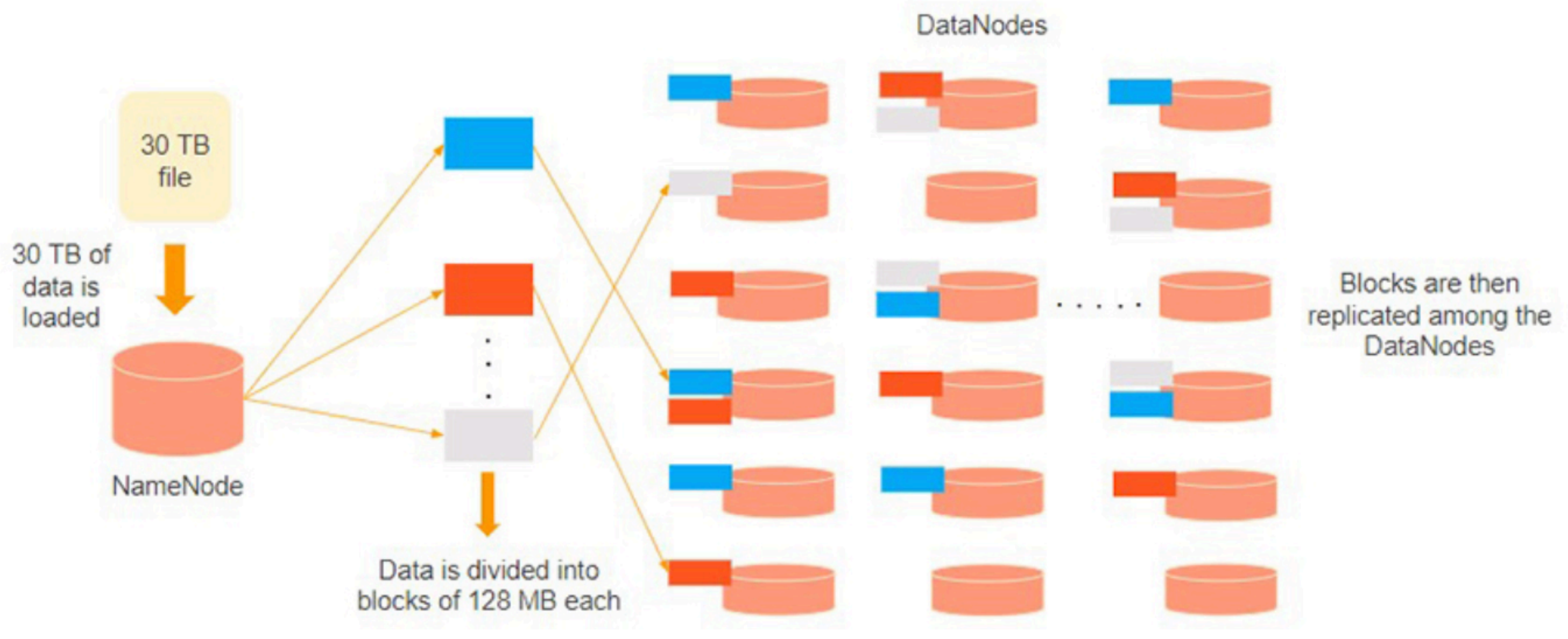
Name node may be backed up by **Secondary NameNode** which main purpose is to create a new NameNode in case of failure. Hadoop 3 enables having multiple name nodes.

Hadoop HDFS
HDFS cluster

Master and slave nodes form the **HDFS cluster**.

The data nodes read, write, process, and replicate the data. Replication of the data is performed three times by default. The data nodes also send signals, known as heartbeats, to the name node. These heartbeats show the status of the data node. By default, the heartbeat interval is 3 seconds. If the NameNode doesn’t get any heartbeat, it waits for 10 minutes and considers the DataNode as dead.

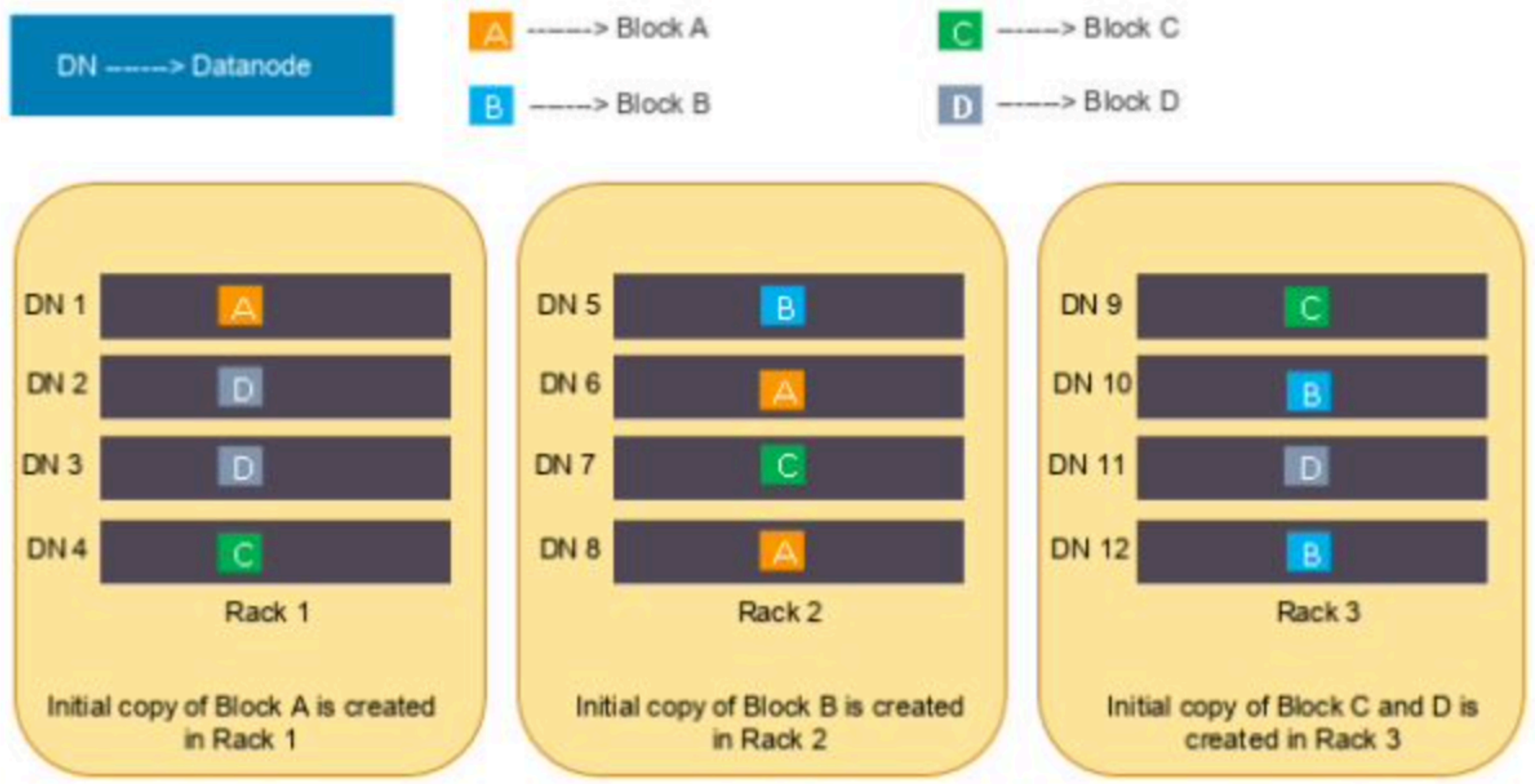
To read or write a file in HDFS, a client must interact with the NameNode. The NameNode checks the privileges of the client and gives permission to read or write on the data blocks.


30 TB file
30 TB of data is loaded
NameNode
Data is divided into blocks of 128 MB each
DataNodes
Blocks are then replicated among the DataNodes

Hadoop HDFS

HDFS in Hadoop Architecture divides large data into blocks. Each block is replicated three times by default, contains 128 MB of data. Replications operate under two rules:

- Two identical blocks cannot be placed on the same DataNode.
- When a cluster is rack aware, all the replicas of a block cannot be placed on the same rack.

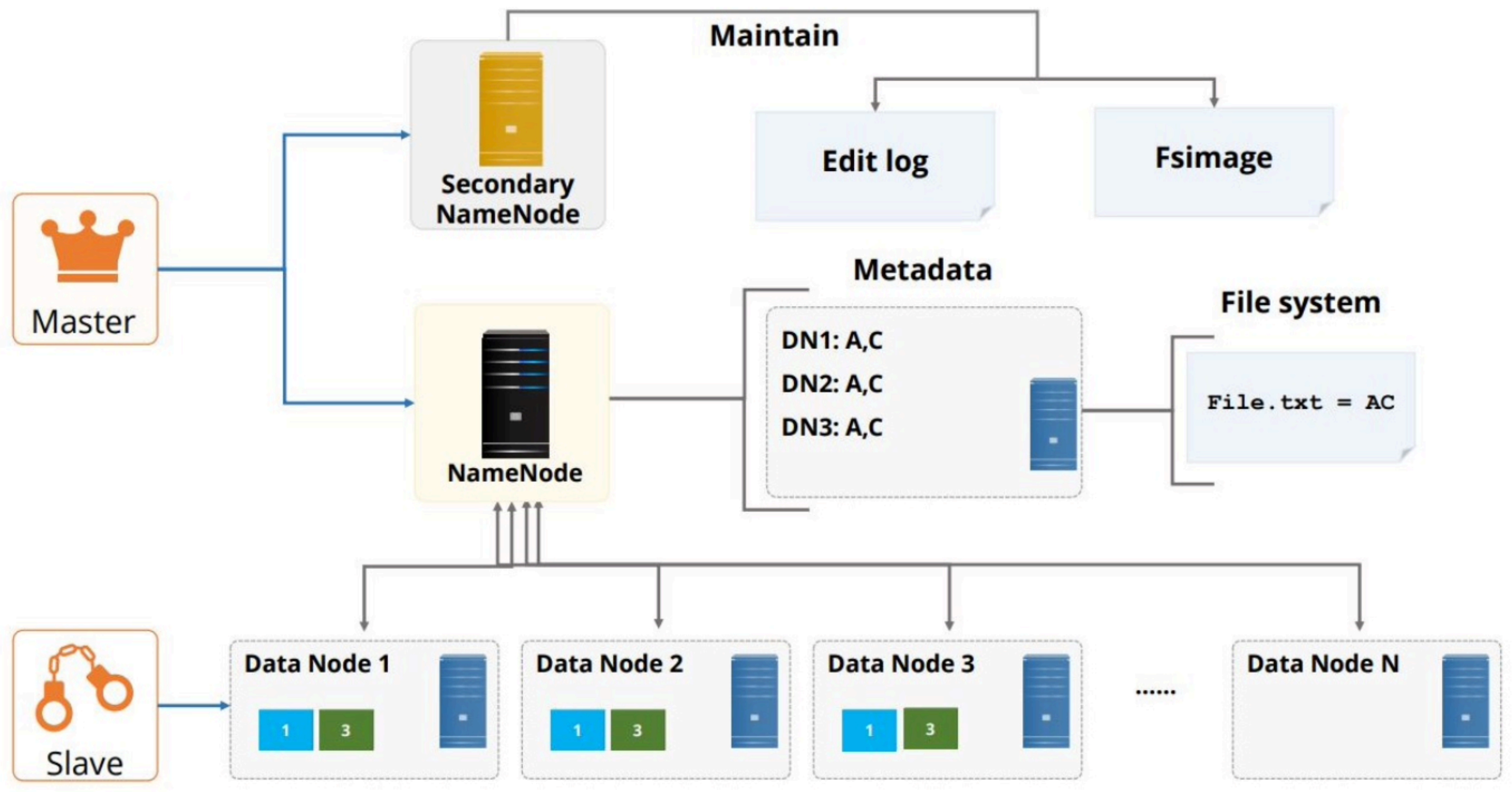


Hadoop HDFS

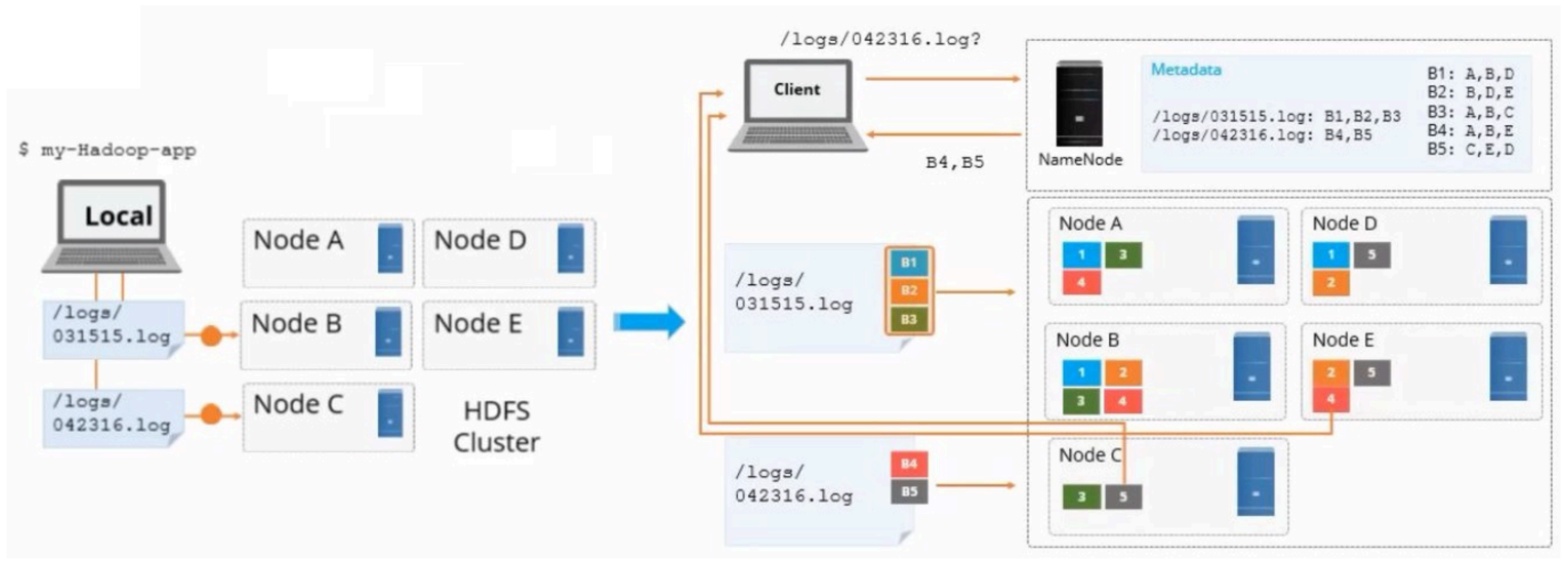
- HDFS has high fault-tolerance.
- HDFS may consist of thousands of server machines. Each machine stores a part of the file system data. HDFS detects faults that can occur on any of the machines and recovers it quickly and automatically.
- HDFS is designed in such a way that it can be built on commodity hardware and heterogeneous platforms, which is low-priced and easily available.
- HDFS supports large datasets in batch-style jobs.
- HDFS has high throughput. Low latency is considered as less important.
- HDFS exposes a file system namespace and allows user data to be stored in files. HDFS has a hierarchical file system with directories and files.

Hadoop HDFS

HDFS architecture is known as the master and slave architecture



Hadoop HDFS

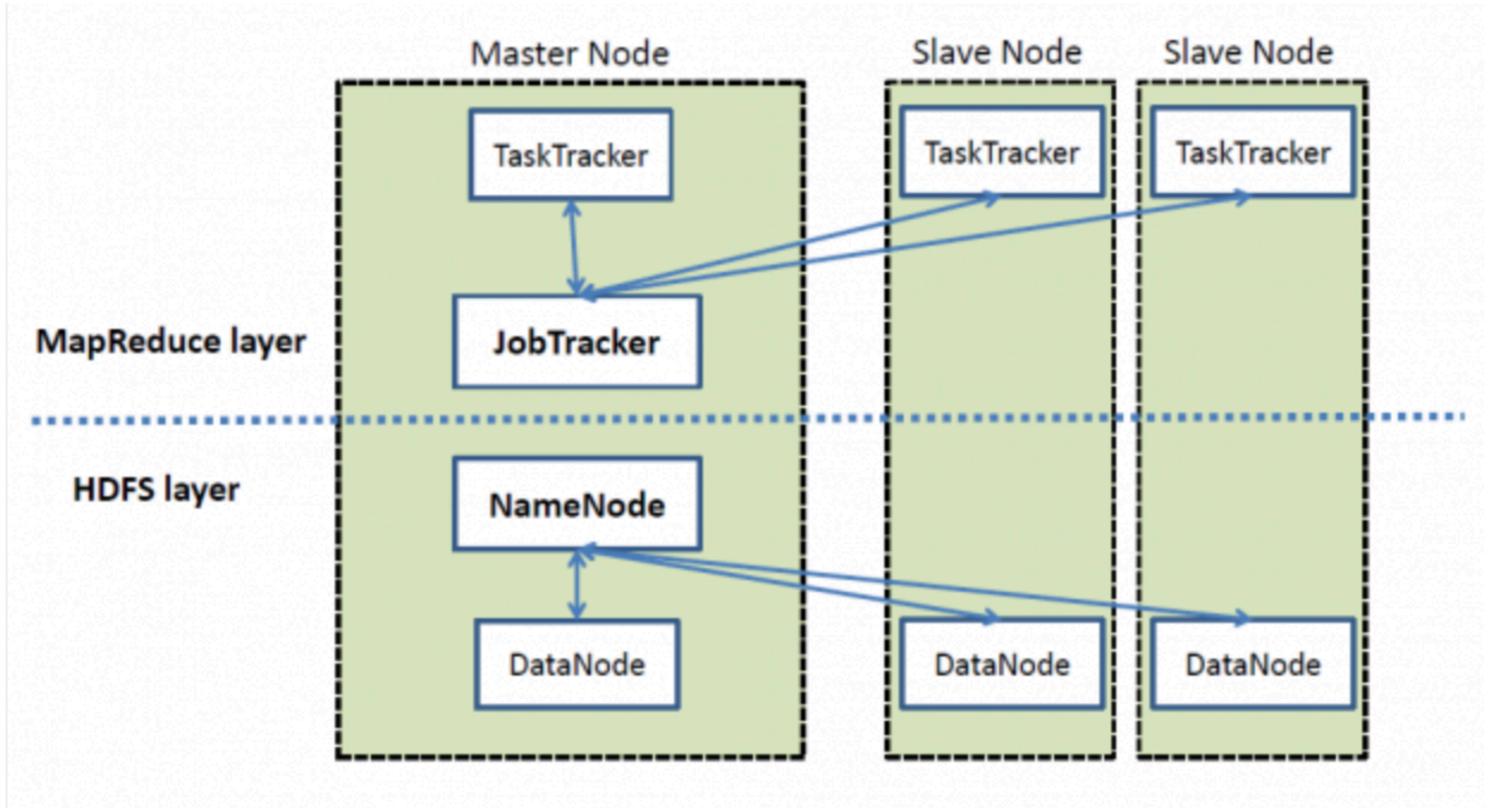


$ my-Hadoop-app
Local
/logs/ 031515.log
/logs/ 042316.log
Node A
Node D
Node B
Node E
Node C
HDFS Cluster
/logs/042316.log?
Client
B4,B5
NameNode
Metadata
/logs/031515.log: B1,B2,B3
/logs/042316.log: B4,B5
B1: A,B,D
B2: B,D,E
B3: A,B,C
B4: A,B,E
B5: C,E,D
/logs/ 031515.log
B1
B2
B3
/logs/ 042316.log
B4
B5

Hadoop HDFS

HDFS Layer and MapRecuce Layer components

It's worth to note that NameNode and DataNode function on HDFS Layer. There are corresponding components function on MapReduce Layer: *JobTracker* and *TaskTracker* respectively.



Hadoop HDFS

HDFS Layer and MapRecuce Layer components

JobTracker is a service in Hadoop which guides the MapReduce task to point to specific data node that has desired data required to process through MapReduce task.

The following workflow is taking a place:

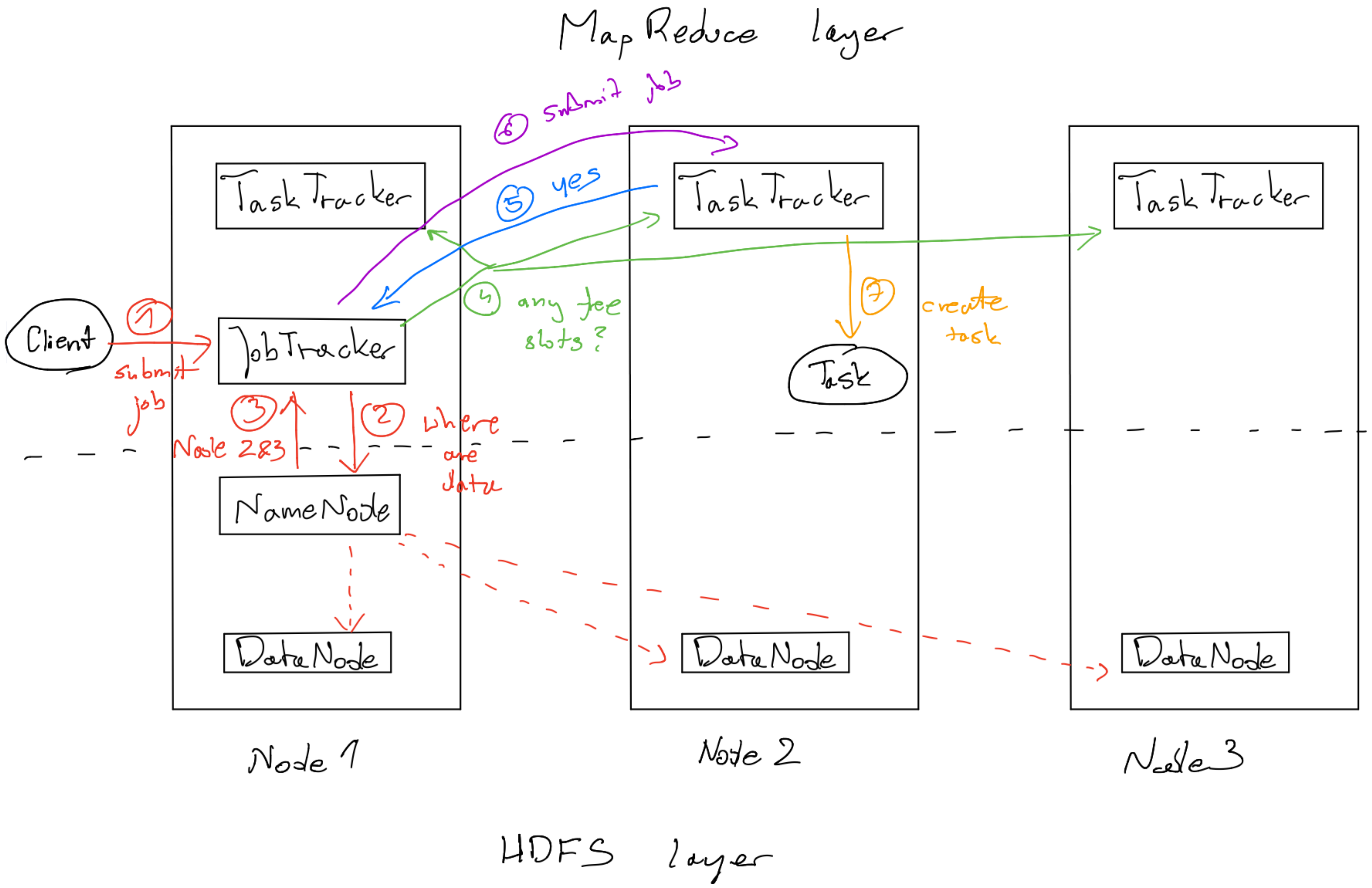
1. Client applications submit jobs to the JobTracker.
2. The JobTracker talks to the NameNode to determine the location of the data.
3. The JobTracker locates TaskTracker nodes with available slots at or near the data.
4. The JobTracker submits the work to the chosen TaskTracker nodes.
5. The TaskTracker nodes are monitored.
6. When the work is completed, the JobTracker updates its status.
7. Client applications can poll the JobTracker for information.

The JobTracker is a point of failure for the Hadoop MapReduce service. If it goes down, all running jobs are halted.

JobTracker and TaskTrackers were replaced by YARN in further Hadoop releases.

Hadoop HDFS

HDFS Layer and MapReduce Layer components



Hadoop HDFS

HDFS Snapshots

HDFS Snapshots are read-only point-in-time copies of the file system. Snapshots can be taken on a subtree of the file system or the entire file system.

The following features of HDFS Snapshots make it very efficient:

- Snapshot creation is instantaneous: the cost is $O(1)$ excluding the inode lookup time.
- Additional memory is used only when modifications are made relative to a snapshot: memory usage is $O(M)$, where M is the number of modified files/directories.
- Blocks in datanodes are not copied: the snapshot files record the block list and the file size. There is no data copying.
- Snapshots do not adversely affect regular HDFS operations: modifications are recorded in reverse chronological order so that the current data can be accessed directly. The snapshot data is computed by subtracting the modifications from the current data.
- Snapshots can be taken on any directory once the directory has been set as *snapshottable*. A snapshottable directory is able to accommodate 65,536 simultaneous snapshots. There is no limit on the number of snapshottable directories. Administrators may set any directory to be snapshottable. If there are snapshots in a snapshottable directory, the directory can be neither deleted nor renamed before all the snapshots are deleted.
- Nested snapshottable directories are currently not allowed. In other words, a directory cannot be set to snapshottable if one of its ancestors/descendants is a snapshottable directory.

For more information see [HHS001].

Hadoop HDFS

Authentication vs. Authorization

Authentication

Authentication is asserting and proving one's identity. I'm XYZ. My identification is "user135" (userID) and I can prove I'm XYZ because I know XYZ's password (and of course no one else knows).

Authentication is the process of validating a user's identity to **grant them access to a system or network**. It **determines the right of a user to access resources** (understood generally as a system) such as services, data servers, networks, databases, files, etc.

Authorization

Once a user is authenticated, the application knows who you are. **Authorization then is determining what that user can do within the application and what data do they have access to.**

Authorization is the process of **giving necessary privileges to the user to access specific resources** such as files, databases, locations, funds, files, information, almost anything within an application. In simple terms, authorization **evaluates a user's ability to access** the system and up to what extent.

Other words:

Authentication - access to the system. This is the process of **verifying if a user is who they claim to be** by checking their credentials.

Authorization - access to the resources. This is the method of **checking the privileges of a user** and granting access to only specific resources.

Hadoop HDFS

Authentication vs. Authorization

Hadoop supports two mechanisms to authenticate a user. This is determined by the value in the `hadoop.security.authentication property`. The values can be of two kinds, which are as follows:

- `simple`: identity of the user is determined and presented by the OS that runs the client process.

- `kerberos`: identity of the user is determined by its Kerberos credentials.

HDFS itself cannot create, modify, or delete any identities. All identity management happens outside HDFS, either in the OS, as with simple authentication, or with Kerberos. HDFS simply uses the identity presented to it and performs authorization checks on the identity.

Hadoop HDFS

Authentication vs. Authorization

Hadoop provides authorization for both HDFS and all Hadoop services.

The HDFS authorization model is very similar to the authorization model in a POSIX system. In POSIX, each resource (files and directories) is associated with an owner user and a group. HDFS is similar to this. Permissions are given to each of these identities separately. There are separate permissions for:

• The owner of the resource.

• The group that is associated with the resource (and this way with all users belonging to this group).

• All other users within the system.

There are two permissions levels: *read* and *write*. In contrast with POSIX, there is no *execute* permission on files in HDFS as files are not executables.

Like in any UNIX-based operating system, permissions are encoded as three octal numbers. The first octal number indicates the `rwx` (`r` for read, `w` for write, `x` for execute) bits for the owner, the second for the group, and the third for all other users.

Hadoop HDFS

Authentication vs. Authorization

Files permissions

Any user, or a user belonging to a group that has the read permission, is allowed to read the contents of a file from HDFS. Similarly, any user, or a user belonging to a group that has write permissions, is allowed to write or append to existing files. A user or group can be given both read and write permissions.

Directories permisions

For directories a read permission allows the user or users belonging to the group to list the contents of the directory. Write permissions allow the users or groups to create files and directories or append to files residing within the directory. Execute permission allows the user or group to access the children of a directory.

Hadoop HDFS

Authentication vs. Authorization

Determine the list of groups the user belongs to

The user to group mapping services is determined by the value in the `hadoop. security.group.mapping property`. By default, this value is set to `org.apache.hadoop. security.ShellBasedUnixGroupsMapping`. When this group mapping is used, the username is sent to a UNIX shell command to determine the list of groups the user belongs to.

Specifying the HDFS superuser

There is no permanent superuser identity. It is strictly determined by the identity that starts the NameNode service. In Hadoop, the HDFS superuser is the user under whom the NameNode runs. The superuser has ultimate privileges in HDFS. No permission checks fail for the superuser, and the superuser identity is allowed to execute all operations.

It is not necessary for the superuser identity to be the administrator of the host that runs the NameNode though.

However, the administrator might also specify a group of users as superuser identities, though they might not be running the NameNode. Setting the configuration property `dfs.permissions.supergroup` to the group of users that need to have superuser privileges does this.

Limiting HDFS usage

HDFS provides quotas to limit usage. Quotas are imposed on

- the number of names
- and amount of space that can be used.

These quotas are fixed at a directory level and applied to all files and directories that are descendants of the directory.

Hadoop HDFS

Command line

Except that HDFS is highly distributed and fault tolerant it shares many typical operation performed on file systems.

HDFS file systems are only one way that Hadoop implements a file system. There are several other Java implementations of file systems that work with Hadoop. These include local file systems (file), WebHDFS (WebHDFS), HAR (Hadoop archive files), View (viewfs), S3 (s3a) and others. For each file system, Hadoop uses a different URI scheme for the file system instance in order to connect with it. For example, to list files stored on the local Linux file system you should use the `ls` command with the following file URI scheme:

```
$ hdfs dfs —ls file:///
```

You can access HDFS in various ways:

- From Java API
- From Command Line Interpreter (CLI)
- Using REST API (WebHDFS)
- Using Hue web application

Hadoop HDFS

Command line

All HDFS commands are invoked via the File System (FS) shell includes various shell-like commands that directly interact with the HDFS as well as other file systems that Hadoop supports.

The FS shell is invoked by:

```
bin/hadoop fs <args>
```

or simply

```
hadoop fs <args>
```

If HDFS is being used, `hdfs dfs` is a synonym used to invoke HDFS commands

```
bin/hdfs dfs <args>
```

HDFS file system commands are in many cases quite similar to familiar Linux file system commands. You may view all available HDFS commands by simply invoking the hdfs dfs command with no options, as shown here:

```
$ hdfs dfs

Usage: hadoop fs [generic options]
        [-appendToFile <localsrc> ... <dst>]
        [-cat [-ignoreCrc] <src> ...]
```

or

```
$ hadoop fs -help

Example: hadoop fs [generic options]
        [-appendToFile ... ]
        [-cat [-ignoreCrc] ...]
```

Hadoop HDFS

List of all commands [HCLI:1]

appendToFile	expunge	renameSnapshot
cat	find	rm
checksum	get	rmdir
chgrp	getfacl	rmr
chmod	getfattr	setfacl
chown	getmerge	setfattr
copyFromLocal	head	setrep
copyToLocal	help	stat
count	ls	tail
cp	lsr	test
createSnapshot	mkdir	text
deleteSnapshot	moveFromLocal	touch
df	moveToLocal	touchz
du	mv	truncate
dus	put	usage

Hadoop HDFS

Commands of limited value [HCLI:1]

Here is the list of shell commands which generally have no effect - and may actually fail.

`appendToFile`

`checksum`

`chgrp`

`chmod`

`chown`

`createSnapshot`

`deleteSnapshot`

`df`

`getfacl`

`getfattr`

`renameSnapshot`

`setfacl`

`setfattr`

`setrep`

`truncate`

The security and permissions models of object stores are usually very different from those of a Unix-style filesystem; operations which query or manipulate permissions are generally unsupported. Operations to which this applies include: `chgrp`, `chmod`, `chown`, `getfacl`, and `setfacl`.

Hadoop HDFS

List of all commands [HCLI:1]

appendToFile	expunge	renameSnapshot
cat	find	rm
checksum	get	rmdir
chgrp	getfacl	rmr
chmod	getfattr	setfacl
chown	getmerge	setfattr
copyFromLocal	head	setrep
copyToLocal	help	stat
count	ls	tail
cp	lsr	test
createSnapshot	mkdir	text
deleteSnapshot	moveFromLocal	touch
df	moveToLocal	touchz
du	mv	truncate
dus	put	usage

Hadoop HDFS

Task

Familiarize yourself with HDFS commands [HCLI:1-3].

Hadoop MapReduce

Hadoop MapReduce

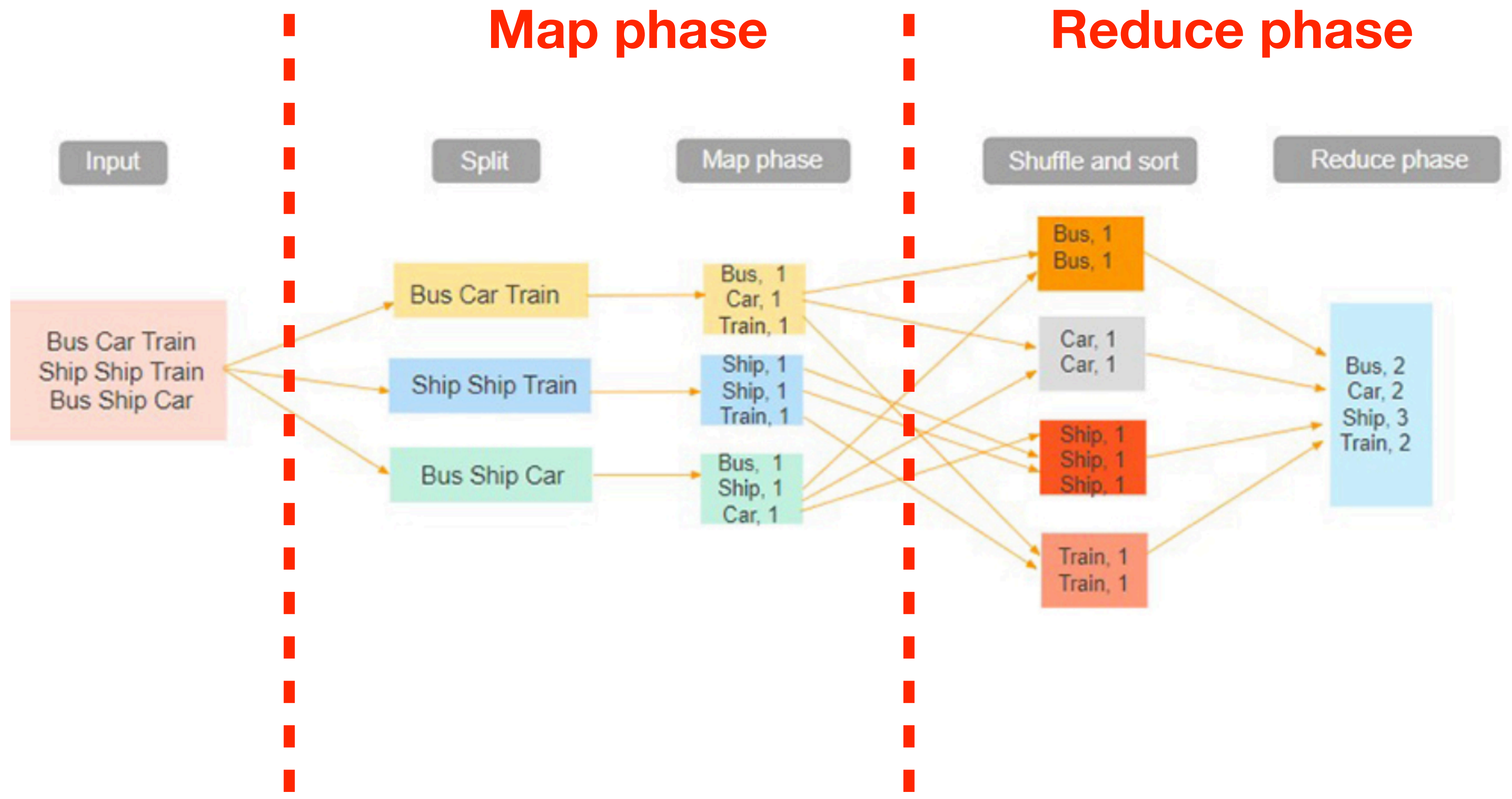
Hadoop MapReduce is the way we process data in Hadoop. In the MapReduce approach, the processing is done in distributed way at the slave nodes, and the final result is sent to the master node.

At a high level of abstraction we can say that a single processing run of the MapReduce is composed of a **map task** and a **reduce task**.

tutu rysunek: input -> map task -> reduce task

Hadoop MapReduce

Behind a scene both **map task** and a **reduce task** consists of multiple stages, so we should rather say about **map phase** and **reduce phase**.



Being more precise, we have the following stages:

- **map**: split, map, [combine], partition;
- **reduce**: shuffle, sort, reduce.

Hadoop MapReduce

MapReduce in details: before we start

To process some data with MapReduce we need to have this data.

For simplicity, let our input consist of data about 10 persons: Anna which is 12 years old, next Bob (34), Celine (63), Diana (51), Edmund (55), Fracesco (67), Giselle (15), Henry (57), Isabell (32) and finally Jane (39):

```
Anna 12
Bob 34
Celine 63
Diana 51
Edmund 55
Francesco 67
Giselle 15
Henry 57
Isabell 32
Jane 39
```

With this data set suppose, we have to perform a count on the person list taking into consideration an age ranges: 0-9 years, 10-19 years, 20-29 years etc. So we expect to get output like this

```
range 10-19: 2 (as w have: Anna, Giselle)
range 30-39: 3 (as w have: Bob, Isabell, Jane)
range 50-59: 3 (as w have: Edmund, Diana, Henryk)
range 60-69: 2 (as w have: Celine, Francesco)
```

Hadoop MapReduce

MapReduce in details: map phase, split stage

- **map**: split, map, [combine], partition;
- **reduce**: shuffle, sort, reduce.

The input is **divided** into splits. This will **distribute** the work among all the map nodes. Each split is parsed into its constituent records as a key-value pair `(K1, V1)`. The key is usually the *ordinal position* of the record, and the value is the *actual record*.

We divide the input in three splits. This will distribute the work among all the map nodes.

- **split 1**: Anna (12), Bob (34), Celine (63); we get a set of pairs:
  `{1: ("anna": 12), 2: ("bob": 34), 3: ("celine": 63)}`
- **split 2**: Diana (51), Edmund (55), Fracesco (67); we get a set of pairs:
  `{4: ("diana": 51), 5: ("edmund": 55), 6: ("francesco": 67)}`
- **split 3**: Giselle (15), Henry (57), Isabell (32), Jane (39); we get a set of pairs:
  `{7: ("giselle": 15), 8: ("henry": 57), 9: ("isabell": 32), 10: ("jane": 39)}`

Hadoop MapReduce

MapReduce in details: map phase, map stage

- **map**: split, **map**, [combine], partition;
- **reduce**: shuffle, sort, reduce.

The map function executes user-defined logic. From the previous stage we know that each split contains multiple key-value pairs, and the mapper is **run once for each key-value pair in the split**. The mapper processes each key-value pair according to the user-defined logic and further generates a new key-value pair as its output `(K2, V2)`.

It should be noted that for an input key-value pair `(K1, V1)`, a mapper may not produce any output key-value pair (filtering) or can generate multiple key-value pairs (demultiplexing).

When all records of the split have been processed, the output is a list of key-value pairs (`list(K2, V2)`) where multiple key-value pairs can exist for the same key.

The mapping process remains the same on all the nodes.

Going back to our example, in each of the mapper we give a hardcoded value (1) to each of the person. The rationale behind giving a hardcoded value equal to 1 is that every given person, in itself, will occur once. Next a list of key-value pair is created where the key is nothing but the age range and value is one. So, for our set of splits we have the following list of a key-value pairs:

- **node 1**: `list(("10-19": 1), ("30-39": 1), ("60-69": 1))`
- **node 2**: `list(("50-59": 1), ("50-59": 1), ("60-69": 1))`
- **node 3**: `list(("10-19": 1), ("50-59": 1), ("30-39": 1), ("30-39": 1))`

Hadoop MapReduce

MapReduce in details: map phase, map stage

Last stage:

- **split 1**: Anna (12), Bob (34), Celine (63); we get a set of pairs: `{1: ("anna": 12), 2: ("bob": 34), 3: ("celine": 63)}`
- **split 2**: Diana (51), Edmund (55), Fracesco (67); we get a set of pairs: `{4: ("diana": 51), 5: ("edmund": 55), 6: ("francesco": 67)}`
- **split 3**: Giselle (15), Henry (57), Isabell (32), Jane (39); we get a set of pairs: `{7: ("giselle": 15), 8: ("henry": 57), 9: ("isabell": 32), 10: ("jane": 39)}`

Now:

Going back to our example, in each of the mapper we give a hardcoded value (1) to each of the person. The rationale behind giving a hardcoded value equal to 1 is that every given person, in itself, will occur once. Next a list of key-value pair is created where the key is nothing but the age range and value is one. So, for our set of splits we have the following list of a key-value pairs:

- **node 1**: `list(("10-19": 1), ("30-39": 1), ("60-69": 1))`
- **node 2**: `list(("50-59": 1), ("50-59": 1), ("60-69": 1))`
- **node 3**: `list(("10-19": 1), ("50-59": 1), ("30-39": 1), ("30-39": 1))`

Hadoop MapReduce

MapReduce in details: map phase, combine stage

- **map**: split, map, [**combine**], partition;
- **reduce**: shuffle, sort, reduce.

Generally, the output of the map function is handled directly by the reduce function.

However, map tasks and reduce tasks are mostly run over different nodes. This requires moving data between mappers and reducers. This data movement can consume a lot of bandwidth and directly contributes to processing latency. With larger datasets, the time taken to move the data between map and reduce stages can exceed the actual processing undertaken by the map and reduce tasks. For this reason, the MapReduce engine provides an **optional combine function** that summarizes a mapper's output before it gets send to reducer and processed by the them.

A **combiner is essentially a reducer function** that locally groups a mapper's output on the same node as the mapper. A reducer function can be used as a combiner function, or a custom user-defined function can be used.

The MapReduce engine combines all values for a given key from the mapper output, creating multiple key-value pairs as input to the combiner where the key is not repeated and the value exists as a list of all corresponding values for that key. The **combiner stage is only an optimization stage**, and may therefore not even be called by the MapReduce engine. Combiner even may have no sense for some tasks. For example, it will work for finding the largest or the smallest number, but will not work for finding the average of all numbers since it only works with a subset of the data.

Hadoop MapReduce

MapReduce in details: map phase, combine stage

Last stage:

- **node 1**: `list(("10-19": 1), ("30-39": 1), ("60-69": 1))`
- **node 2**: `list(("50-59": 1), ("50-59": 1), ("60-69": 1))`
- **node 3**: `list(("10-19": 1), ("50-59": 1), ("30-39": 1), ("30-39": 1))`

Now:

Combine locally combines all values for a given key from the mapper output, creating multiple key-value pairs as input to the combiner where the key is not repeated and the value exists as a list of all corresponding values for that key.

- **node 1**: `list(("10-19": [1]), ("30-39": [1]), ("60-69": [1]))`
- **node 2**: `list(("50-59": [1, 1]), ("60-69": [1]))`
- **node 3**: `list(("10-19": [1]), ("50-59": [1]), ("30-39": [1, 1]))`

and finally

- **node 1**: `list(("10-19": 1), ("30-39": 1), ("60-69": 1))`
- **node 2**: `list(("50-59": 2), ("60-69": 1))`
- **node 3**: `list(("10-19": 1), ("50-59": 1), ("30-39": 2))`

Hadoop MapReduce

MapReduce in details: map phase, partition stage

- **map**: split, map, [combine], **partition**;
- **reduce**: shuffle, sort, reduce.

The output from the previous stage is divided into partitions between reducer instances. **Although each partition contains multiple key-value pairs, all records for a particular key are assigned to the same partition.** The MapReduce engine guarantees a random and fair distribution between reducers while making sure that all of the same keys across multiple mappers end up with the same reducer instance.

Depending on the nature of the job, certain reducers can sometimes receive a large number of key-value pairs compared to others. As a result of this uneven workload, some reducers will finish earlier than others. Overall, this is less efficient and leads to longer job execution times than if the work was evenly split across reducers. This can be rectified by customizing the partitioning logic in order to guarantee a fair distribution of key-value pairs.

Hadoop MapReduce

MapReduce in details: map phase, partition stage

Last stage:

- **node 1**: `list(("10-19": 1), ("30-39": 1), ("60-69": 1))`
- **node 2**: `list(("50-59": 2), ("60-69": 1))`
- **node 3**: `list(("10-19": 1), ("50-59": 1), ("30-39": 2))`

Now:

Each partition contains multiple key-value pairs, all records for a particular key are assigned to the same partition.

- **node 1:** `{"10-19": 1` ⟶ `P1, "30-39": 1` ⟶ `P1, "60-69": 1` ⟶ `P3}`
- **node 2**: `{"50-59": 2` ⟶ `P2, "60-69": 1` ⟶ `P3}`
- **node 3**: `{"10-19": 1` ⟶ `P1, "50-59": 1` ⟶ `P2, "30-39": 2` ⟶ `P1}`

Hadoop MapReduce

MapReduce in details: map phase, shuffle stage

- **map**: split, map, [combine], partition;
- **reduce**: **shuffle**, sort, reduce.

The output from all partitioners is **copied** across the network to the nodes running the reduce task. This is known as *shuffling*. The list based key-value output from each partitioner can contain the same key multiple times.

Hadoop MapReduce

MapReduce in details: map phase, shuffle stage

Last stage:

- **node 1:** `{"10-19": 1` $\longrightarrow$ `P1, "30-39": 1` $\longrightarrow$ `P1, "60-69": 1` $\longrightarrow$ `P3}`
- **node 2**: `{"50-59": 2` $\longrightarrow$ `P2, "60-69": 1` $\longrightarrow$ `P3}`
- **node 3**: `{"10-19": 1` $\longrightarrow$ `P1, "50-59": 1` $\longrightarrow$ `P2, "30-39": 2` $\longrightarrow$ `P1}`

Now:

Output from all partitioners is copied across the network to the nodes running the reduce task (*shuffling*).

- **partition on node 1**: `{"10-19": 1, "30-39": 1, "10-19": 1, "30-39": 2}`
- **partition on node 2**: `{"50-59": 2, "50-59": 1}`
- **partition on node 3**: `{"60-69": 1, "60-69": 1}`

Hadoop MapReduce

MapReduce in details: map phase, sort stage

- **map**: split, map, [combine], partition;
- **reduce**: shuffle, **sort**, reduce.

Next, the MapReduce engine automatically **groups and sorts** the key-value pairs according to the keys so that the output contains a sorted list of all input keys and their values with the same keys appearing together. The way in which keys are grouped and sorted can be customized.

This merge creates a single key-value pair per group, where key is the group key and the value is the list of all group values.

Hadoop MapReduce

MapReduce in details: map phase, sort stage

Last stage:

- **partition on node 1**: `{"10-19": 1, "30-39": 1, "10-19": 1, "30-39": 2}`
- **partition on node 2**: `{"50-59": 2, "50-59": 1}`
- **partition on node 3**: `{"60-69": 1, "60-69": 1}`

Now:

Next, the MapReduce engine automatically groups

- **partition on node 1**: `{"30-39": 1, "30-39": 2, "10-19": 1, "10-19": 1}`
- **partition on node 2**: `{"50-59": 2, "50-59": 1}`
- **partition on node 3**: `{"60-69": 1, "60-69": 1}`

and sorts the key-value pairs according to the keys so that the output contains a sorted list of all input keys and their values with the same keys appearing together.

- **node 1**: `{"10-19": [1, 1], "30-39": [1, 2]}`
- **node 2**: `{"50-59": [2, 1]}`
- **node 3**: `{"60-69": [1, 1]}`

Hadoop MapReduce

MapReduce in details: map phase, reduce stage

- **map**: split, map, [combine], partition;

- **reduce**: shuffle, sort, **reduce**.

Reduce is the final stage of the reduce phase. Depending on the user-defined logic specified in the reduce function (reducer), the reducer will either further summarize its input or will emit the output without making any changes. In either case, for each key-value pair that a reducer receives, the list of values stored in the value part of the pair is processed and another key-value pair is written out.

Hadoop MapReduce

MapReduce in details: map phase, reduce stage

Last stage:

- **node 1**: `{"10-19": [1, 1], "30-39": [1, 2]}`
- **node 2**: `{"50-59": [2, 1]}`
- **node 3**: `{"60-69": [1, 1]}`

Now:

The reducer will summarize its input. For each key-value pair that a reducer receives, the list of values stored in the value part of the pair is processed and another key-value pair is written out.

- **node 1**: `{"10-19": 2, "30-39": 3}`
- **node 2**: `{"50-59": 3}`
- **node 3**: `{"60-69": 2}`

Hadoop MapReduce

MapReduce in details: result

The result output is ordered by keys

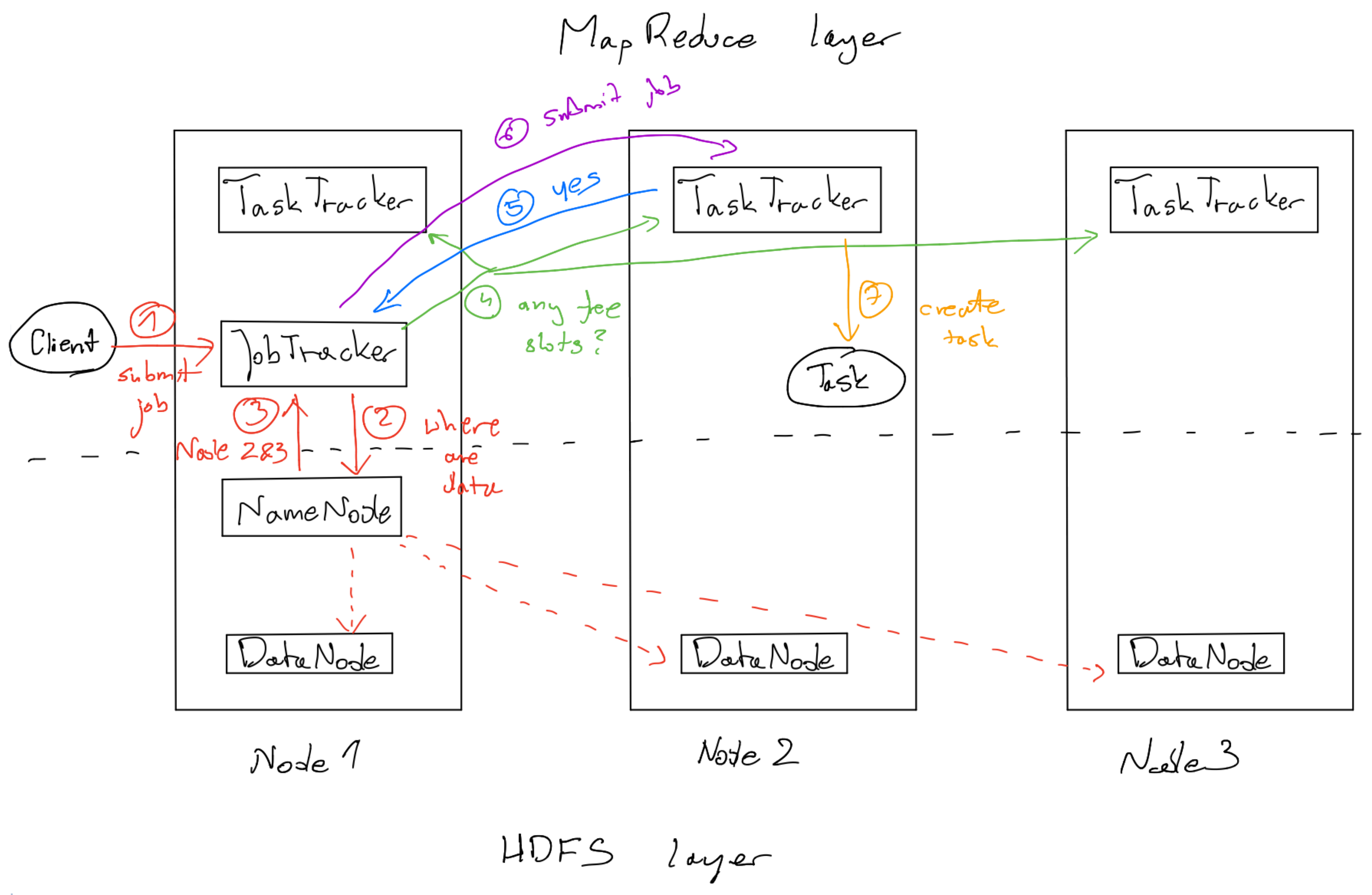
```
"10-19": 2
"30-39": 3
"50-59": 3
"60-69": 2
```

Hadoop YARN

Hadoop YARN

Hadoop before YARN

It was discussed on previous slides: *Hadoop HDFS. HDFS Layer and MapReduce Layer components*

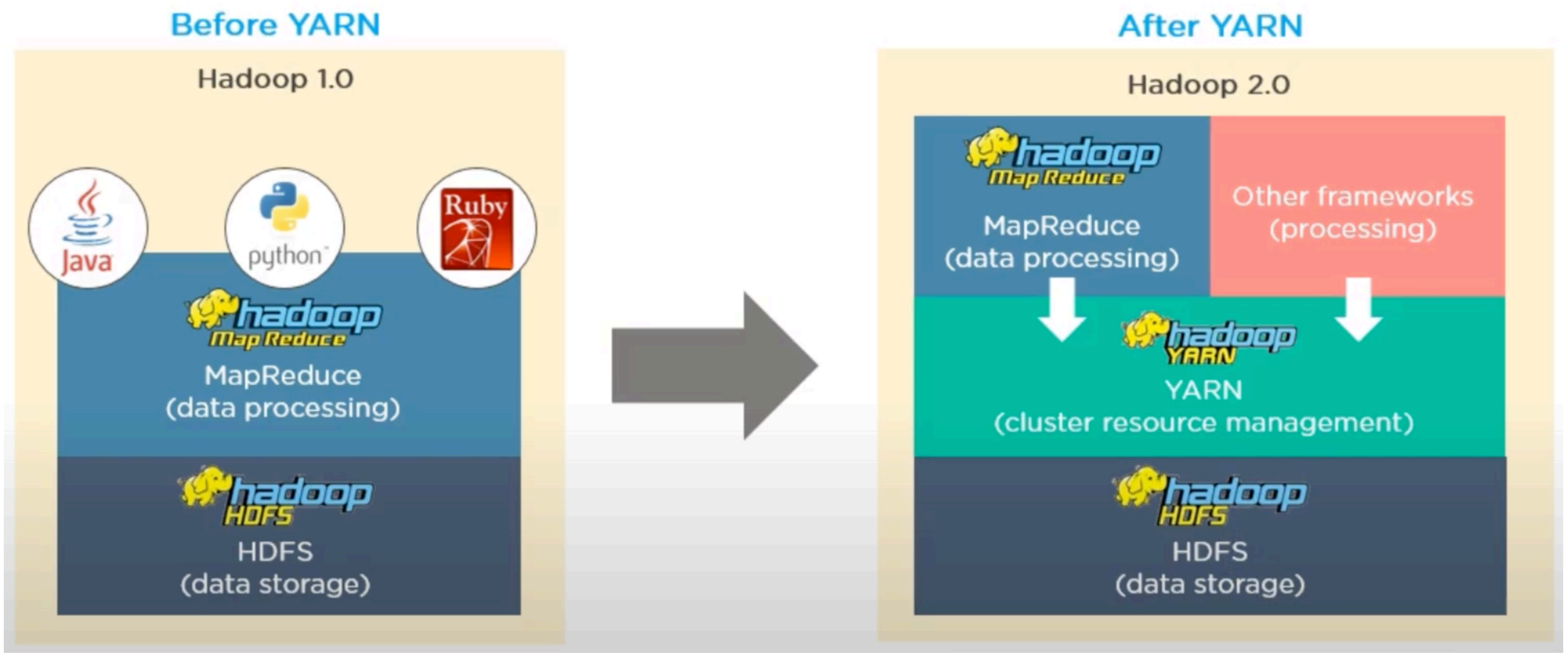


Drawback:

- There is only one JobTracker so
  - scalability became a bottleneck;
  - it became a SPOF (single point of failure) - any failure kills all running jobs.
- Limitations in running non-MapReduce applications as all task must pass through MapReduce layer.

Hadoop YARN

Transition to YARN



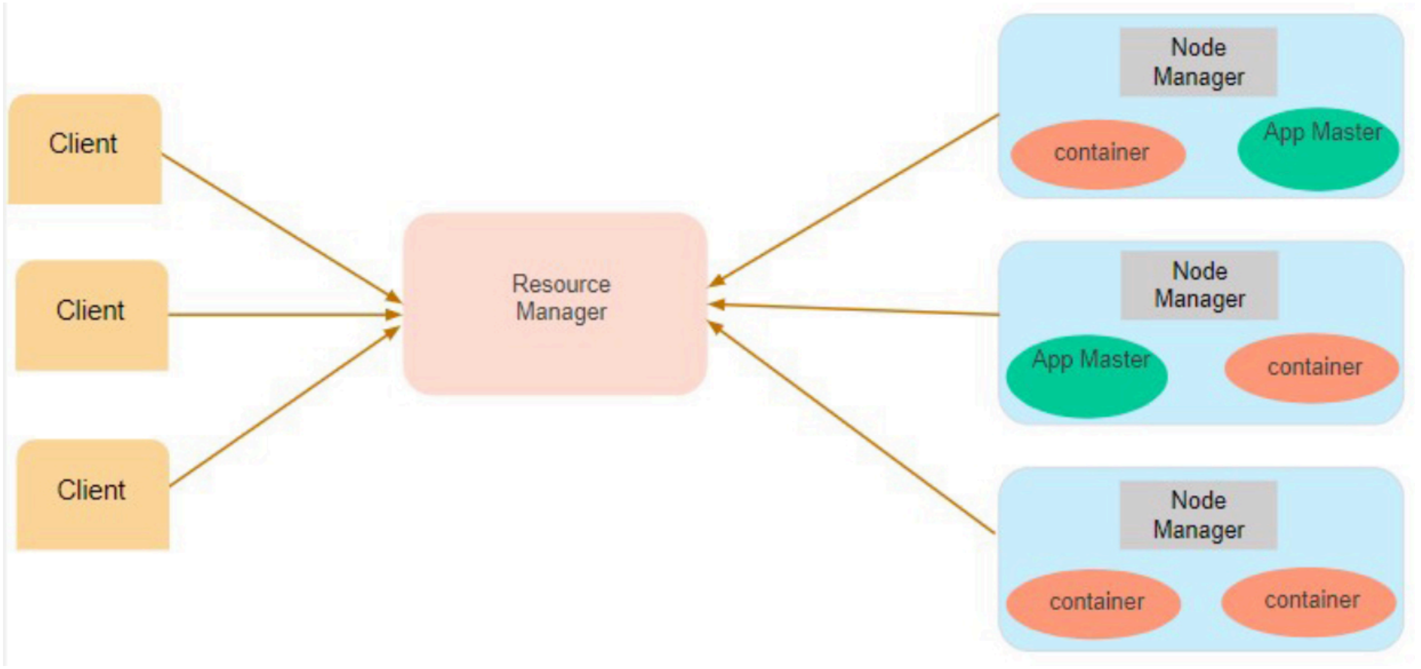
Hadoop YARN

Hadoop YARN (Yet Another Resource Negotiator) is the cluster resource management layer of Hadoop, placed between HDFS and MapReduce in the Hadoop architecture, and is responsible for resource allocation and job scheduling. It was introduced in the Hadoop 2.0 version.

It is responsible for managing cluster resources. It performs job scheduling to make sure that the jobs are scheduled in the right place.

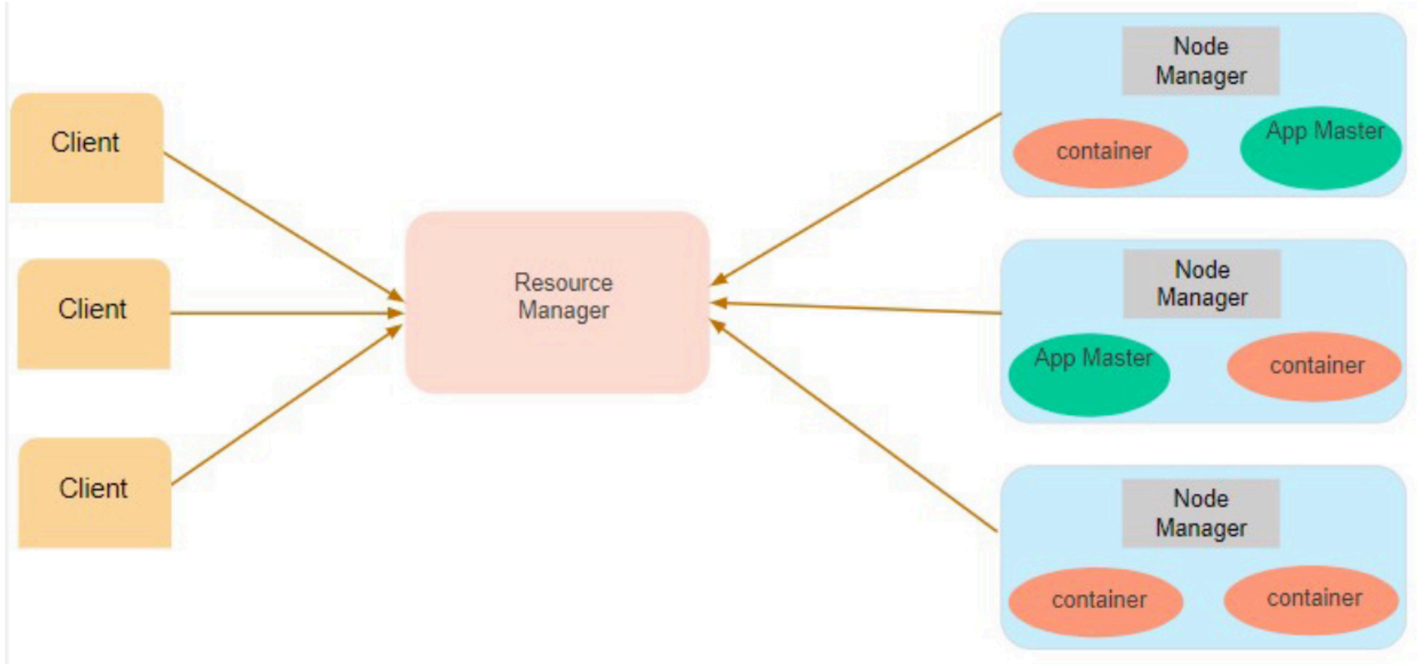
The elements of YARN include:

- ResourceManager (one per cluster)
    - Scheduler
    - Applications Manager
- ApplicationMaster (one per application)
- NodeManagers (one per node)



Hadoop YARN

The Resource Manager maintains the list of applications on the cluster and available resources on the Node Manager. The Resource Manager determines the next application that receives a portion of the cluster resource. The decision is subject to many constraints such as queue capacity, Access Control Lists, and fairness.



The Scheduler is responsible for allocating resources to various running applications depending on the common constraints of capacities, queues, and so on.

The Application Manager is an interface which maintains a list of applications that have been submitted, currently running, or completed. The Application Manager is responsible for accepting job-submissions, negotiating the first container for executing the application-specific Application Master and restarting the Application Master container on failure.

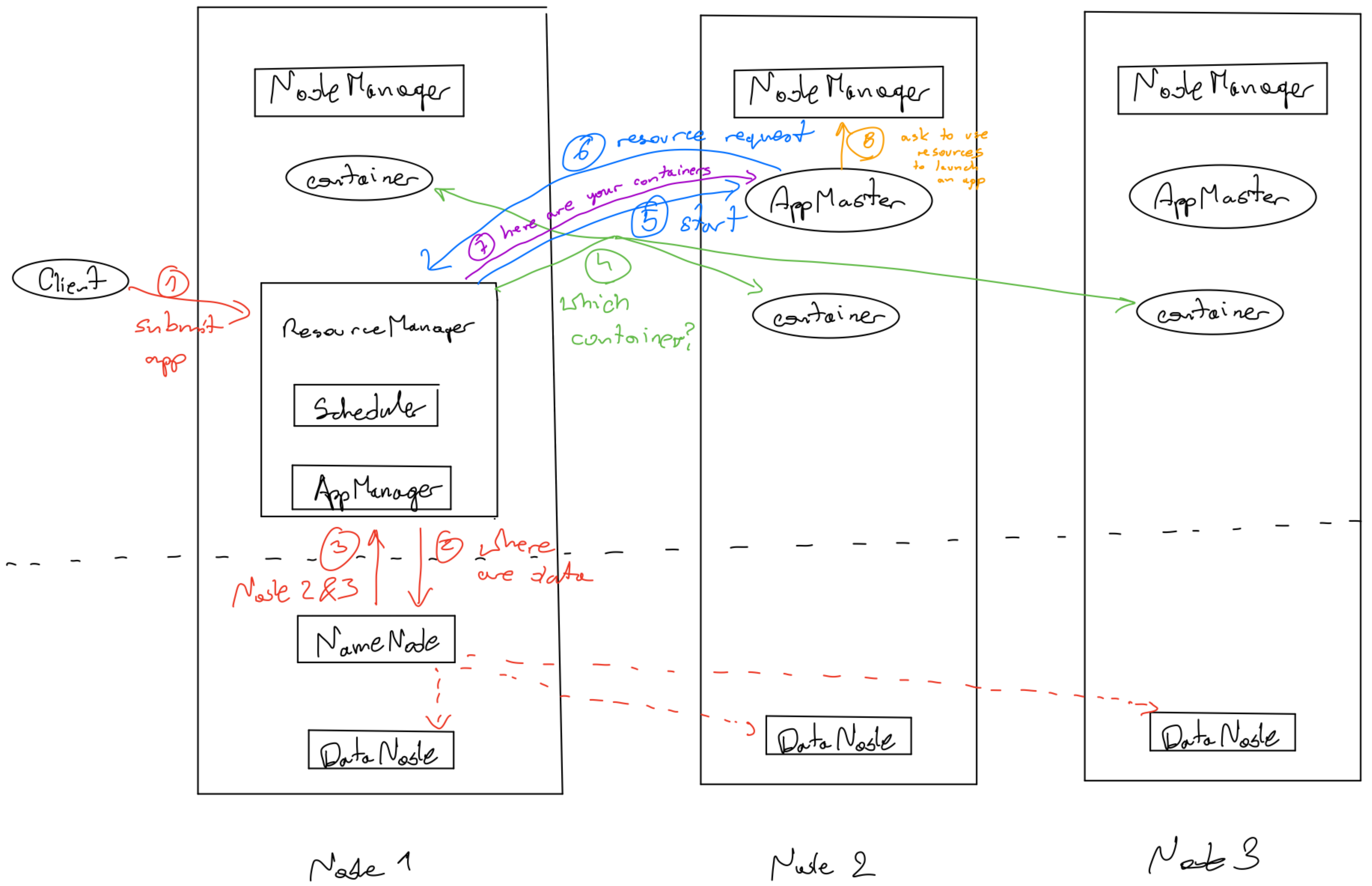
the Application Master negotiates containers to launch all of the tasks needed to complete its application. It also monitors the progress of an application and its tasks, restarts failed tasks in newly requested containers, and reports progress back to the client that submitted the application.

Each of the nodes has its *node manager*. NodeManager does not monitor tasks; it only monitors the resource usage in the nodes (containers). For example, it kills a container if it consumes more memory than initially allocated. The *containers* contain a collection of physical resources, which could be RAM, CPU, or hard drives. Whenever a job request comes in, **the *app master* requests the container from the node manager**. Once the node manager gets the resource, it goes back to the Resource Manager.

Steps to Running an application with YARN

1. Client submits an application to the ResourceManager.
2. Determine the location of the data.
3. ResourceManager selects a container.
4. ApplicationMaster is started.
5. ApplicationMaster sends resource request to ResourceManager.
6. ResourceManager sends back information about containers.
7. ApplicationManager asks the related NodeManager to use resources to launch an application specific task. The Application Master is responsible for the entire life-cycle of that particular application.
8. NodeManager launches the container.
9. ApplicationMaster executes application and manages its life-cycle.

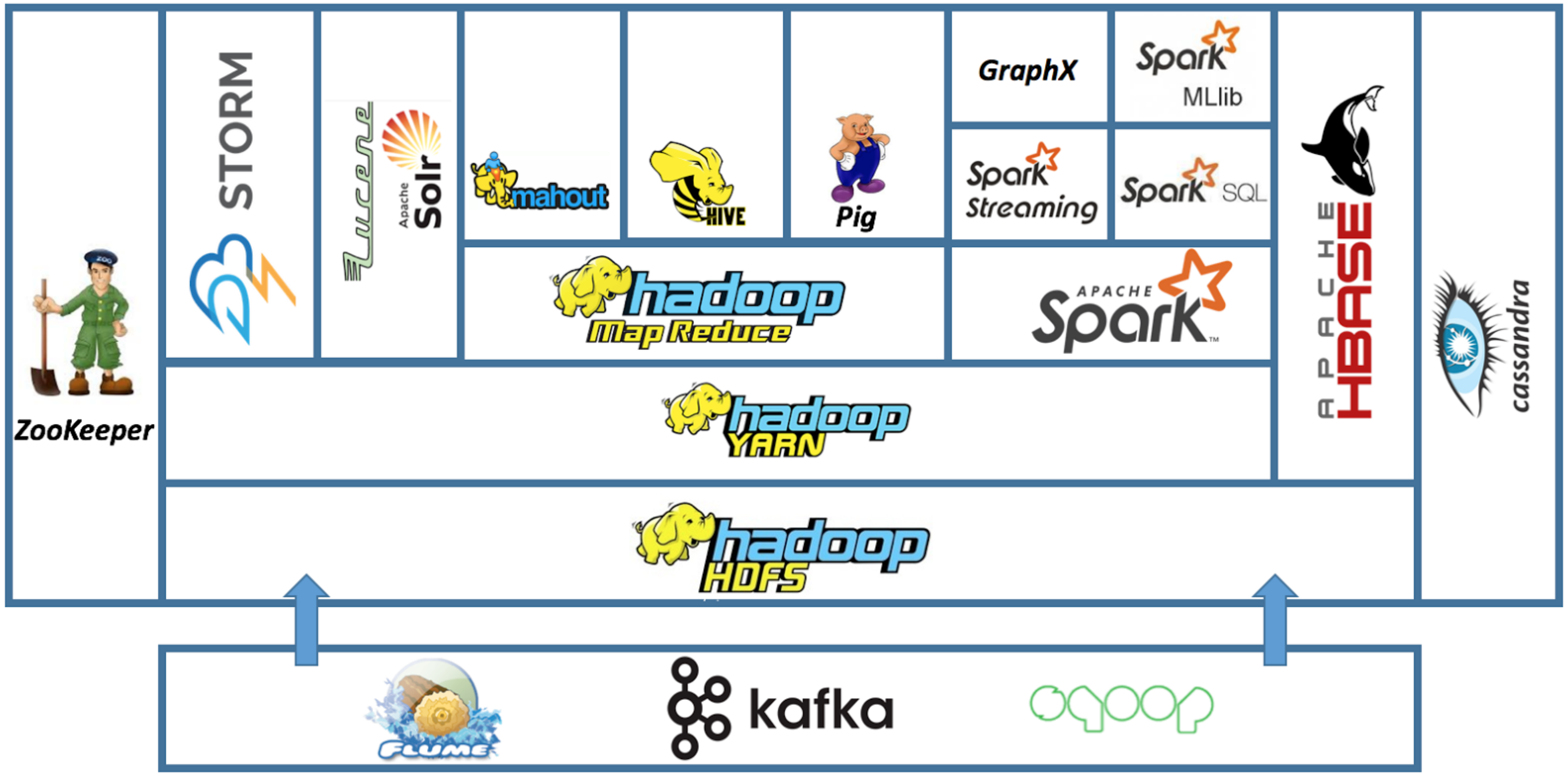
Steps to Running an application with YARN



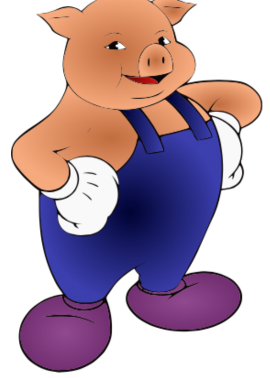
Hadoop ecosystem

Hadoop ecosystem

Hadoop ecosystem is build on top of HDFS, YARN and MapReduce and consist of numerous projects.



Hadoop ecosystem
Pig



Apache Pig is a high-level platform for creating programs that run on Apache Hadoop.

The language for this platform is called Pig Latin. Pig can execute its Hadoop jobs in MapReduce, Apache Spark and Apache Tez.

Pig Latin abstracts the programming from the Java MapReduce idiom into a notation which makes MapReduce programming high level, similar to that of SQL for relational database management systems. Although similar to SQL, Pig Latin is still a procedural language which is in contrast to declarative nature of SQL.

Pig Latin can be extended using user-defined functions which the user can write in Java, Python, JavaScript, Ruby or Groovy.

Apache Pig was originally developed at Yahoo Research around 2006 for researchers to have an ad-hoc way of creating and executing MapReduce jobs on very large data sets. In 2007, it was moved into the Apache Software Foundation. As for now (November 2020) latest stable version is 0.17 dated 2017-06-19.

Hadoop ecosystem

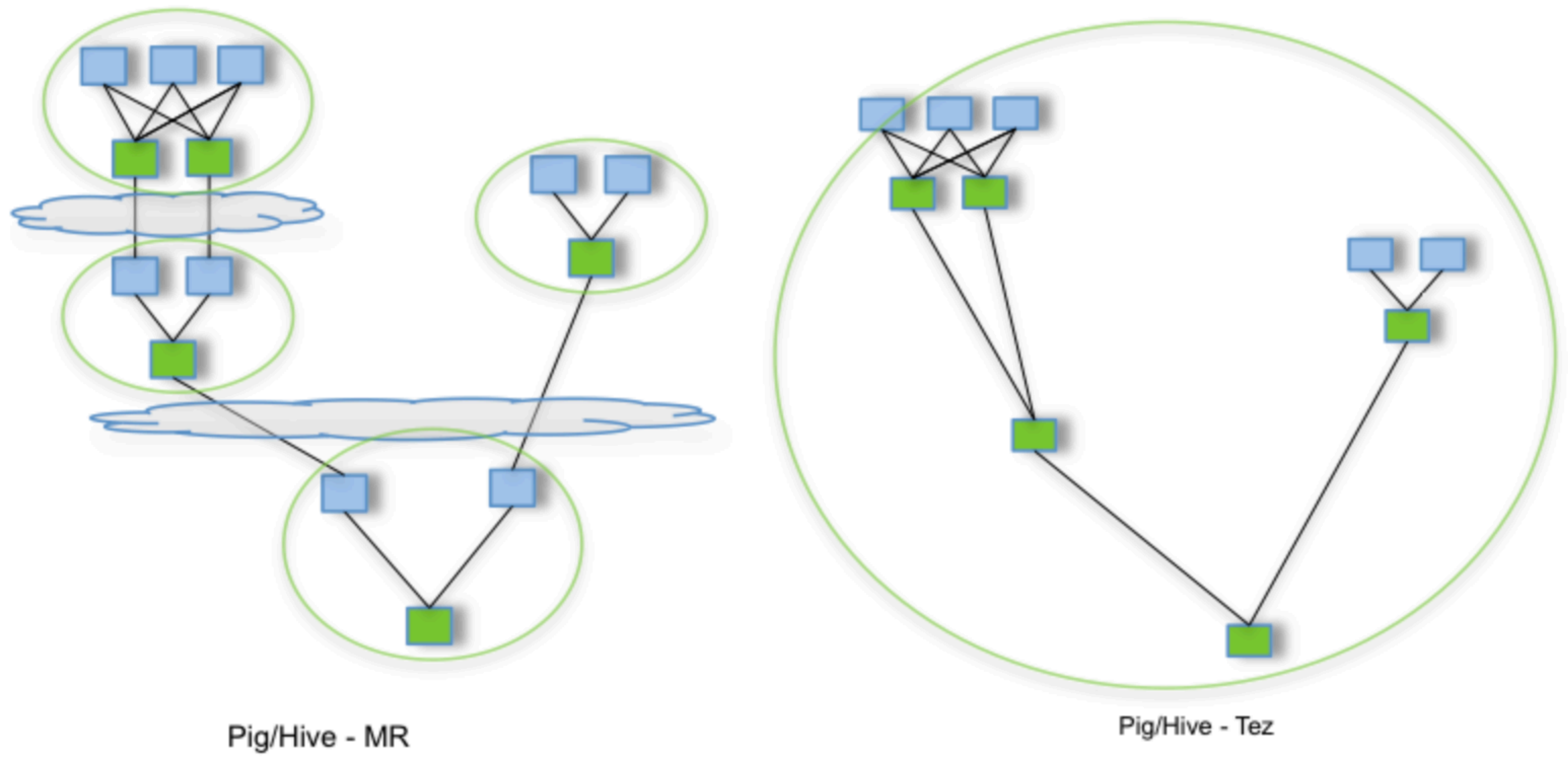
Pig

- In-built operators: Apache Pig provides a very good set of operators for performing several data operations like sort, join, filter, etc.
- Ease of programming: Since Pig Latin has similarities with SQL, it is very easy to write a Pig script.
- Automatic optimization: The tasks in Apache Pig are automatically optimized. This makes the programmers concentrate only on the semantics of the language.
- Handles all kinds of data: Apache Pig can analyze both structured and unstructured data and store the results in HDFS.
- Extensibility: users can develop their own functions.

Hadoop ecosystem

Pig

The Pig script is transformed by the parser into a logical execution plan expressed as a DAG (Directed Acyclic Graph). Next DAG (after passing by logical optimizer) is converted by the compiler into MapReduce jobs. As DAG are the base mechanizm under the Pig, this explains why Apache Tez, which is a framework allowing for a complex DAC of tasks for processing data, may be used with Pig.



Hadoop ecosystem

Pig

- example???

Hadoop ecosystem

Hive

Apache Hive is a project built on top of Apache Hadoop for providing data query and analysis.

Hive provides a SQL-like query language called HiveQL to query data stored in various databases and file systems that integrate with Hadoop. Internally, a compiler translates HiveQL statements into a directed acyclic graph of MapReduce, Tez, or Spark jobs, which are submitted to Hadoop for execution.

Hive provides the necessary SQL abstraction to integrate SQL-like queries (HiveQL) into the underlying Java without the need to implement queries in the low-level Java API.

Since most data warehousing applications work with SQL-based querying languages, Hive aids portability of SQL-based applications to Hadoop.

Hadoop ecosystem

Hive

- example???

Hadoop ecosystem

Pig vs. Hive

Pig	Hive
Procedural Data Flow Language	Declarative SQLish Language
Mainly used by Researchers and Programmers	Mainly used by Data Analysts
Pig is SQL like but varies to a great extent.	Directly leverages SQL and is easy to learn for database experts.

Generally speaking Apache Pig is considered to be faster than Apache Hive (36% faster for join operations, 10% faster for filtering, 18% faster for filtering 90% of the data, 46% faster for arithmetic operations [PVHSIS]).

Hadoop ecosystem

Spark

Apache Spark is a cluster computing technology, designed for fast computation. In its ideas it is based on Hadoop MapReduce model but it extends it to efficiently use for more types of computations, which includes interactive queries and stream processing. **The main feature of Spark is its in-memory cluster computing that increases the processing speed of an application.** One of the most important capabilities in Spark is persisting (or caching) a dataset in memory across operations. This allows future actions to be much faster (often by more than 10x).

Spark is an independent computation framework and is not a modified version of Hadoop. Is not strictly dependent on Hadoop but Hadoop and Spark are not mutually exclusive and can work together. Spark only does processing and it uses dynamic memory to perform the task, but to store the data you need some data storage system. So, real-time and faster data processing in Hadoop is not possible without Spark. On the other hand, Spark doesn't have any file system for distributed storage.

Spark can run without Hadoop. You may run it as a standalone mode without any resource manager. But if you want to run in multi-node setup, you need a resource manager like YARN or Mesos and a distributed file system like HDFS, S3 etc.

Hadoop ecosystem

Spark - main features

Apache Spark has following features:

- **Speed** – Spark helps to run an application in Hadoop cluster, up to 100 times faster in memory, and 10 times faster when running on disk. This is possible by reducing number of read/write operations to disk as it stores the intermediate processing data in memory.
- **Advanced Analytics** – Spark not only supports *map* and *reduce*. It also supports
  - SQL queries,
  - Streaming data,
  - Machine learning (ML),
  - and Graph algorithms.
- **Supports multiple languages** – Spark provides built-in APIs in Java, Scala, or Python. Therefore, you can write applications in different languages.
- It is possible to work with Spark in **interactive mode.**

Hadoop ecosystem

Spark - RDD

MapReduce is widely adopted for processing and generating large datasets with a parallel, distributed algorithm on a cluster. Unfortunately, **in most current MapReduce frameworks, the only way to reuse data between computations is to write it to an external stable storage system**.

- **Example**: Iterative Operations on MapReduce
  Reuse intermediate results across multiple computations in multi-stage applications.

- **Example**: Interactive Operations on MapReduce
  User runs ad-hoc queries - different queries are run on the same set of data repeatedly. Unfortunately each query will do the disk I/O on the stable storage, which can dominate application execution time.

Regarding storage system, most of the Hadoop applications spend more than 90% of the time doing HDFS read-write operations. Data sharing is slow in MapReduce due to replication, serialization, and disk IO.

Hadoop ecosystem

Spark - RDD

The key idea of spark is **Resilient Distributed Datasets** (RDD); it supports in-memory processing computation. This means, it stores the state of memory as an object across the jobs and the object is sharable between those jobs. Data sharing in memory is 10 to 100 times faster than network and Disk.

RDD is actually an immutable (no edit) distributed collection of data, held by the nodes of a computing cluster. RDD can be processed in a distributed and multithreaded manner, and the basic operations on RDD are actions and transformations.

RDDs support two types of operations:

- transformations, which create a new dataset from an existing one,
- and actions, which return a value to the driver program after running a computation on the dataset.

For example, map is a transformation that passes each dataset element through a function and returns a new RDD representing the results. On the other hand, reduce is an action that aggregates all the elements of the RDD using some function and returns the final result to the driver program.

Spark revolves around the concept of a resilient distributed dataset (RDD), which is a fault-tolerant collection of elements that can be operated on in parallel. There are two ways to create RDDs:

- parallelizing an existing collection in your driver program,
- or referencing a dataset in an external storage system, such as a shared filesystem, HDFS, HBase, or any data source offering a Hadoop InputFormat.

All transformations in Spark are lazy, in that they do not compute their results right away. Instead, they just remember the transformations applied to some base dataset (e.g. a file). The transformations are only computed when an action requires a result to be returned to the driver program. This design enables Spark to run more efficiently. For example, we can realize that a dataset created through map will be used in a reduce and return only the result of the reduce to the driver, rather than the larger mapped dataset.

Hadoop ecosystem

Spark - ancestors of RDD

Along with the development of Spark and the increase in its popularity, the creators wanted to make the tool available to a wider group of users by extending the API. Following the "data frames" concept available in R and Python (Pandas), the developers of Apache Spark created a new type called **DataFrame**. Similar to RDD, DataFrame is a distributed data collection, however, **data is organized into named columns**, similar to databases or Apache Hive tools. Additionally, DataFrame has many optimizations (Catalyst optimizer) and API (domain-specific language) facilities. Also, similar to RDD, all operations are buffered and performed only when necessary (lazy), so Spark can apply many optimizations.

Apache Spark version 1.6 introduced a third API called **Dataset**. Unlike DataFrame, the new Api **is strongly type-safe and object-oriented** programming interface. Additionally, it has even more optimizations (Catalyst Optimizer and Tungsten project). The use of API is very similar to RDD, thus Dataset has become an optimal solution combining the advantages of both DataFrame and RDD, with much higher performance and less RAM usage than RDD.

Hadoop ecosystem

Spark or Hadoop?

Spark seems to be very similar to Hadoop - we can say even more: it's seems to be fast version of Hadoop. So which one should we choose? Should we still care about Hadoop? At the core, the choice is not between Hadoop and Spark itself but rather say to choose Disk Based Computing or RAM Based Computing.

Hadoop ecosystem

Mahout

Apache Mahout (pronaunced [mahut] or [mahał]) is a project used for creating scalable machine learning algorithms focused primarily on linear algebra. It implements popular machine learning techniques such as:

- recommendation,
- classification,
- and clustering.

Apache Mahout started as a sub-project of Apache’s Lucene in 2008. In 2010, Mahout became a top level project of Apache. In the past, many of the implementations use the Apache Hadoop platform, however for performance reasons today it is primarily focused on Apache Spark.

Hadoop ecosystem
HBase

HBase is a wide-column store and has been widely adopted because of its lineage with Hadoop and HDFS. HBase runs on top of HDFS and is well-suited for fast read and write operations on large datasets with high throughput and low input/output latency.

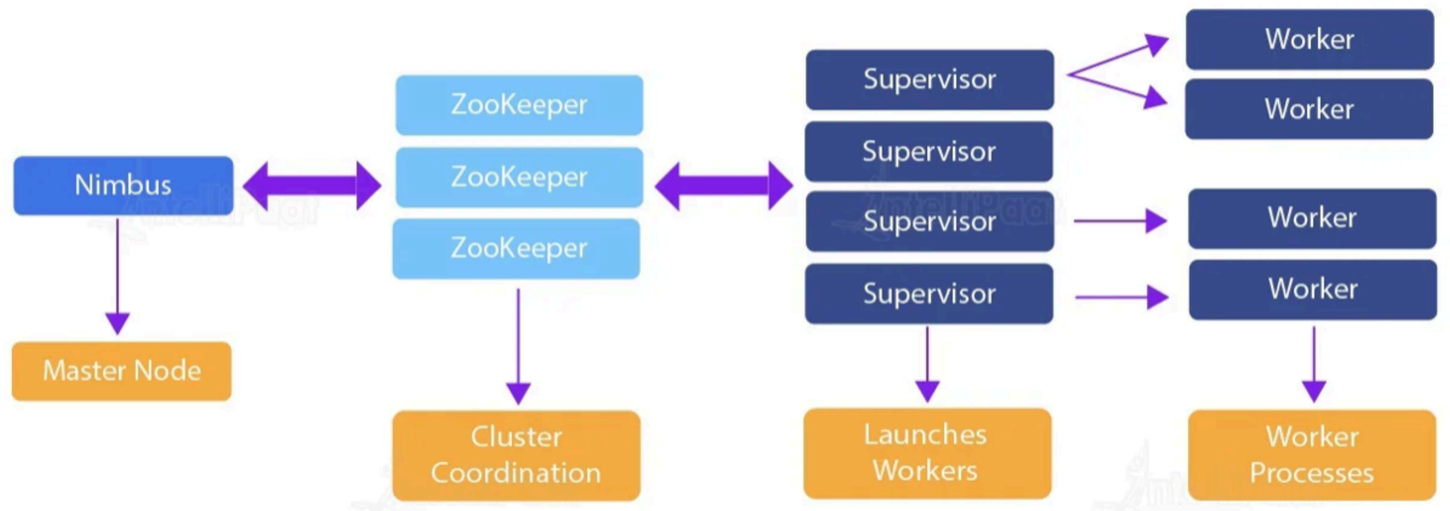
Tables in HBase can serve as the input and output for MapReduce jobs run in Hadoop, and may be accessed through the Java API, REST, Avro or Thrift gateway APIs.

HBase is not a direct replacement for a classic SQL database, however Apache Phoenix project provides a SQL layer for HBase.

Hadoop ecosystem

Storm

Apache Storm is an open-source distributed real-time computational system for processing data streams. Similar to what Hadoop does for batch processing, Apache Storm does for unbounded streams of data.



Physical level

Apache Storm architecture is quite similar to that of Hadoop however, there are certain differences. At physical level there are two types of nodes in the Storm cluster, similar to Hadoop, which are the master node and the worker nodes.

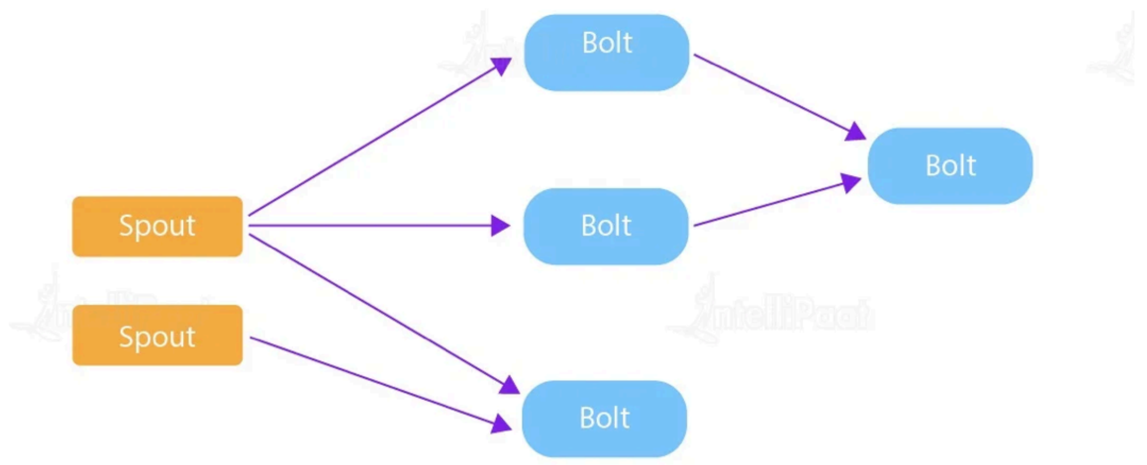
- The master node runs a daemon called *Nimbus*, which is similar to the *Job Tracker* in the Hadoop cluster. Nimbus is responsible for distributing codes, assigning tasks to machines, and monitoring their status.

- Worker nodes run *Supervisors* that are able to handle one or more worker processes on their nodes. Each supervisor handles the works assigned to it by Nimbus and starts and stops the worker processes when required. Every worker process runs a specific set of topology which consists of worker processes working around machines.

Since Apache Storm does not have the ability to manage its cluster state, it depends on Apache ZooKeeper for this purpose. ZooKeeper facilitates communication between Nimbus and Supervisors with the help of message acknowledgments, processing status, etc.

Hadoop ecosystem

Storm



Logical level

There are basically four components which are responsible for performing tasks in Apache Storm:

- **Stream** which is an unbounded pipeline of tuples. A **tuple** is component containing a named list of values or elements.
- A **spout** is a source of streams in a computation. Typically a spout reads from a queueing broker such as RabbitMQ or Kafka, but a spout can also generate its own stream or read from somewhere.
- A **bolt** processes any number of input streams and produces any number of new output streams. Most of the logic of a computation goes into bolts, such as functions, filters, streaming joins, streaming aggregations, talking to databases, and so on.
- A Storm application is designed as a **topology** in the shape of a directed acyclic graph (DAG) with spouts and bolts acting as the graph vertices. Edges on the graph are named streams and direct data from one node to another. All together, the topology, acts as a data transformation pipeline. Topologies run indefinitely when deployed which differs from a MapReduce job which must eventually end.

Hadoop ecosystem

Kafka

We will describe Apache Kafka comparing it with seemingly similar system: Apache Spark

In short:

- Apache Storm: real time message processing.
- Apache Kafka: distributed messaging system.

Although seemingly similar, Kafka and Storm have a slightly different purpose:

Storm is a scalable, fault-tolerant, real-time analytic system (something like Hadoop in realtime). It consumes data from sources (Spouts) and passes it to pipeline (Bolts). You can combine them in the topology. So Storm is basically a computation unit (aggregation, machine learning).

Kafka is a distributed message broker which can handle big amount of messages per second. It uses publish-subscribe paradigm and relies on topics and partitions. Kafka uses Zookeeper to share and save state between brokers. So Kafka is basically responsible for transferring messages from one machine to another.

The common flow of these tools may goes as follows:

```
real-time-system --> Kafka --> Storm --> NoSql --> BI
```

Hadoop ecosystem

Flume

Apache Flume is a distributed, reliable, and available software for efficiently collecting, aggregating, and moving large amounts of log data. It has a simple and flexible architecture based on streaming data flows.

Hadoop ecosystem

Sqoop

Sqoop [skup] got the name from *SQL-to-Hadoop*. As it states, Apache Sqoop is a tool designed for efficiently transferring bulk data between Apache Hadoop and structured datastores such as relational databases.

Hadoop ecosystem

ZooKeeper

Apache ZooKeeper is a service for distributed systems offering a hierarchical key-value store, which is used to provide a distributed configuration service, synchronization service, and naming registry for large distributed systems. It was a sub-project of Hadoop but is now a top-level Apache project in its own right.

Bibliography

- [FULPCBD] Piotr Fulmański, *Processing concepts for Big Data*,
  `https://fulmanski.pl/tutorials/computer-science/big-data/processing-concepts-for-big-data/`
- [HADALA] Sam R. Alapati, *Expert Hadoop Administration: Managing, Tuning, and Securing Spark, YARN, and HDFS*, Addison-Wesley, 2016,
  `https://www.informit.com/articles/article.aspx?p=2755708`
- [HBG] Apache HBase Reference Guide,
  `https://hbase.apache.org/book.html`
- [HCLI]
    - [1] The Hadoop File System shell commands list,
      `https://hadoop.apache.org/docs/current/hadoop-project-dist/hadoop-common/FileSystemShell.html`
    - [2] Sam R. Alapati, *HDFS Commands, HDFS Permissions and HDFS Storage*,
      `https://www.informit.com/articles/article.aspx?p=2755708`
    - [3] Top 10 Hadoop HDFS Commands with Examples and Usage,
      `https://data-flair.training/blogs/top-hadoop-hdfs-commands-tutorial/`
- [HHS001] HDFS Snapshots,
  `https://hadoop.apache.org/docs/current/hadoop-project-dist/hadoop-hdfs/HdfsSnapshots.html`
- [HTBS] Hadoop Tutorial for Beginners,
  `https://www.simplilearn.com/tutorials/hadoop-tutorial`
- [KAF] Apache Kafka,
  `http://dzikowski.github.io/2014/12/07/kafka/`
- [KAFFLU] Difference Between Apache Kafka and Flume,
  `https://www.educba.com/apache-kafka-vs-flume/`
- [PIGORSKT] Christopher Olston, Benjamin Reed, Utkarsh Srivastava, Ravi Kumar, Andrew Tomkins, *Pig Latin: A Not-So-Foreign Language for Data Processing*, ACM SigMod, 2008
- [PVHSIS] Divya Sistla, *Difference between Pig and Hive-The Two Key Components of Hadoop Ecosystem*,
  `https://www.dezyre.com/article/difference-between-pig-and-hive-the-two-key-components-of-hadoop-ecosystem/79`
- [SANKAR] Sandeep Karanth, *Mastering Hadoop*, Packt Publishing, 2014
- [SPAO001] Spark Overview,
  `https://spark.apache.org/docs/latest/index.html`
- [SQU] Sqoop Tutorial,
  `https://www.tutorialspoint.com/sqoop/index.htm`