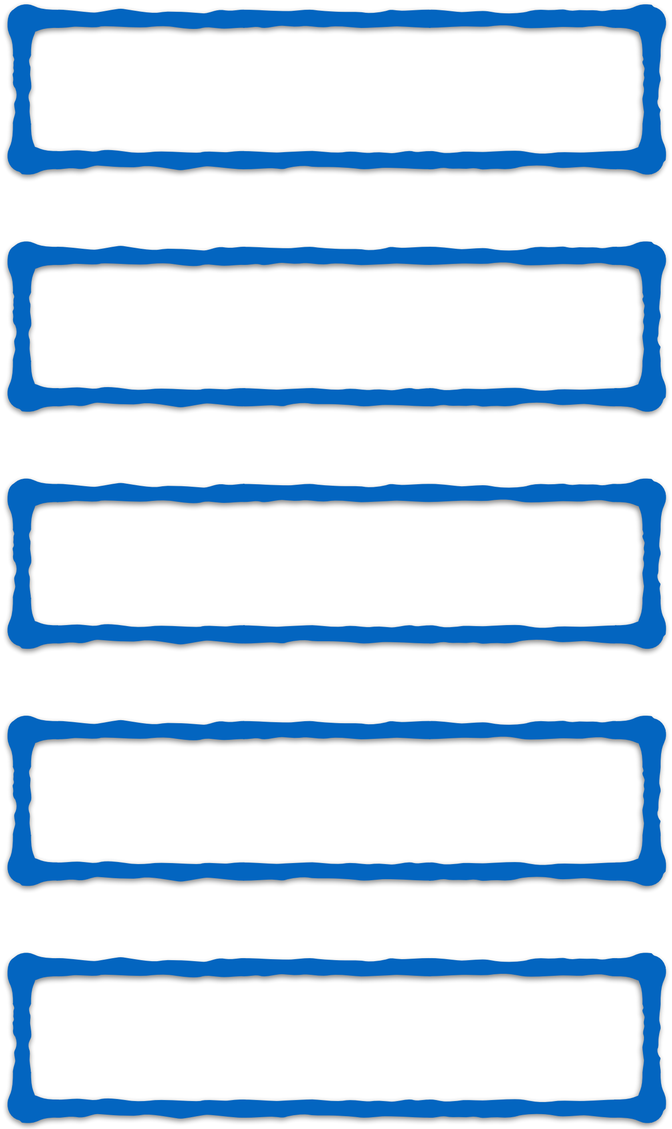
HDFS
Replication

RECAP

HDFS



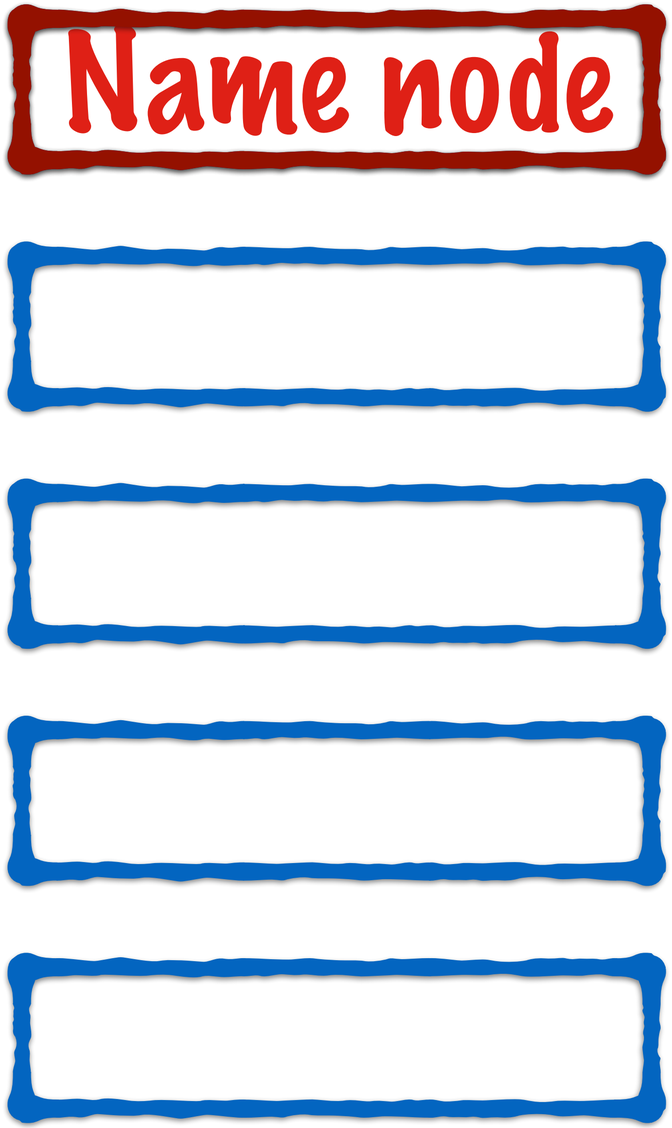
Hadoop is normally deployed on a group of machines

Cluster

Each machine in the cluster is **a node**

RECAP

HDFS

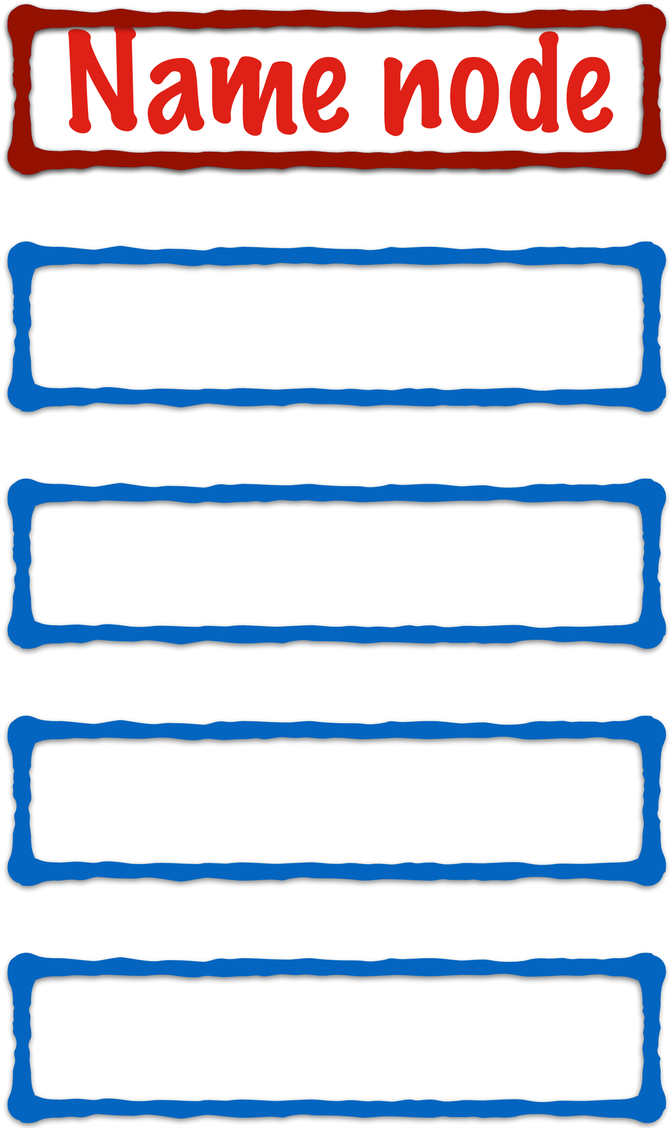


One of the nodes acts as the **master node**

This node **manages** the overall file system

RECAP

HDFS



The name node stores

1. The directory structure

2. Metadata for all the files

RECAP

HDFS

Name node

Data node 1

Data node 2

Data node 3

Data node 4

Other nodes are called data nodes

The data is physically stored on these nodes

RECAP

HDFS

Here is a large text file

next up previous contents index
Next: Dynamic indexing Up: Index construction Previous: Single-pass in-memory indexing Contents Index

Distributed indexing

Collections are often so large that we cannot perform index construction efficiently on a single machine. This is particularly true of the World Wide Web for which we need large computer clusters [*]to construct any reasonably sized web index. Web search engines, therefore, use distributed indexing algorithms for index construction. The result of the construction process is a distributed index that is partitioned across several machines - either according to term or according to document. In this section, we describe distributed indexing for a
term-partitioned index . Most large search engines prefer a document-partitioned index (which can be easily generated from a term-partitioned index). We discuss this topic further in Section 20.3 (page [*]).

The distributed index construction method we describe in this section is an application of MapReduce , a general architecture for distributed computing. MapReduce is designed for large computer clusters. The point of a cluster is to solve large computing problems on cheap commodity machines or nodes that are built from standard parts (processor, memory, disk) as opposed to on a supercomputer with specialized hardware. Although hundreds or thousands of machines are available in such clusters, individual machines can fail at any time. One requirement for robust distributed indexing is, therefore, that we divide the work up into chunks that we can easily assign and - in case of failure - reassign. A master node directs the process of assigning and reassigning tasks to individual worker nodes.

The map and reduce phases of MapReduce split up the computing job into chunks that standard machines can process in a short time. The various steps of MapReduce are shown in Figure 4.5 and an example on a collection consisting of two documents is shown in Figure 4.6 . First, the input data, in our case a collection of web pages, are split into n splits where the size of the split is chosen to ensure that the work can be distributed evenly (chunks should not be too large) and efficiently (the total number of chunks we need to manage should not be too large); 16 or 64 MB are good sizes in distributed indexing. Splits are not preassigned to machines, but are instead assigned by the master node on an ongoing basis: As a machine finishes processing one split, it is assigned the next one. If a machine dies or becomes a laggard due to hardware problems, the split it is working on is simply reassigned to another machine.

Figure 4.5: An example of distributed indexing with MapReduce. Adapted from Dean and Ghemawat (2004).
\includegraphics[width=11.5cm]{art/mapreduce2.eps}
In general, MapReduce breaks a large computing problem into smaller parts by recasting it in terms of manipulation of key-value pairs . For indexing, a key-value pair has the form (termID,docID). In distributed indexing, the mapping from terms to termIDs is also distributed and therefore more complex than in single-machine indexing. A simple solution is to maintain a (perhaps precomputed) mapping for frequent terms that is copied to all nodes and to use terms directly (instead of termIDs) for infrequent terms. We do not address this problem here and assume that all nodes share a consistent term $\rightarrow$ termID mapping.

The map phase of MapReduce consists of mapping splits of the input data to key-value pairs. This is the same parsing task we also encountered in BSBI and SPIMI, and we therefore call the machines that execute the map phase parsers . Each parser writes its output to local intermediate files, the segment files (shown as \fbox{a-f\medstrut} \fbox{g-p\medstrut} \fbox{q-z\medstrut} in Figure 4.5).

For the reduce phase , we want all values for a given key to be stored close together, so that they can be read and processed quickly. This is achieved by partitioning the keys into j term partitions and having the parsers write key-value pairs for each term partition into a separate segment file. In Figure 4.5 , the term partitions are according to first letter: a-f, g-p, q-z, and $j=3$. (We chose these key ranges for ease of exposition. In general, key ranges need not correspond to contiguous terms or termIDs.) The term partitions are defined by the person who operates the indexing system (Exercise 4.6). The parsers then write corresponding segment files, one for each term partition. Each term partition thus corresponds to r segments files, where r is the number of parsers. For instance, Figure 4.5 shows three a-f segment files of the a-f partition, corresponding to the three parsers shown in the figure.

Collecting all values (here: docIDs) for a given key (here: termID) into one list is the task of the inverters in the reduce phase. The

Let's see how this file is stored in HDFS

RECAP

HDFS

Block 1

Block 2

Block 3

Block 4

Block 5

Block 6

Block 7

Block 8

First the file is broken into blocks of size 128 MB

RECAP

HDFS

Block 1

Block 2

Block 3

Block 4

Block 5

Block 6

Block 7

Block 8

First the file is broken into **blocks of size 128 MB**

This size is chosen to minimize the time to seek to the block on the disk

RECAP

HDFS

Block 1

Block 2

Block 3

Block 4

Block 5

Block 6

Block 7

Block 8

These blocks are then stored across the data nodes

RECAP

HDFS

Data node 1

Block 1 | Block 2

Data node 3

Block 5 | Block 6

Data node 2

Block 3 | Block 4

Data node 4

Block 7 | Block 8

Name node

The name node stores metadata

RECAP

HDFS

Block locations for each file are stored in the name node

Name node

File 1	Block 1	DN 1
File 1	Block 2	DN 1
File 1	Block 3	DN 2
File 1	Block 4	DN 2
File 1	Block 5	DN 3

RECAP

HDFS

A file is read using

1. The **metadata** in name node
2. The **blocks** in the data nodes

Name node

File 1	Block 1	DN 1
File 1	Block 2	DN 1
File 1	Block 3	DN 2
File 1	Block 4	DN 2
File 1	Block 5	DN 3

RECAP

HDFS

Data node 1

Block 1 | Block 2

Data node 3

Block 5 | Block 6

Name node

File 1	Block 1	DN 1
File 1	Block 2	DN 1
File 1	Block 3	DN 2
File 1	Block 4	DN 2
File 1	Block 5	DN 3

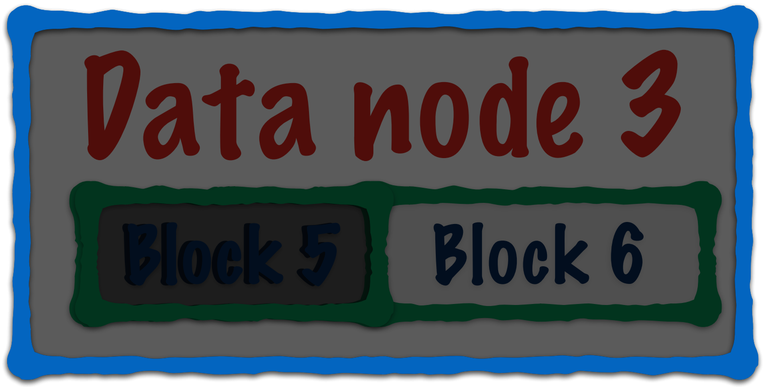
Data node 2

Data node 4

What if one of the blocks gets corrupted?

RECAP

HDFS



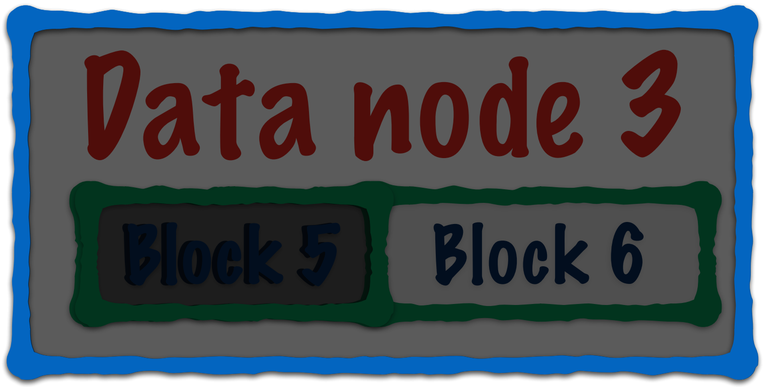
Name node

File 1	Block 1	DN 1
File 1	Block 2	DN 1
File 1	Block 3	DN 2
File 1	Block 4	DN 2
File 1	Block 5	DN 3

Or one of the data nodes crashes?

RECAP

HDFS



Name node

File 1	Block 1	DN 1
File 1	Block 2	DN 1
File 1	Block 3	DN 2
File 1	Block 4	DN 2
File 1	Block 5	DN 3

This is one of the key challenges in distributed storage

HDFS

Data node 1

Block 1

Data node 3

Block 6

Data node 2

Block 3

Block 4

Data node 4

Block 7

Block 8

You can define a Replication factor in HDFS

Name node

File 1	Block 1	DN 1
File 1	Block 2	DN 1
File 1	Block 3	DN 2
File 1	Block 4	DN 2
File 1	Block 5	DN 3

HDFS Replication

This replication factor is defined in the configuration file hdfs-site.xml

Name node

File 1	Block 1	DN 1
File 1	Block 2	DN 1
File 1	Block 3	DN 2
File 1	Block 4	DN 2
File 1	Block 5	DN 3

HDFS Replication

```
<configuration>
<property>
<name>dfs.replication</name>
<value>1</value>
</property>
</configuration>
```

This replication factor is defined in the configuration file hdfs-site.xml

HDFS Replication

```
<configuration>
<property>
<name>dfs.replication</name>
<value>1</value>
</property>
</configuration>
```

In Pseudo-distributed or standalone mode, the replication factor is normally set to 1

HDFS Replication

```
<configuration>
<property>
<name>dfs.replication</name>
<value>3</value>
</property>
</configuration>
```

In a fully-distributed Hadoop setup, the replication factor is usually set to 3 by default

HDFS Replication

```
<configuration>
<property>
<name>dfs.replication</name>
<value>3</value>
</property>
</configuration>
```

This means 3 copies of a block are stored when it's written to HDFS

HDFS Replication

The replica locations are chosen to

maximize redundancy

while **minimizing** write **bandwidth**

Let's parse that

HDFS Replication

maximize redundancy

In case of failure of a block/node

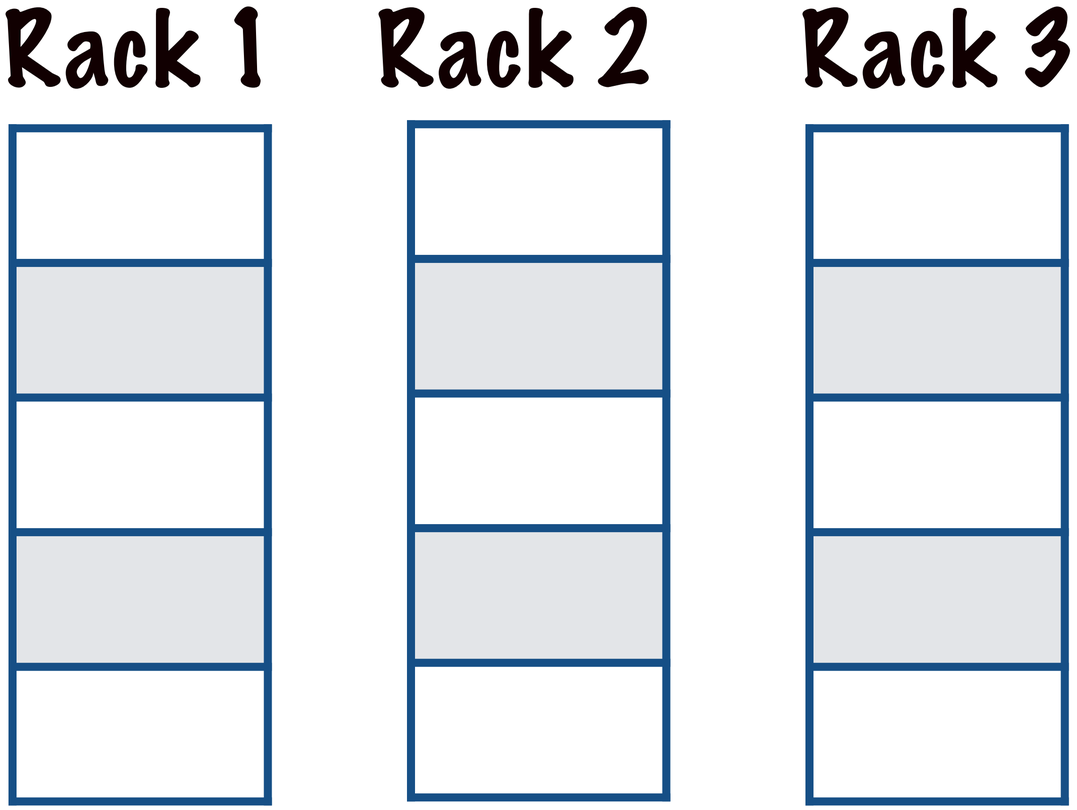
we want to maximize the chances
of being able to reconstruct the file

HDFS Replication

maximize redundancy

In a datacenter

servers are arranged in racks

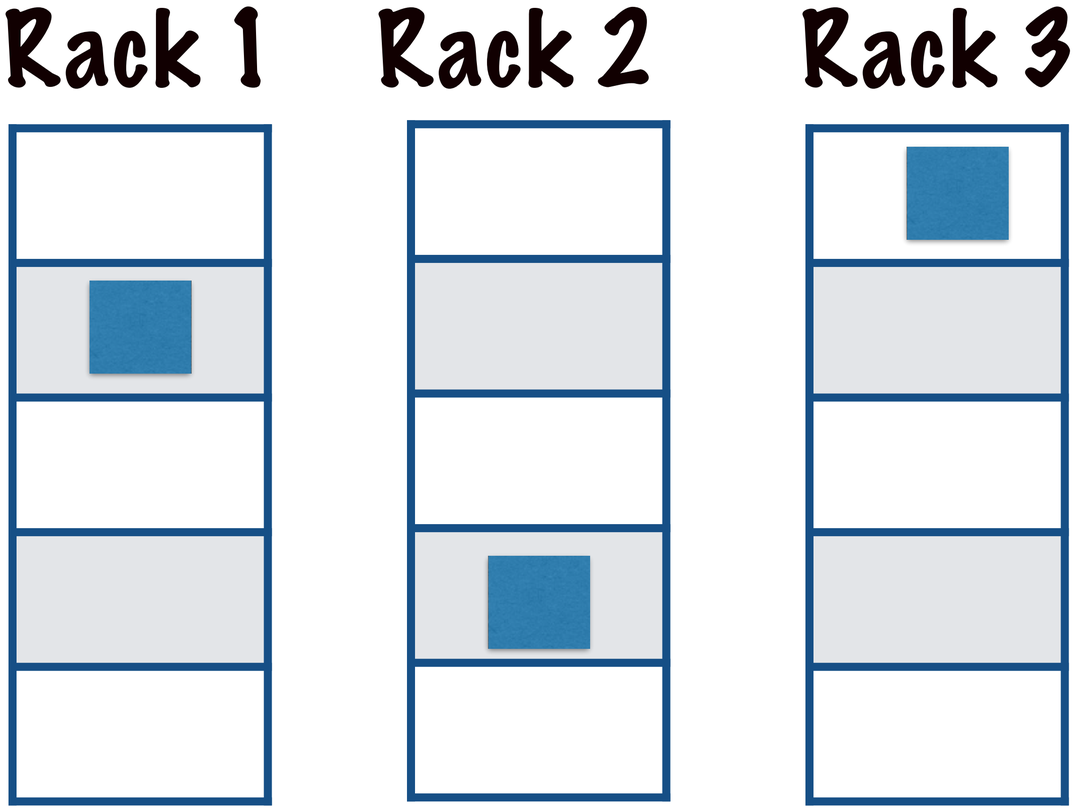


HDFS Replication

maximize redundancy

To maximize redundancy

all replicas
should be stored
"far away" i.e.
different racks



HDFS Replication

The replica locations are chosen to

maximize redundancy

while **minimizing** write **bandwidth**

Let's parse that

HDFS Replication

The replica locations are chosen to

maximize redundancy

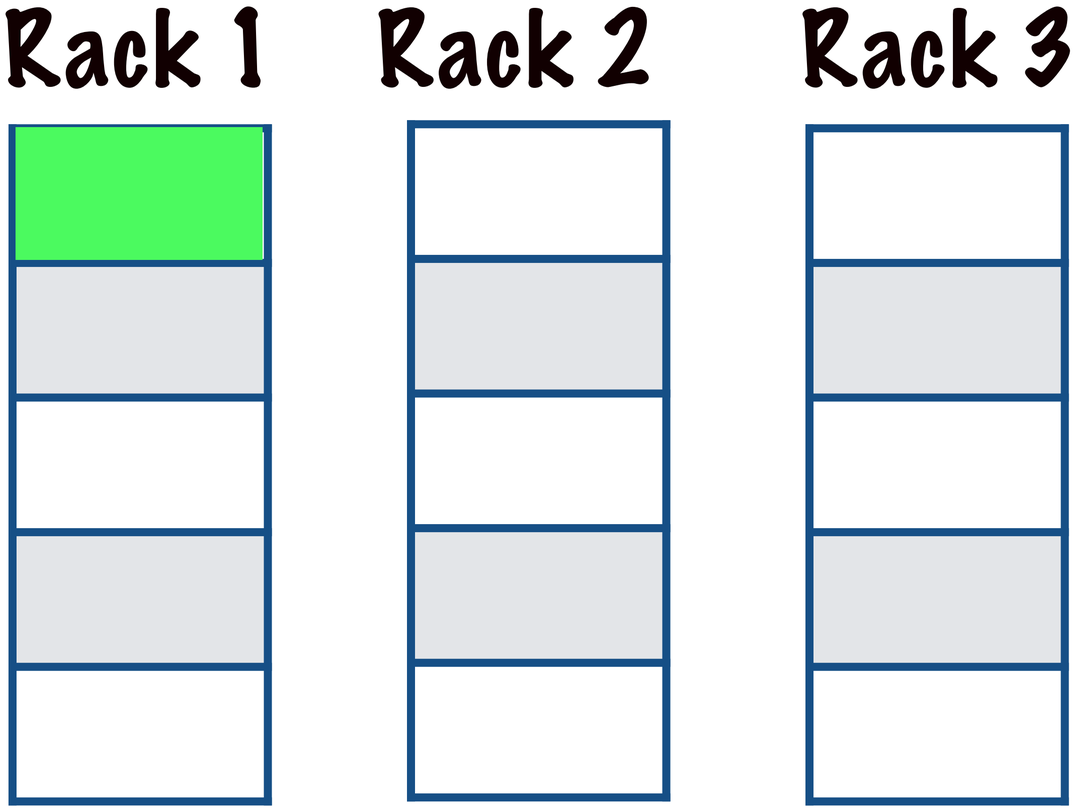
while minimizing write bandwidth

Let's parse that

HDFS Replication

minimizing write bandwidth

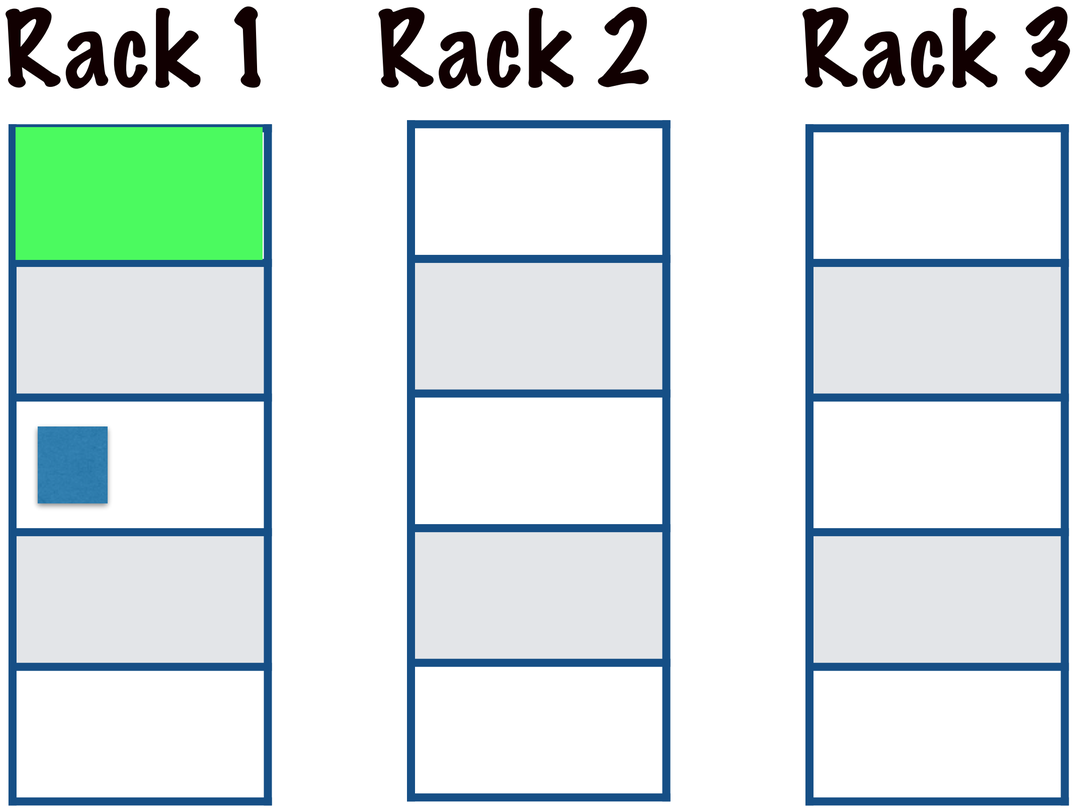
When a file is written to HDFS, a single node runs the replication pipeline



HDFS Replication

minimizing write bandwidth

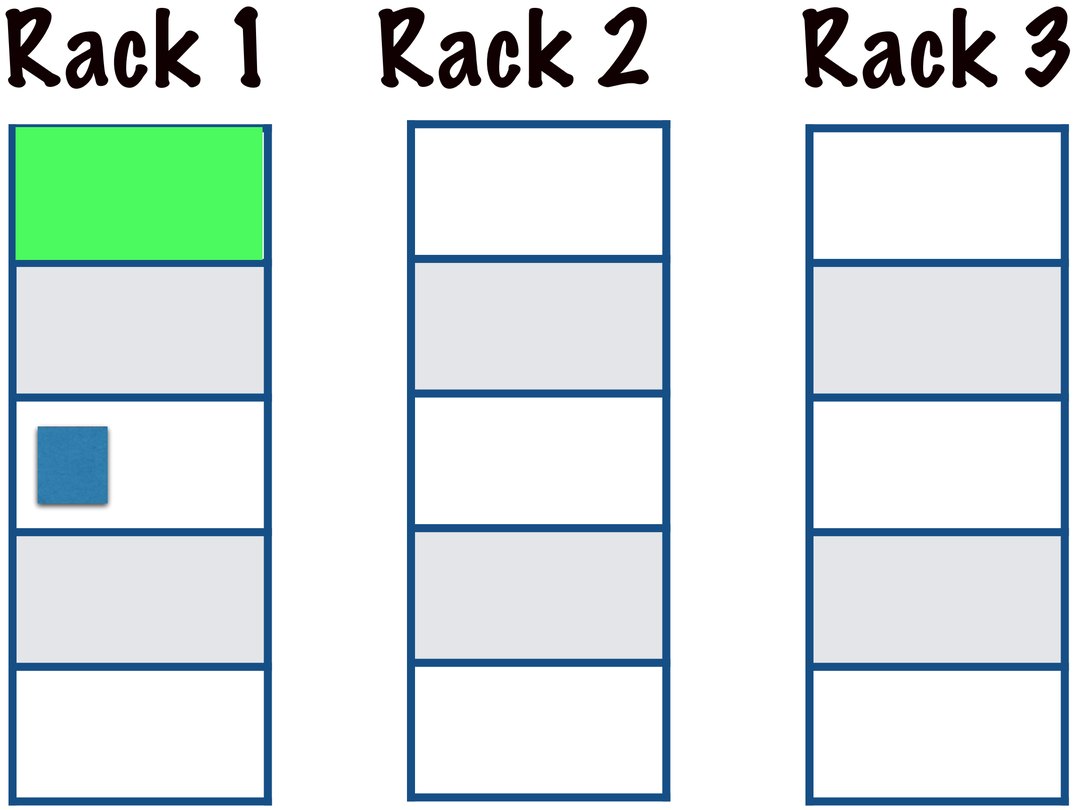
A node is chosen to store the first replica



HDFS Replication

minimizing write bandwidth

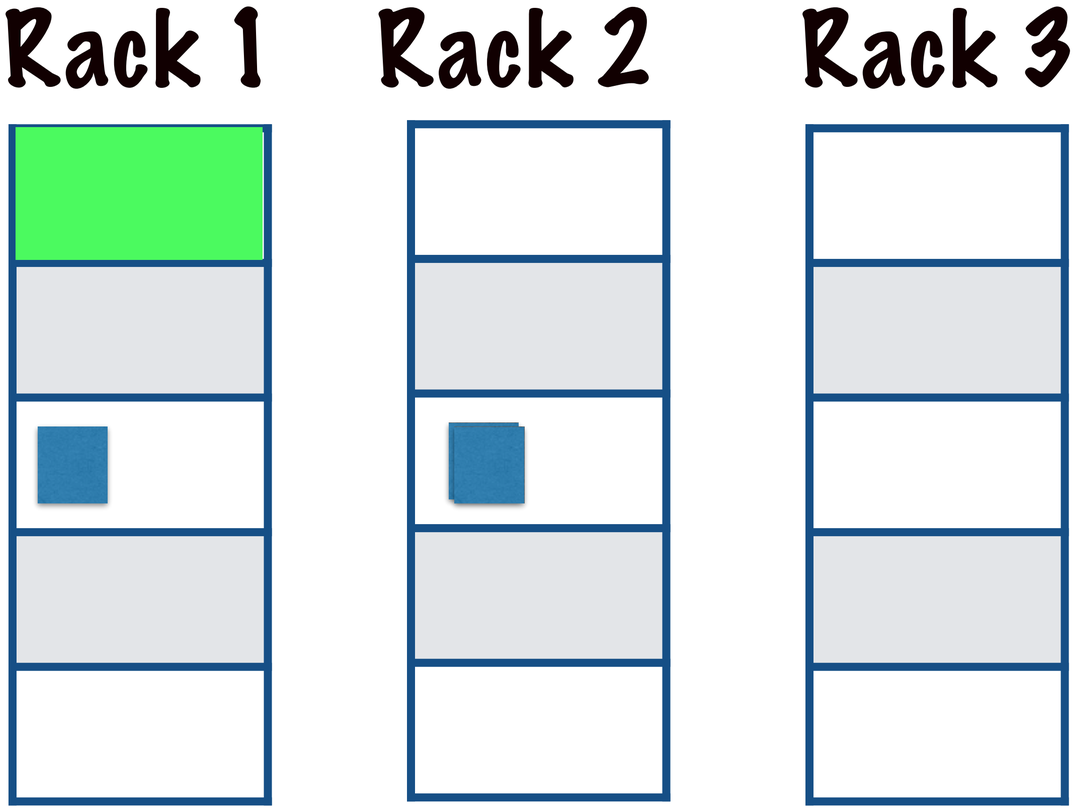
This node forwards the data to the location chosen for the second replica



HDFS Replication

minimizing write bandwidth

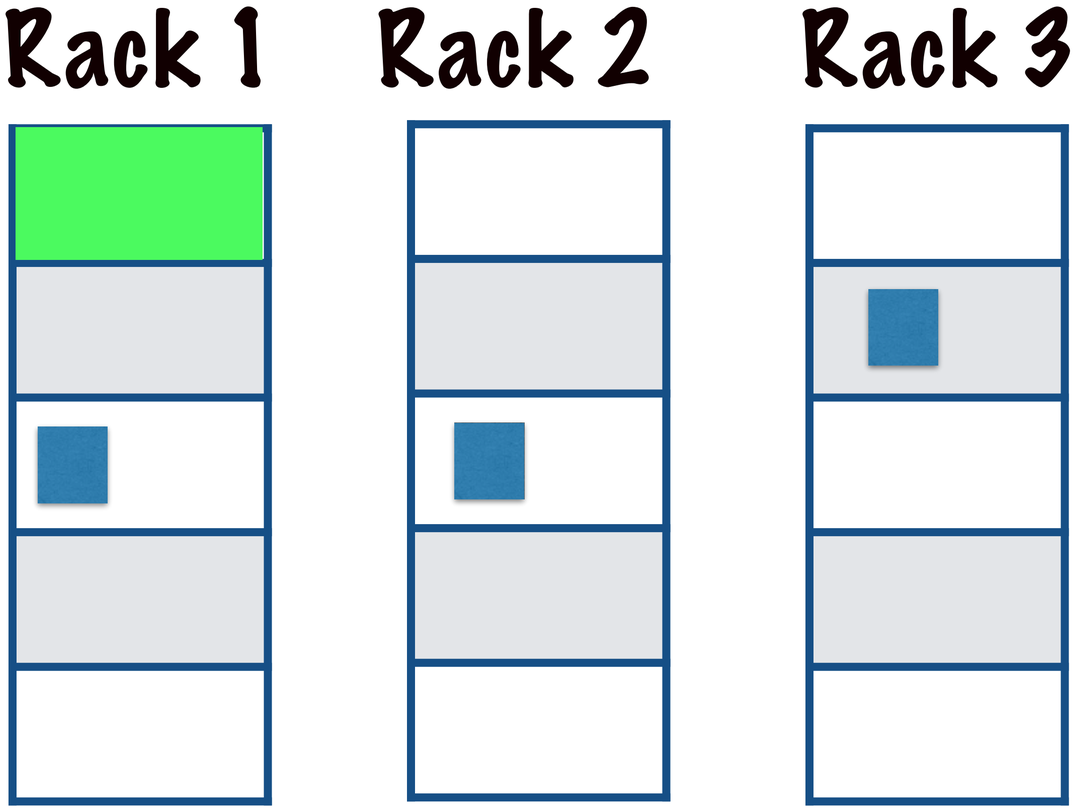
Then it is forwarded further



HDFS Replication

minimizing write bandwidth

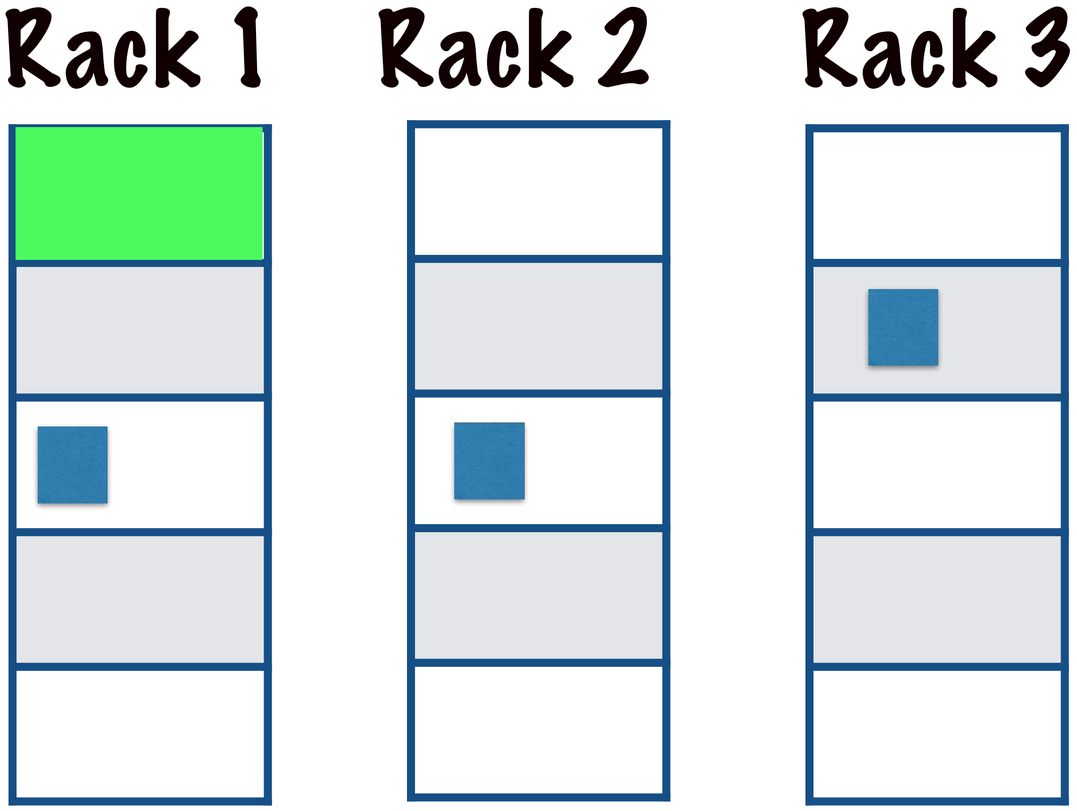
All this consumes network bandwidth



HDFS Replication

minimizing write bandwidth

More the number of racks to traverse, more the bandwidth consumed



HDFS Replication

The replica locations are chosen to

maximize redundancy

while minimizing write bandwidth

HDFS Replication

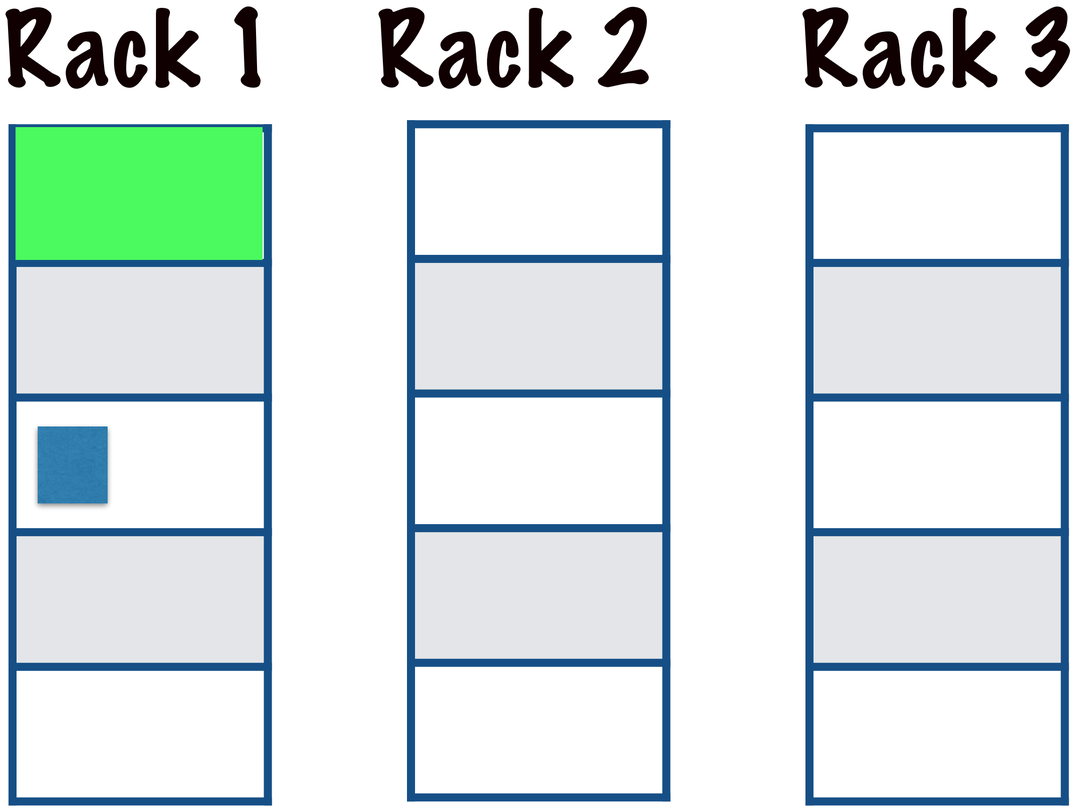
The replica locations are chosen to maximize redundancy while minimizing write bandwidth

Hadoop has a default strategy for storing 3 replicas

HDFS Replication

Default Strategy

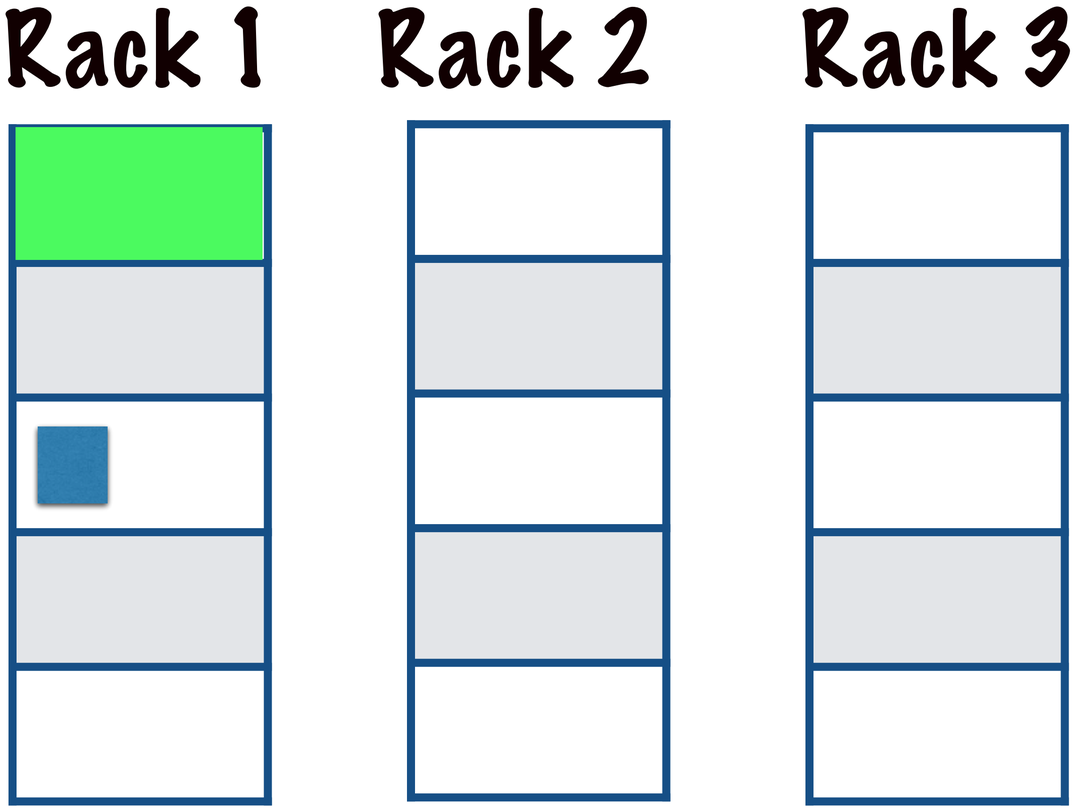
The first replica location is chosen at random



HDFS Replication

Default Strategy

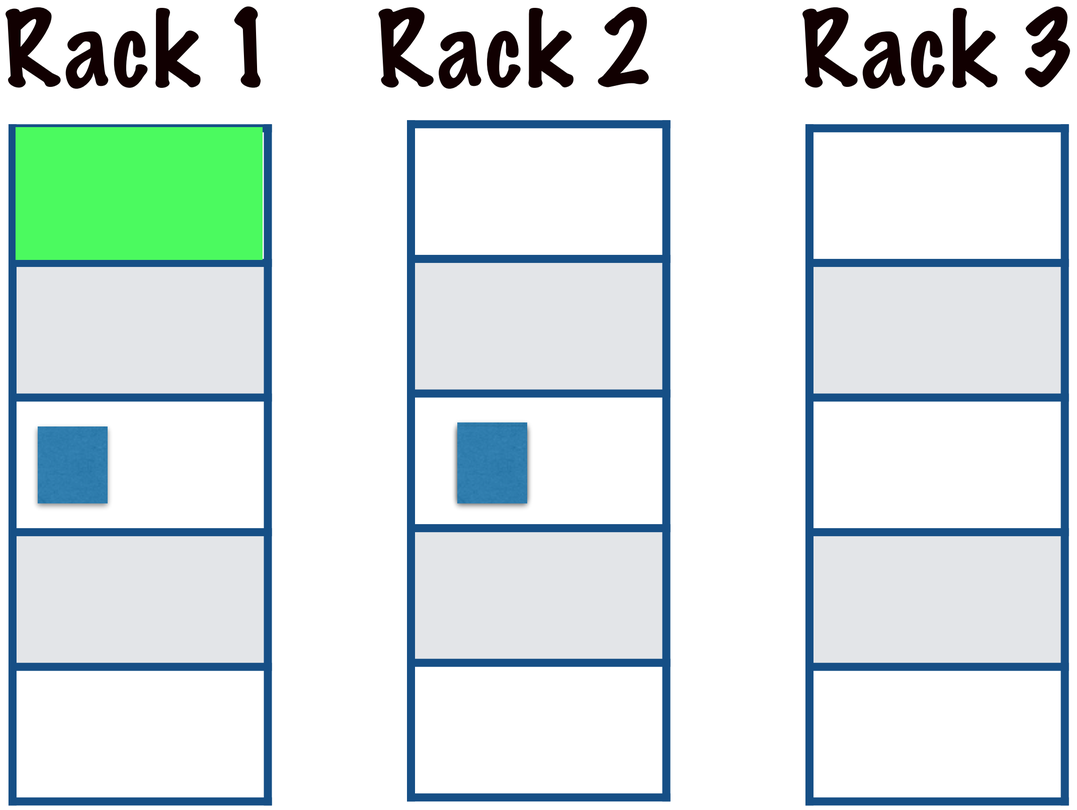
The second replica is stored on a different rack



HDFS Replication

Default Strategy

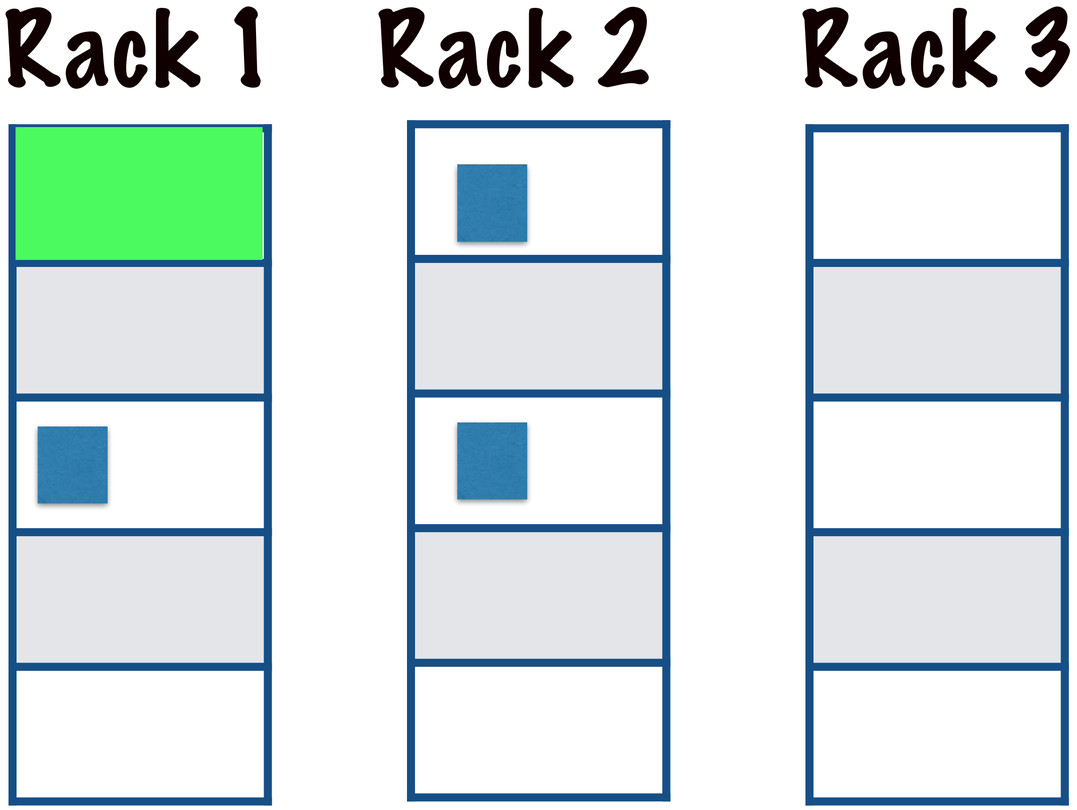
The second replica is stored on a different rack



HDFS Replication

Default Strategy

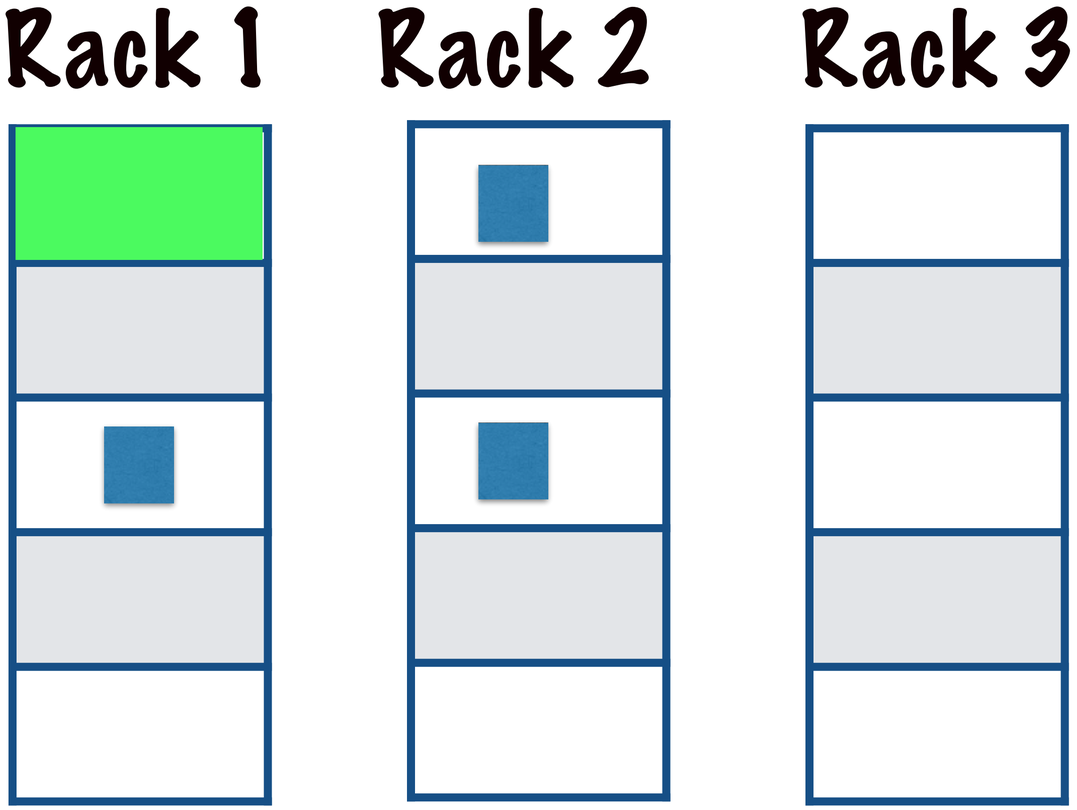
The third replica is stored on the same rack as the second, but a different node



HDFS Replication

Default Strategy

This gives a good balance between redundancy and bandwidth

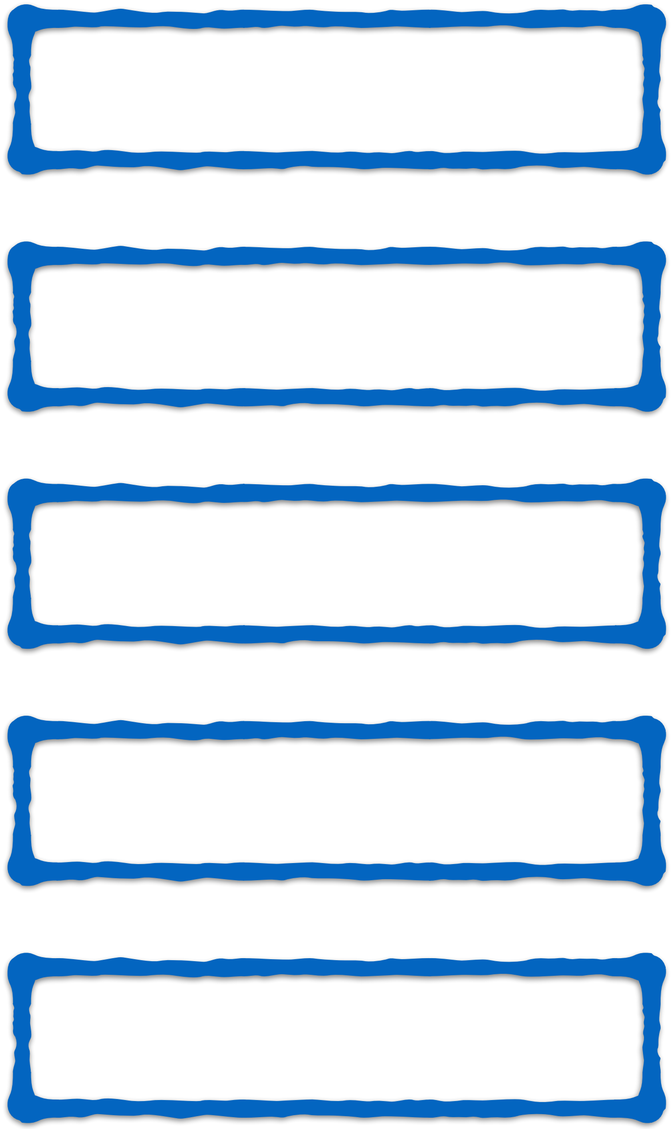


HDFS

Namenode failure management

RECAP

HDFS



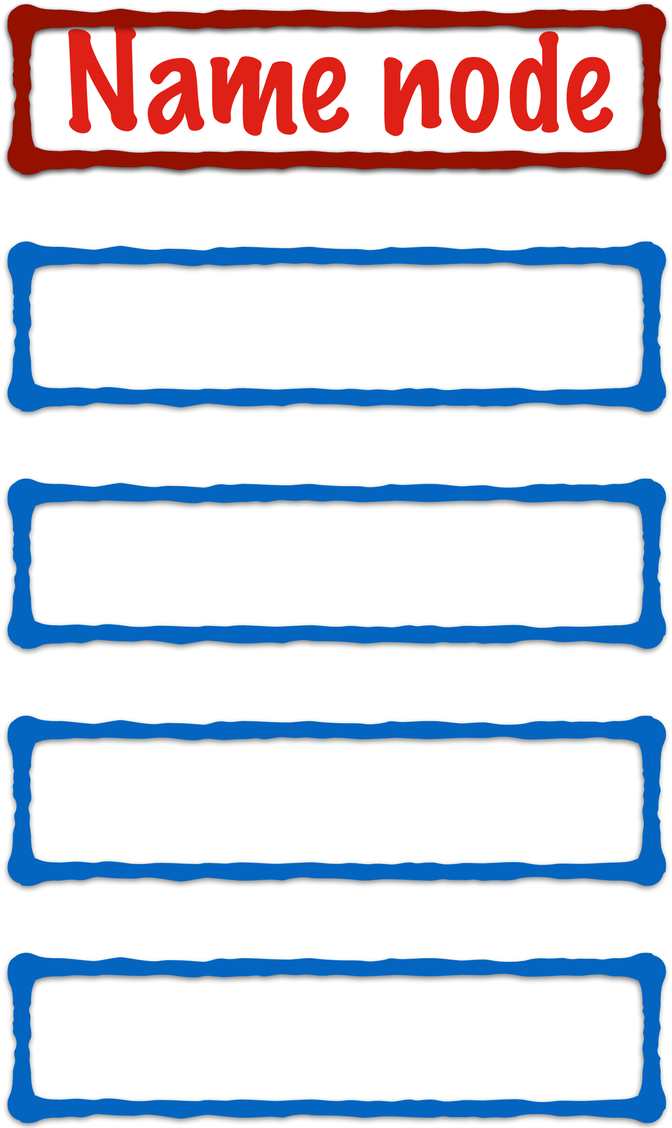
Hadoop is normally deployed on a group of machines

Cluster

Each machine in the cluster is **a node**

RECAP

HDFS

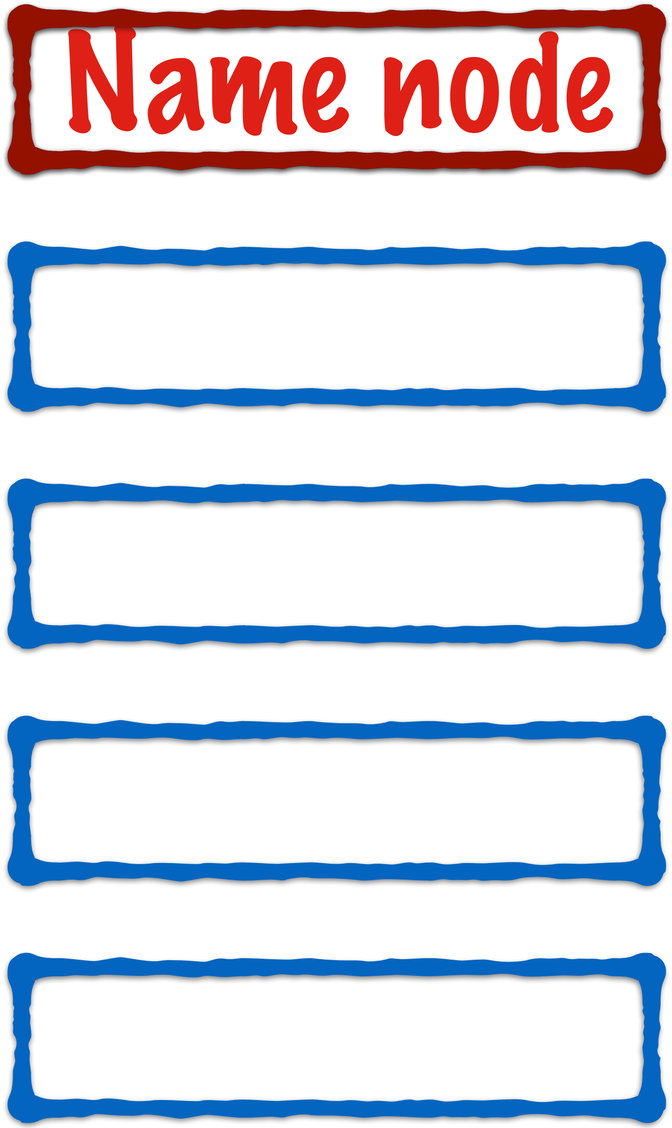


One of the nodes acts as the **master node**

This node **manages** the overall file system

RECAP

HDFS



The name node stores

1. The directory structure

2. Metadata for all the files

RECAP

HDFS

Name node

Data node 1

Data node 2

Data node 3

Data node 4

Other nodes are called data nodes

The data is physically stored on these nodes

RECAP

HDFS

Here is a large text file

next up previous contents index
Next: Dynamic indexing Up: Index construction Previous: Single-pass in-memory indexing Contents Index

Distributed indexing

Collections are often so large that we cannot perform index construction efficiently on a single machine. This is particularly true of the World Wide Web for which we need large computer clusters [*]to construct any reasonably sized web index. Web search engines, therefore, use distributed indexing algorithms for index construction. The result of the construction process is a distributed index that is partitioned across several machines - either according to term or according to document. In this section, we describe distributed indexing for a
term-partitioned index . Most large search engines prefer a document-partitioned index (which can be easily generated from a term-partitioned index). We discuss this topic further in Section 20.3 (page [*]).

The distributed index construction method we describe in this section is an application of MapReduce , a general architecture for distributed computing. MapReduce is designed for large computer clusters. The point of a cluster is to solve large computing problems on cheap commodity machines or nodes that are built from standard parts (processor, memory, disk) as opposed to on a supercomputer with specialized hardware. Although hundreds or thousands of machines are available in such clusters, individual machines can fail at any time. One requirement for robust distributed indexing is, therefore, that we divide the work up into chunks that we can easily assign and - in case of failure - reassign. A master node directs the process of assigning and reassigning tasks to individual worker nodes.

The map and reduce phases of MapReduce split up the computing job into chunks that standard machines can process in a short time. The various steps of MapReduce are shown in Figure 4.5 and an example on a collection consisting of two documents is shown in Figure 4.6 . First, the input data, in our case a collection of web pages, are split into n splits where the size of the split is chosen to ensure that the work can be distributed evenly (chunks should not be too large) and efficiently (the total number of chunks we need to manage should not be too large); 16 or 64 MB are good sizes in distributed indexing. Splits are not preassigned to machines, but are instead assigned by the master node on an ongoing basis: As a machine finishes processing one split, it is assigned the next one. If a machine dies or becomes a laggard due to hardware problems, the split it is working on is simply reassigned to another machine.

Figure 4.5: An example of distributed indexing with MapReduce. Adapted from Dean and Ghemawat (2004).
\includegraphics[width=11.5cm]{art/mapreduce2.eps}
In general, MapReduce breaks a large computing problem into smaller parts by recasting it in terms of manipulation of key-value pairs . For indexing, a key-value pair has the form (termID,docID). In distributed indexing, the mapping from terms to termIDs is also distributed and therefore more complex than in single-machine indexing. A simple solution is to maintain a (perhaps precomputed) mapping for frequent terms that is copied to all nodes and to use terms directly (instead of termIDs) for infrequent terms. We do not address this problem here and assume that all nodes share a consistent term $\rightarrow$ termID mapping.

The map phase of MapReduce consists of mapping splits of the input data to key-value pairs. This is the same parsing task we also encountered in BSBI and SPIMI, and we therefore call the machines that execute the map phase parsers . Each parser writes its output to local intermediate files, the segment files (shown as \fbox{a-f\medstrut} \fbox{g-p\medstrut} \fbox{q-z\medstrut} in Figure 4.5).

For the reduce phase , we want all values for a given key to be stored close together, so that they can be read and processed quickly. This is achieved by partitioning the keys into j term partitions and having the parsers write key-value pairs for each term partition into a separate segment file. In Figure 4.5 , the term partitions are according to first letter: a-f, g-p, q-z, and $j=3$. (We chose these key ranges for ease of exposition. In general, key ranges need not correspond to contiguous terms or termIDs.) The term partitions are defined by the person who operates the indexing system (Exercise 4.6). The parsers then write corresponding segment files, one for each term partition. Each term partition thus corresponds to r segments files, where r is the number of parsers. For instance, Figure 4.5 shows three a-f segment files of the a-f partition, corresponding to the three parsers shown in the figure.

Collecting all values (here: docIDs) for a given key (here: termID) into one list is the task of the inverters in the reduce phase. The

Let's see how this file is stored in HDFS

RECAP

HDFS

Block 1

Block 2

Block 3

Block 4

Block 5

Block 6

Block 7

Block 8

First the file is broken into blocks of size 128 MB

RECAP

HDFS

Block 1

Block 2

Block 3

Block 4

Block 5

Block 6

Block 7

Block 8

These blocks are then stored across the data nodes

RECAP

HDFS

Data node 1

Block 1 | Block 2

Data node 3

Block 5 | Block 6

Data node 2

Block 3 | Block 4

Data node 4

Block 7 | Block 8

Name node

The name node stores metadata

RECAP

HDFS

Block locations for each file are stored in the name node

Name node

File 1	Block 1	DN 1
File 1	Block 2	DN 1
File 1	Block 3	DN 2
File 1	Block 4	DN 2
File 1	Block 5	DN 3

RECAP

HDFS

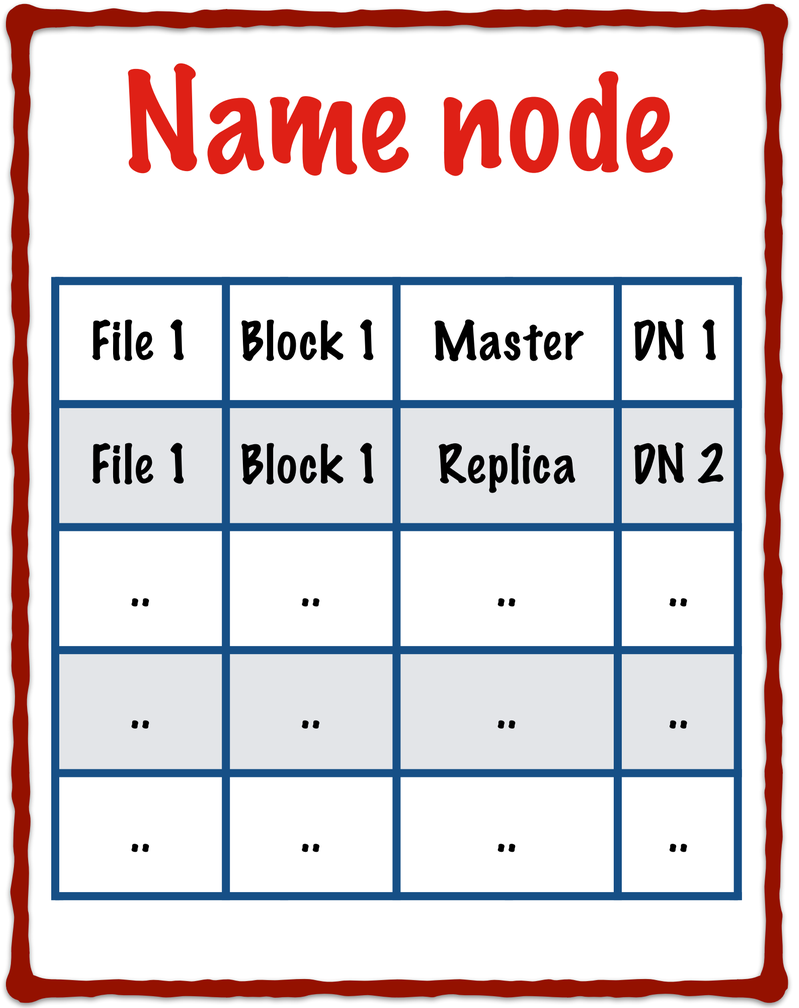
If a replication factor is set, then the replica locations are also stored

Name node

File 1	Block 1	Master	DN 1
File 1	Block 1	Replica	DN 2
..	..	..	..
..	..	..	..
..	..	..	..

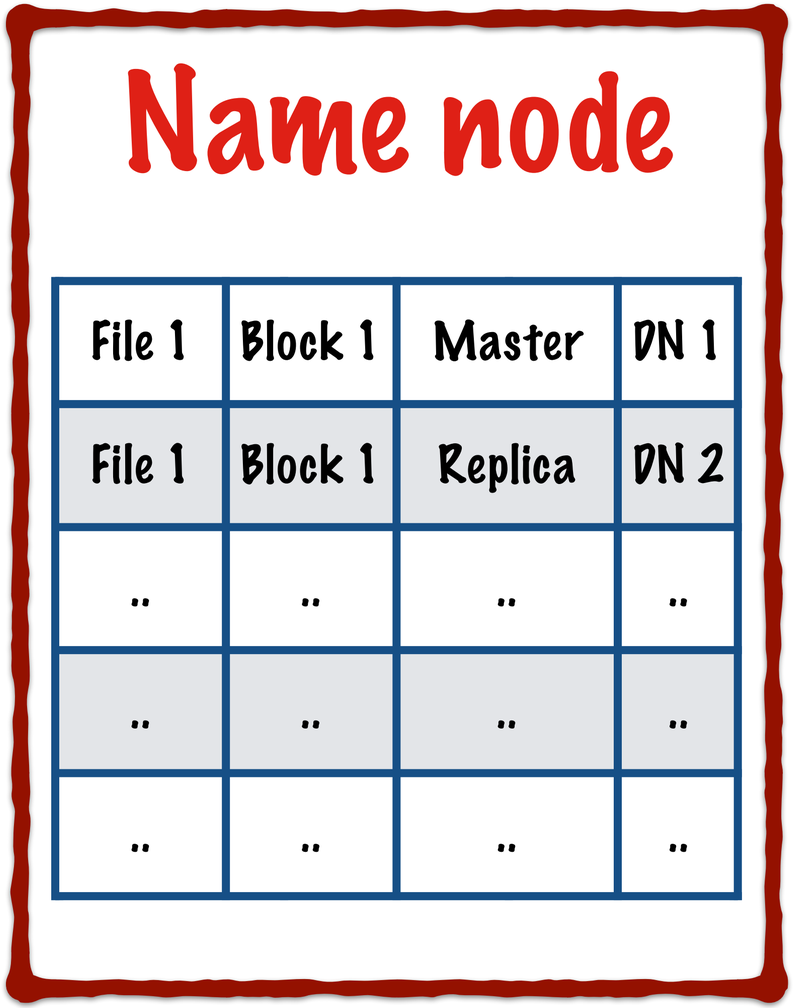
Name node

The name node is the heart of HDFS



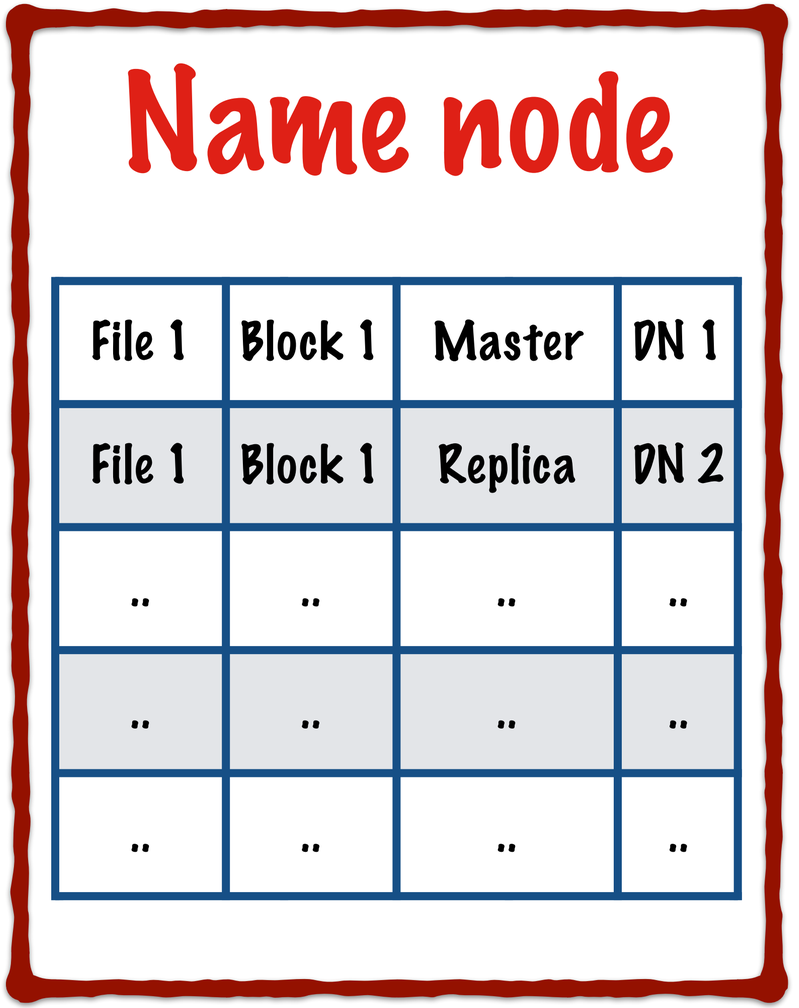
Name node

It stores all the metadata of files such as name, owner, permissions etc



Name node

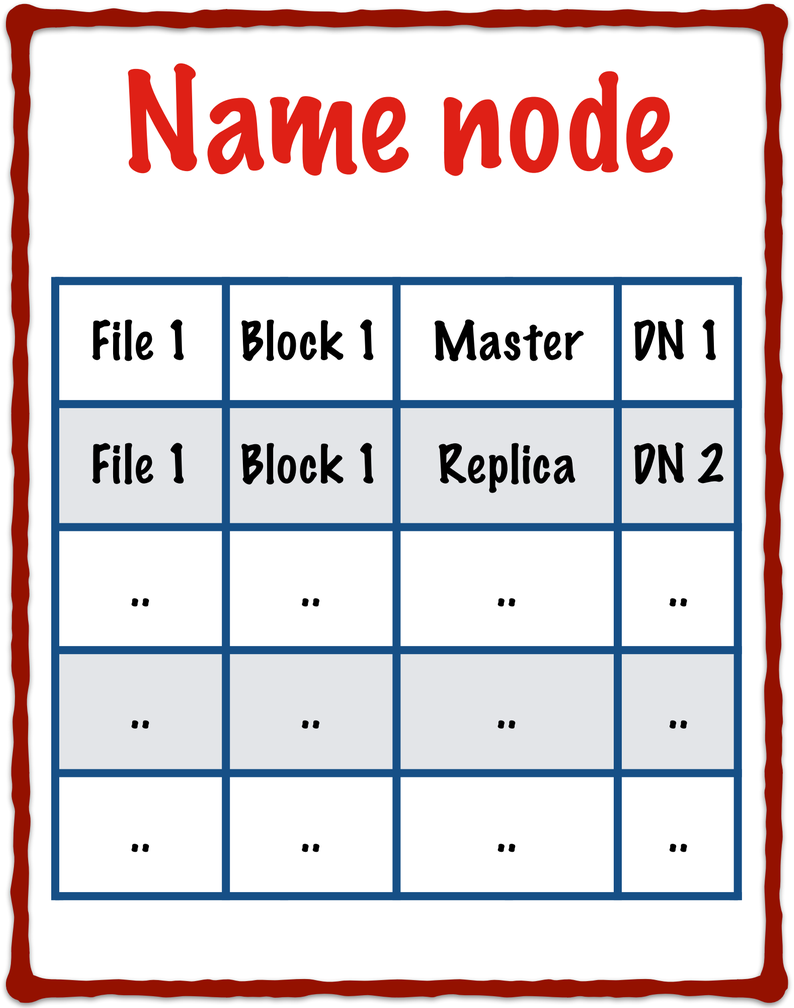
It knows which data nodes the blocks and their replicas of a file are stored on



Name node

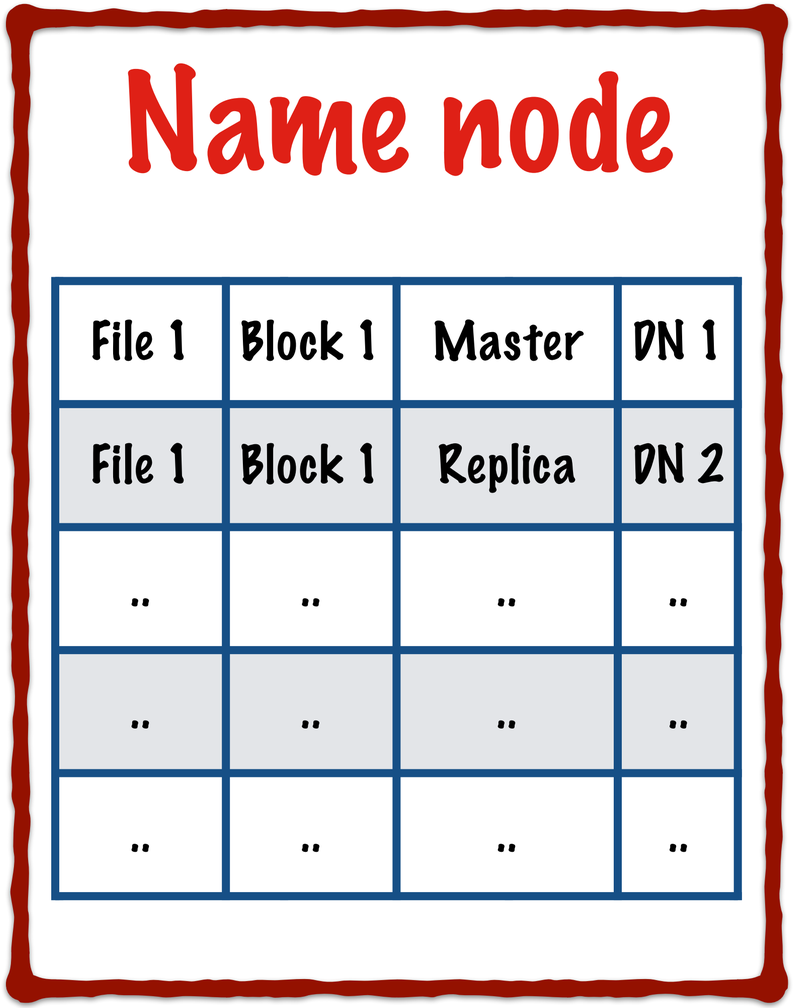
File metadata, directory structure etc are persistently stored on the Name node

ie. on disk



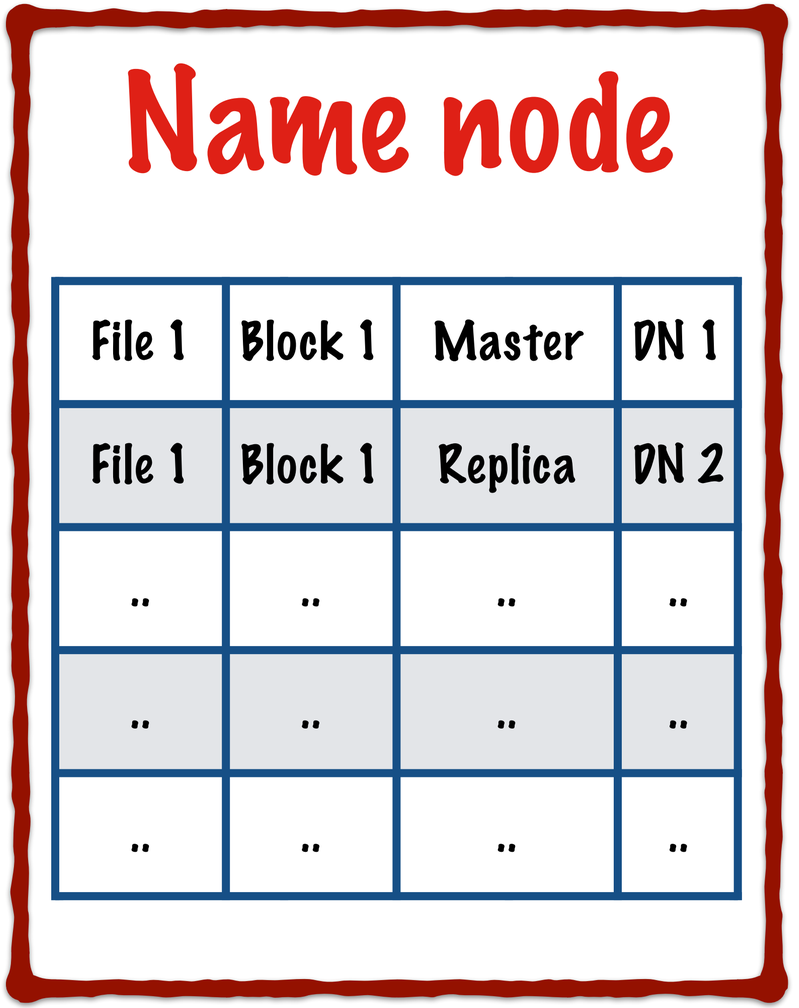
Name node

Block locations on the other hand are stored in memory



Name node

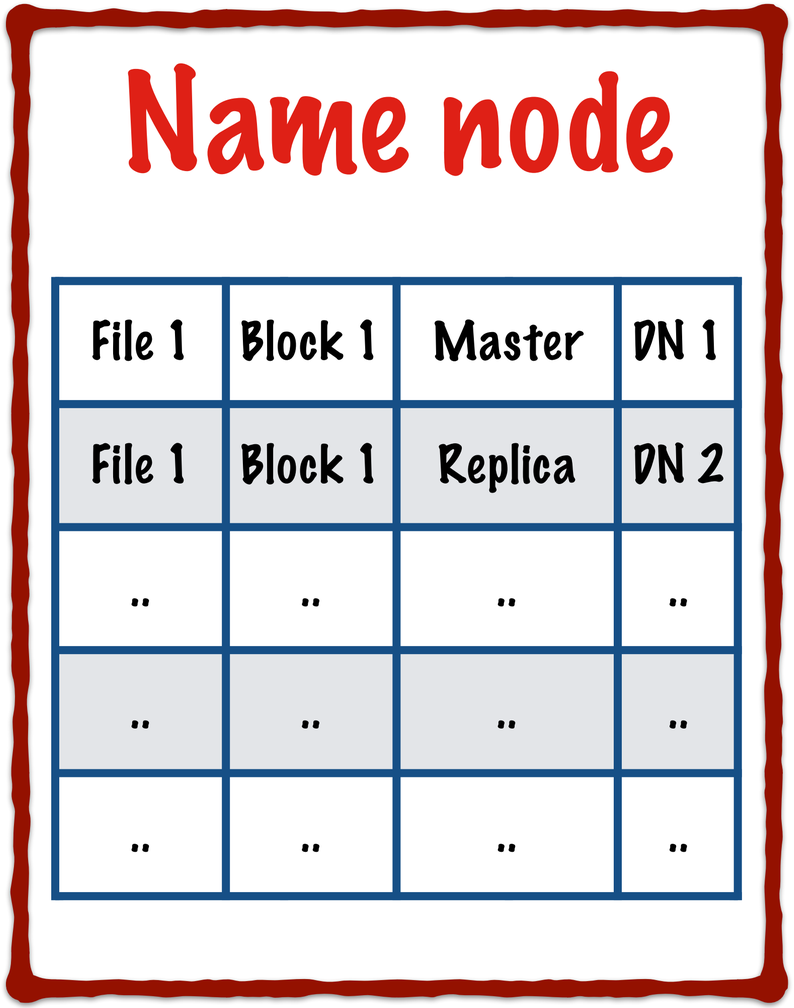
If the name node fails,
all the files are lost,
because there is no way
to reconstruct them



Name node

Even though the data is still physically present on the data nodes

Without the name node, that's useless



Name node

Hadoop makes the name node failure resistant in 2 ways

1. Back up all the files that store filesystem metadata

2. Maintain a secondary name node

Name node

1. Back up filesystem metadata

There are 2 files that store the filesystem metadata

`fsimage` State of the filesystem at start up

`edits`

Name node

1. Back up filesystem metadata

There are 2 files that store the filesystem metadata

fsimage This is a snapshot that is loaded into memory

edits

Name node

1. Back up filesystem metadata

There are 2 files that store the filesystem metadata

fsimage

edits

All changes to the file system are then made in-memory

Name node

1. Back up filesystem metadata

There are 2 files that store the filesystem metadata

fsimage

edits

The edits file stores a log of all in-memory edits to the filesystem

Name node

1. Back up filesystem metadata

`fsimage`

`edits`

These 2 files together make up the current state of the file system

Name node

1. Back up filesystem metadata

`fsimage`

`edits`

If the name node fails, the current state can be reconstructed using these 2 files

Name node

1. Back up filesystem metadata

`fsimage`

`edits`

These files are usually backed up on the local filesystem of the name node

Name node

1. Back up filesystem metadata

`fsimage`
`edits`

They can also be backed up to a remote drive

Name node

1. Back up filesystem metadata

You can specify the backup location by setting the property

`dfs.namenode.name.dir`

in hdfs-site.xml

Name node

1. Back up filesystem metadata

`dfs.namenode.name.dir`

Set this property using a comma separated list of paths, preferably on different hosts

The metadata will be backed up in each of the specified directories

Name node

1. Back up filesystem metadata

`fsimage`

`edits`

Merging these 2 files is very compute heavy, so bringing a system back online could take a long time

Name node

1. Back up filesystem metadata

`fsimage`
`edits`

This is where the Secondary name node comes in

Name node

Hadoop makes the name node failure resistant in 2 ways

1. Back up all the files that store filesystem metadata

2. Maintain a secondary name node

Name node

2. Maintain a secondary name node

`fsimage`
`edits`

The secondary name node maintains a merged fsimage

Name node

2. Maintain a secondary name node

It periodically merges the fsimage and edits files and copies it to the name node

The merge and copy process is called checkpointing

Name node

2. Maintain a secondary name node

The secondary name node can be made to take the place of name node in case of failure

Name node

2. Maintain a secondary name node

You can set the checkpoint frequency using the properties

`dfs.namenode.checkpoint.period`

`dfs.namenode.checkpoint.txns`

`dfs.namenode.checkpoint.check.period`

in hdfs-site.xml

Name node

2. Maintain a secondary name node

`dfs.namenode.checkpoint.period`

The number of seconds between each checkpoint

`dfs.namenode.checkpoint.txns`

`dfs.namenode.checkpoint.check.period`

Name node

2. Maintain a secondary name node

`dfs.namenode.checkpoint.txns`

The number of transactions (edits to the file system) before checkpointing

`dfs.namenode.checkpoint.check.period`

How often to query for the number of uncheckpointed transactions

`dfs.namenode.checkpoint.period`

Name node

2. Maintain a secondary name node

A checkpoint occurs

`dfs.namenode.checkpoint.txns`

Whenever the specified number of transactions is reached

`dfs.namenode.checkpoint.period`

Or the specified amount of time has passed