

# Storing Data with HDFS

---



**Janani Ravi**

CO-FOUNDER, LOONYCORN

[www.loyncorn.com](http://www.loyncorn.com)

# Overview

**Understanding the components of HDFS**

**Moving files in and out of HDFS**

**Managing replication strategies for data nodes**

**Understanding failure management strategies for the name node**

HDFS

# Hadoop Distributed File System



# HDFS



**Built on commodity hardware**

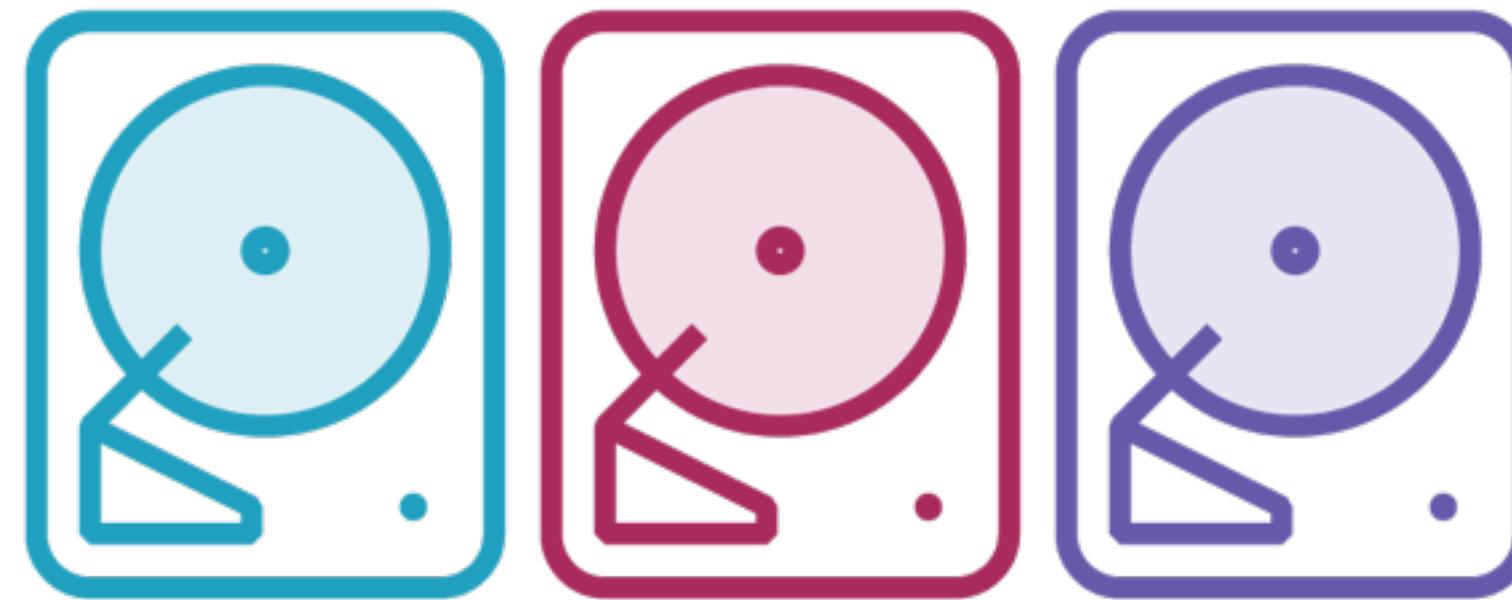
**Highly fault tolerant, hardware failure is the norm**

**Suited to batch processing - data access has high throughput rather than low latency**

**Supports very large data sets**

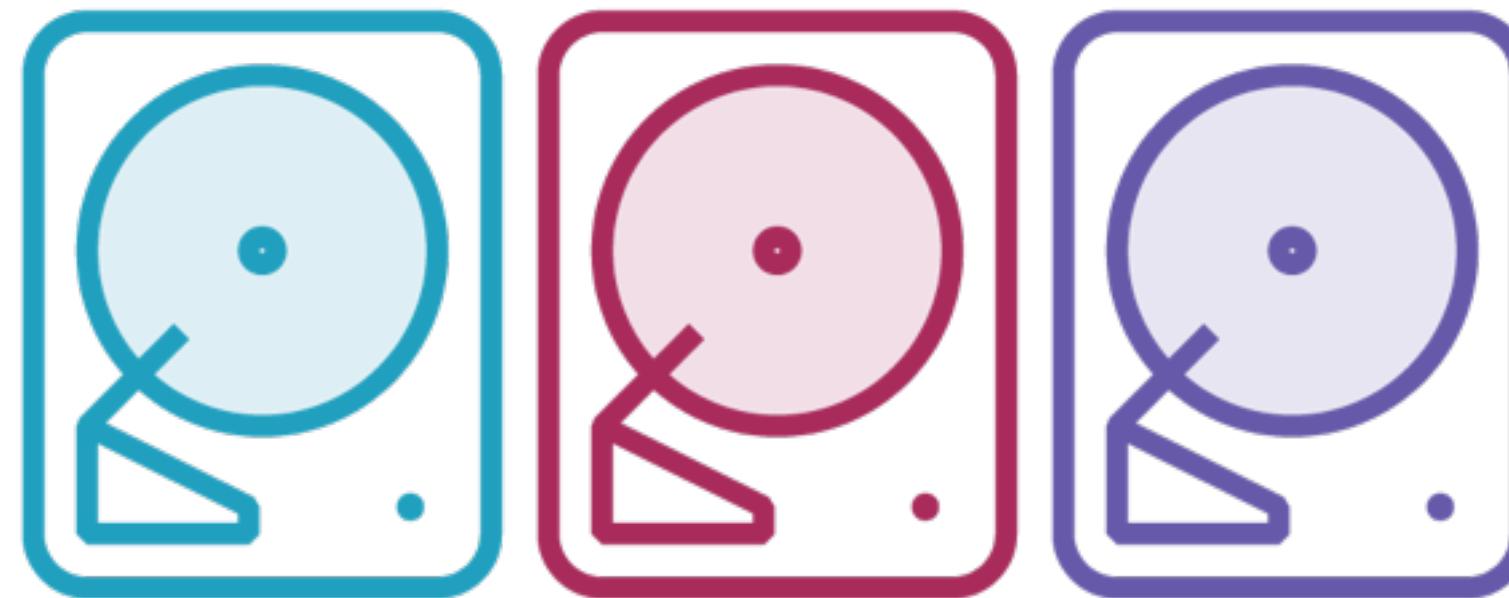
HDFS

Manage file storage across  
multiple disks



HDFS

**Each disk on a different machine  
in a cluster**



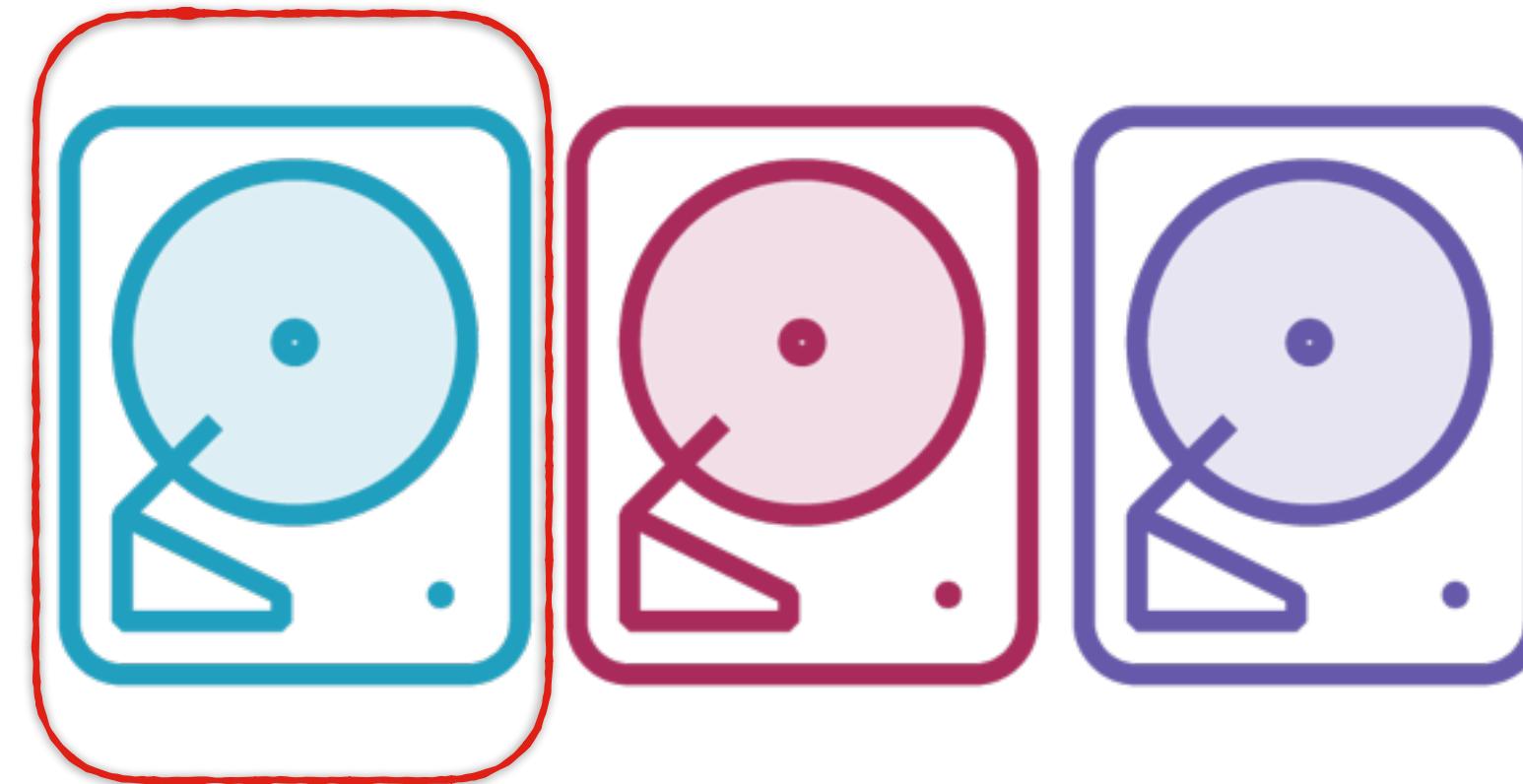
HDFS

# A cluster of machines



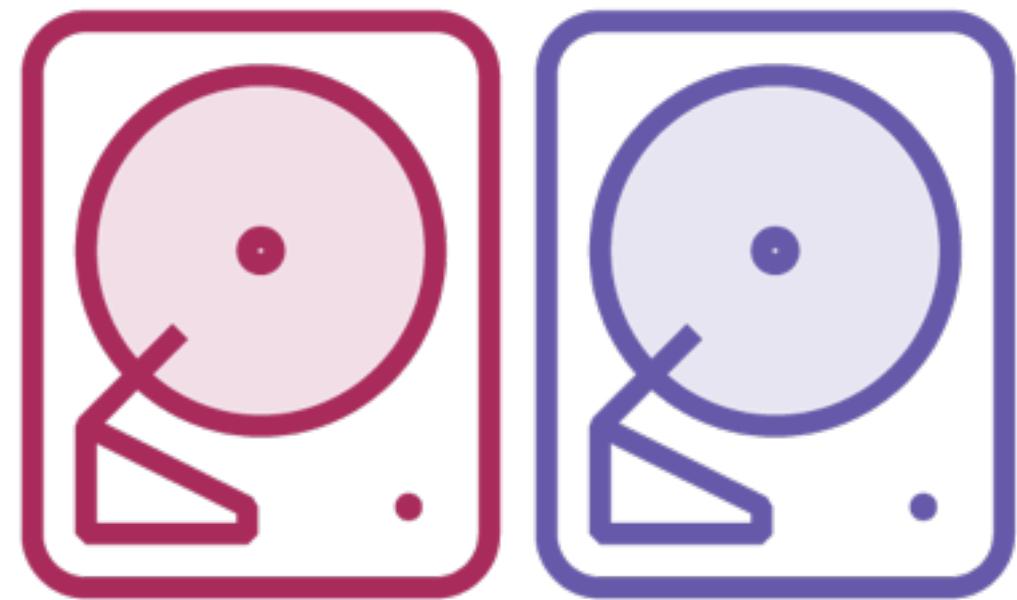
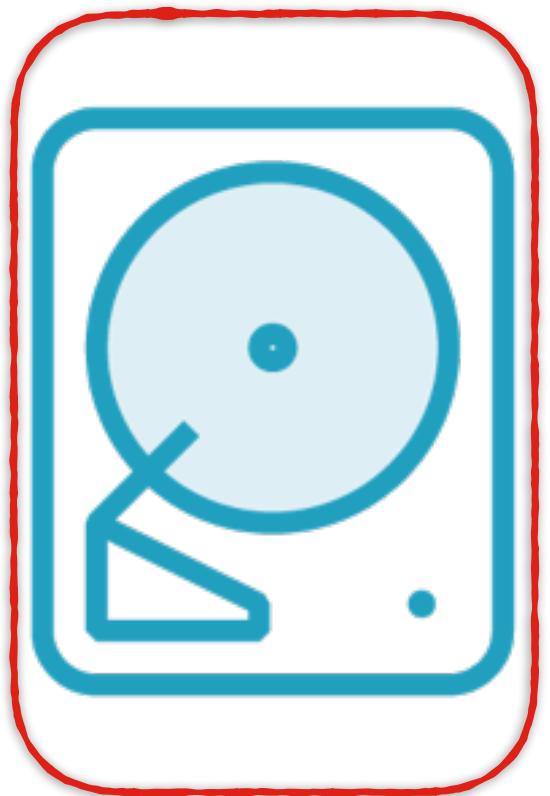
HDFS

# A node in the cluster



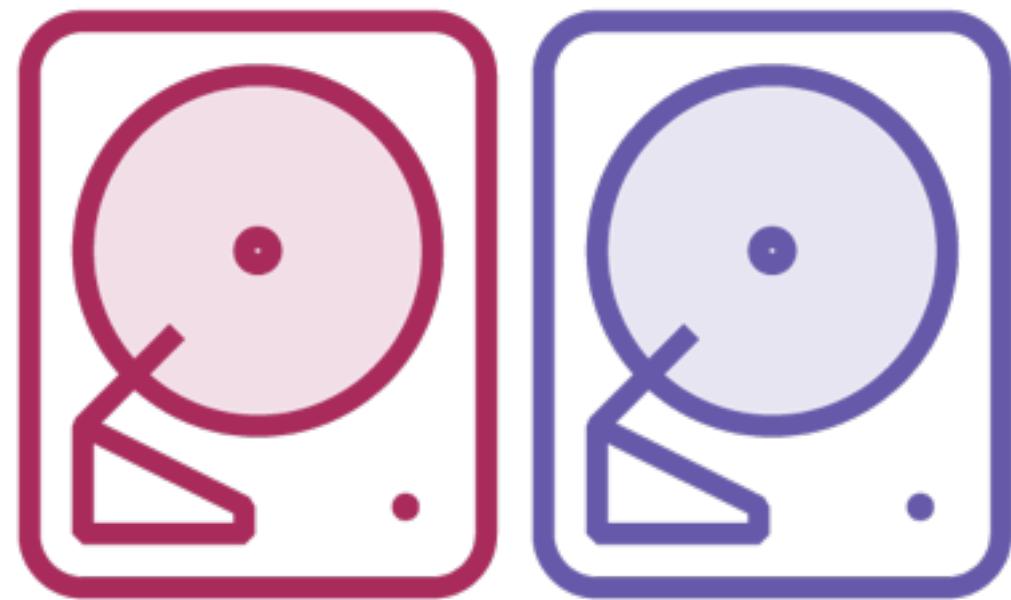
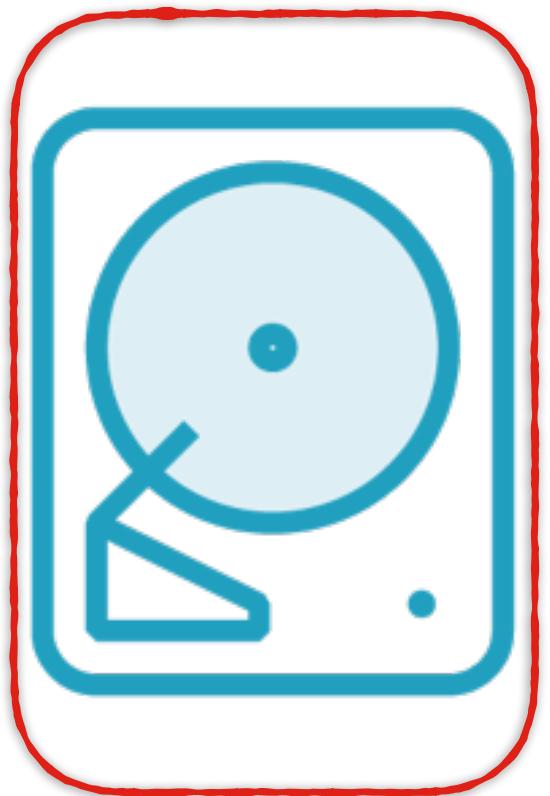
HDFS

1 node is the  
master node



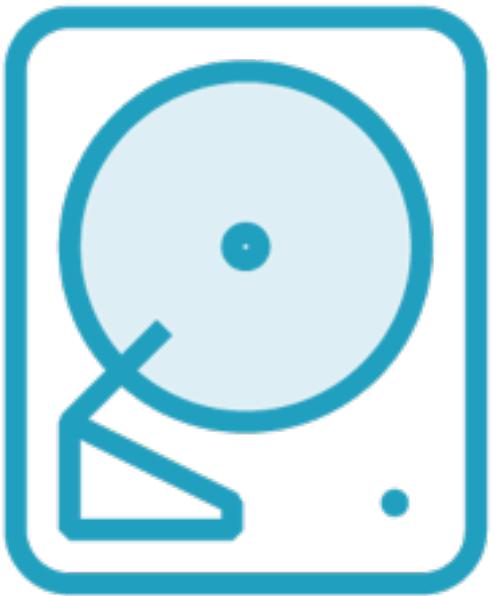
HDFS

# Name node

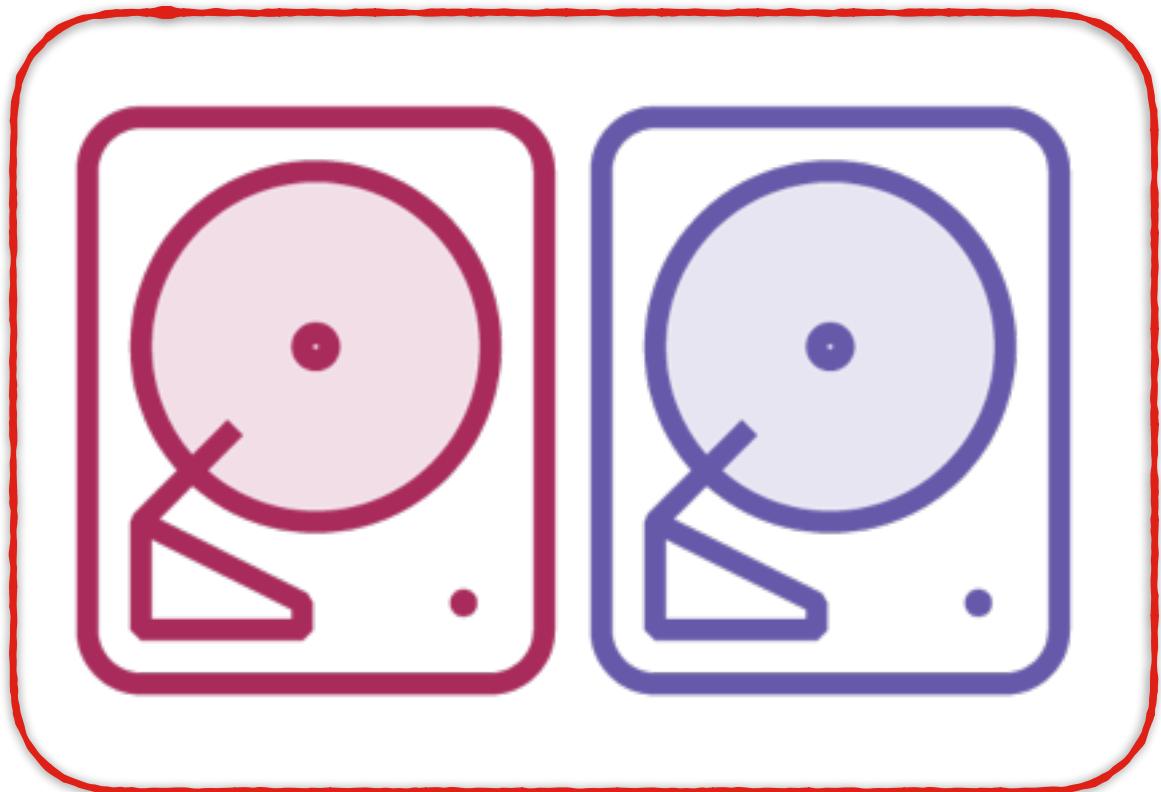


HDFS

Name node

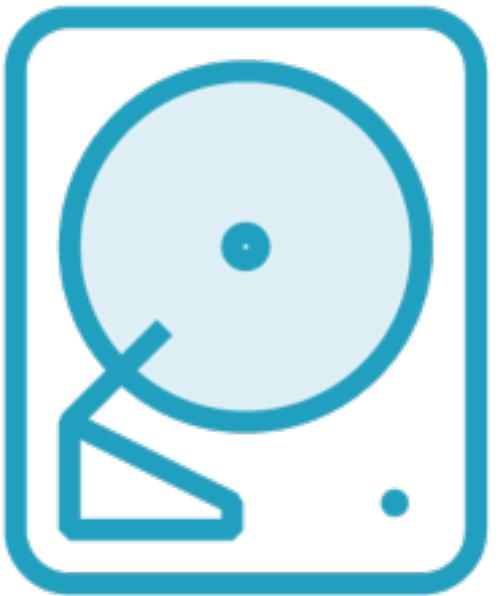


Data nodes

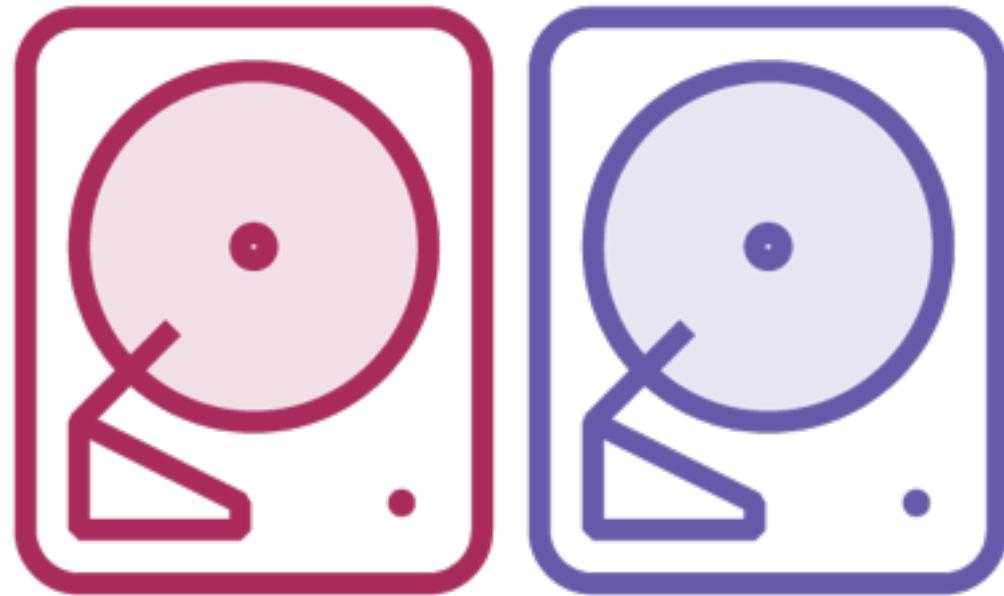


HDFS

Name node



Data nodes

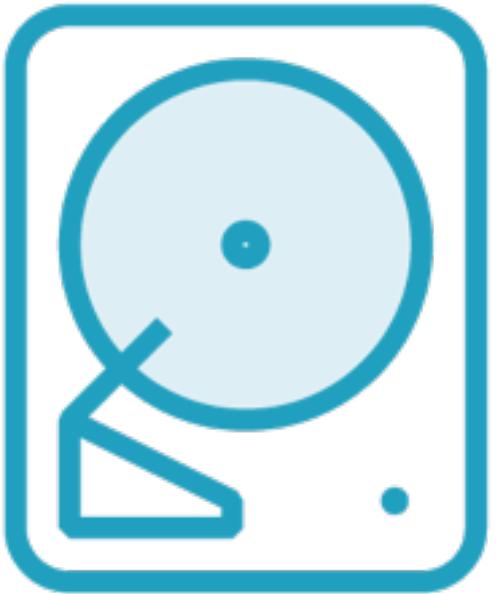


# HDFS



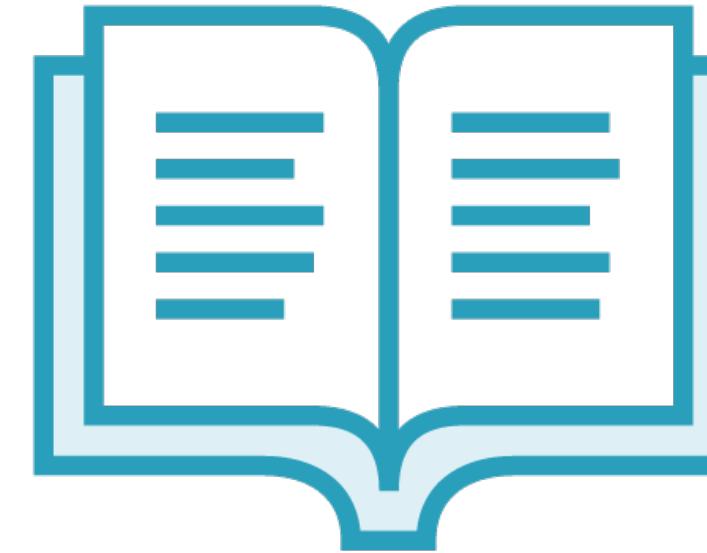
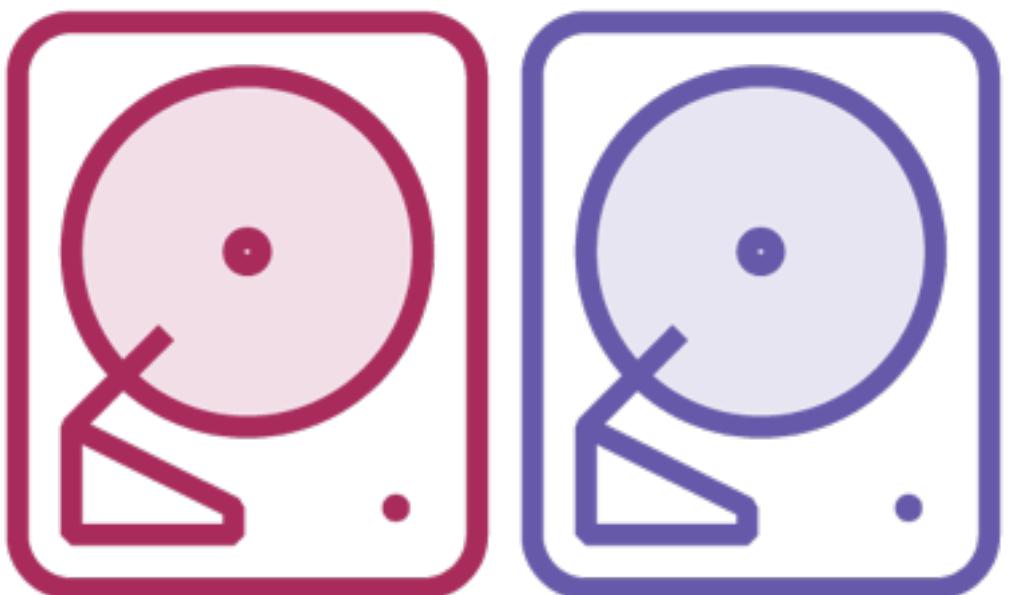
If the data in the distributed  
file system is a book

# Name node



The name node  
is the table of  
contents

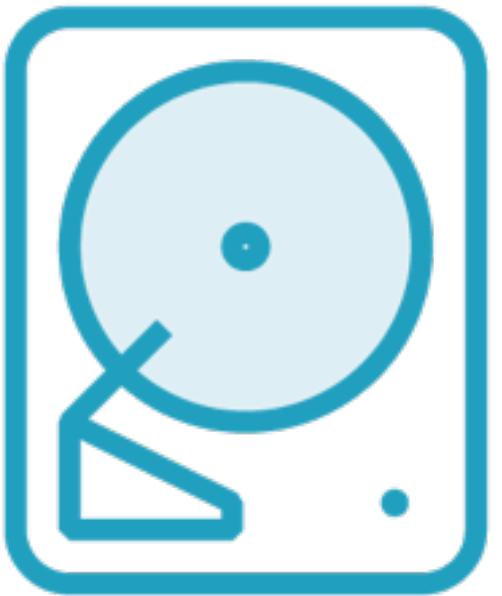
## Data nodes



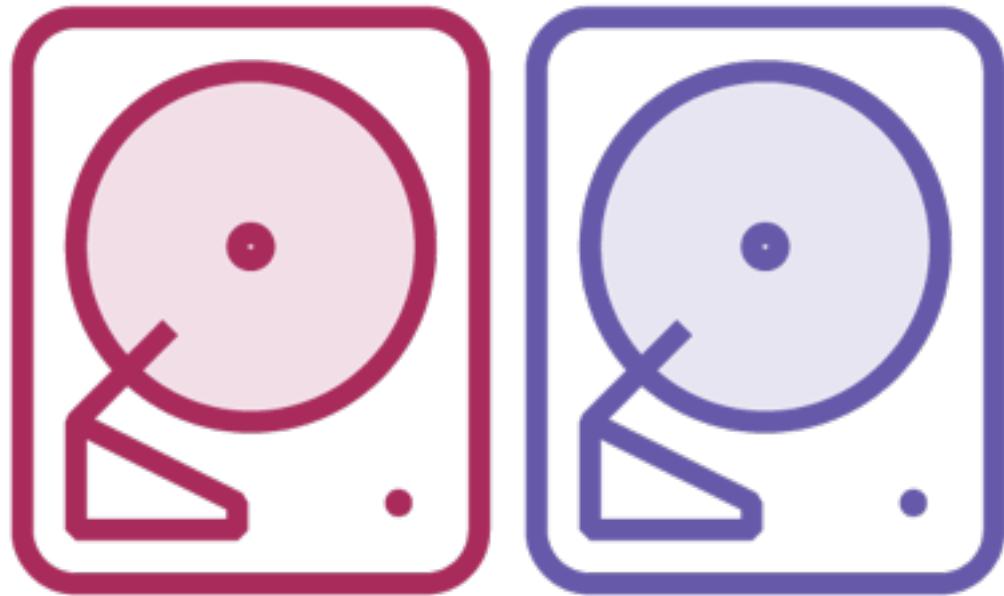
The data nodes hold the actual text in each page

HDFS

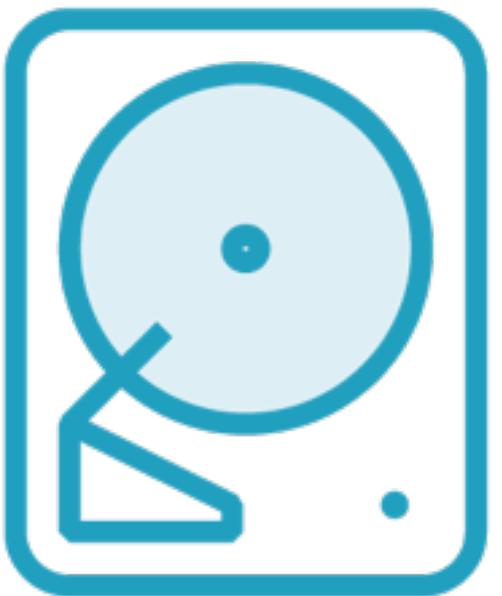
Name node



Data nodes



# Name node

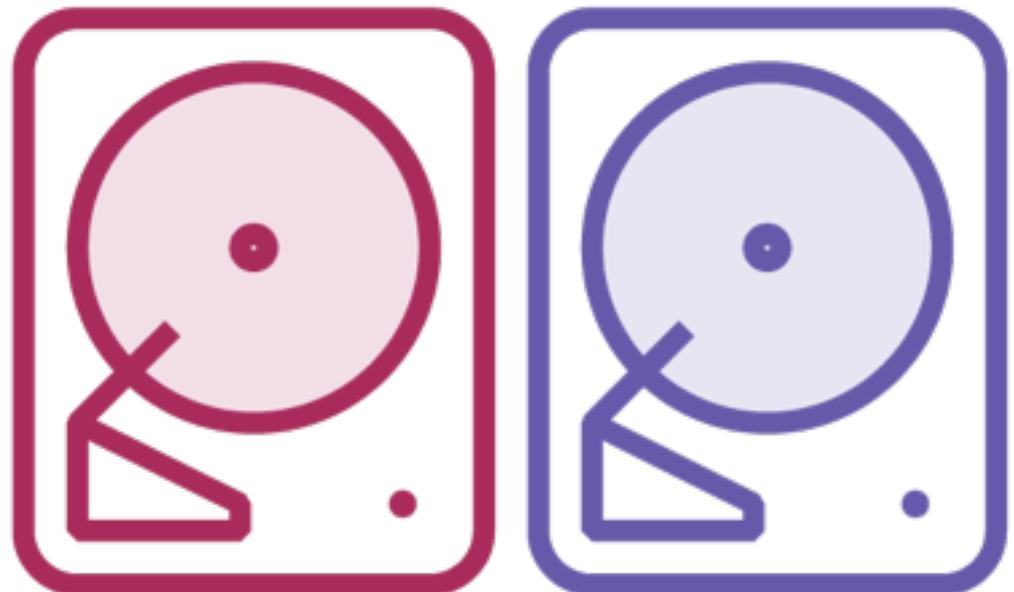


**Manages the overall file system**

## **Stores**

- The directory structure
- Metadata of the files

# Data nodes



**Physically stores the data  
in the files**

# Storing a File in HDFS

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

A large  
text file

# Storing a File in HDFS

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

## Block 1

Distributed indexing

## Block 2

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 document ID 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-

## Block 3

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 of the key features of MapReduce is that it is fault-tolerant, so that if a machine fails, its work can be reassigned to another machine.

## Block 4

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 of 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 \$ns 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 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.

## Block 5

Figure 4.5: An example of distributed indexing with MapReduce. [Adapted from Dean and Ghemawat (2004).]

## Block 6

In general, MapReduce breaks a large computing problem into smaller parts by recasting it in terms of manipulation of key-value pairs .

## Block 7

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 termID) for infrequent terms. We do not address this problem here and assume that all nodes share a consistent term \$\rightarrow termID mapping.

## Block 8

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 local intermediate files, the segment files (shown as \lbox{a-f}\medstrut} \lbox{g-p}\medstrut} \lbox{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 continuous terms or termIDs.) The term partitions are 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

# Break the data into blocks

# Storing a File in HDFS

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

## Block 1

Distributed indexing

## Block 2

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 document ID 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-

## Block 3

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 of the key features of MapReduce is that it is fault-tolerant, so that if a machine fails, the work can be reassigned to another machine.

## Block 4

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 of 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 \$ns 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

## Block 5

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).

## Block 6

\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 .

## Block 7

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 termID for infrequent terms. We do not address this problem here and assume that all nodes share a consistent term \$\rightarrow termID mapping.

## Block 8

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 perform in the inverted index construction phase.

local intermediate files, the segment files (shown as \lbox{a-f}\medstrut} \lbox{g-p}\medstrut} \lbox{q-z}\medstrut} in Figure 4.5 ).

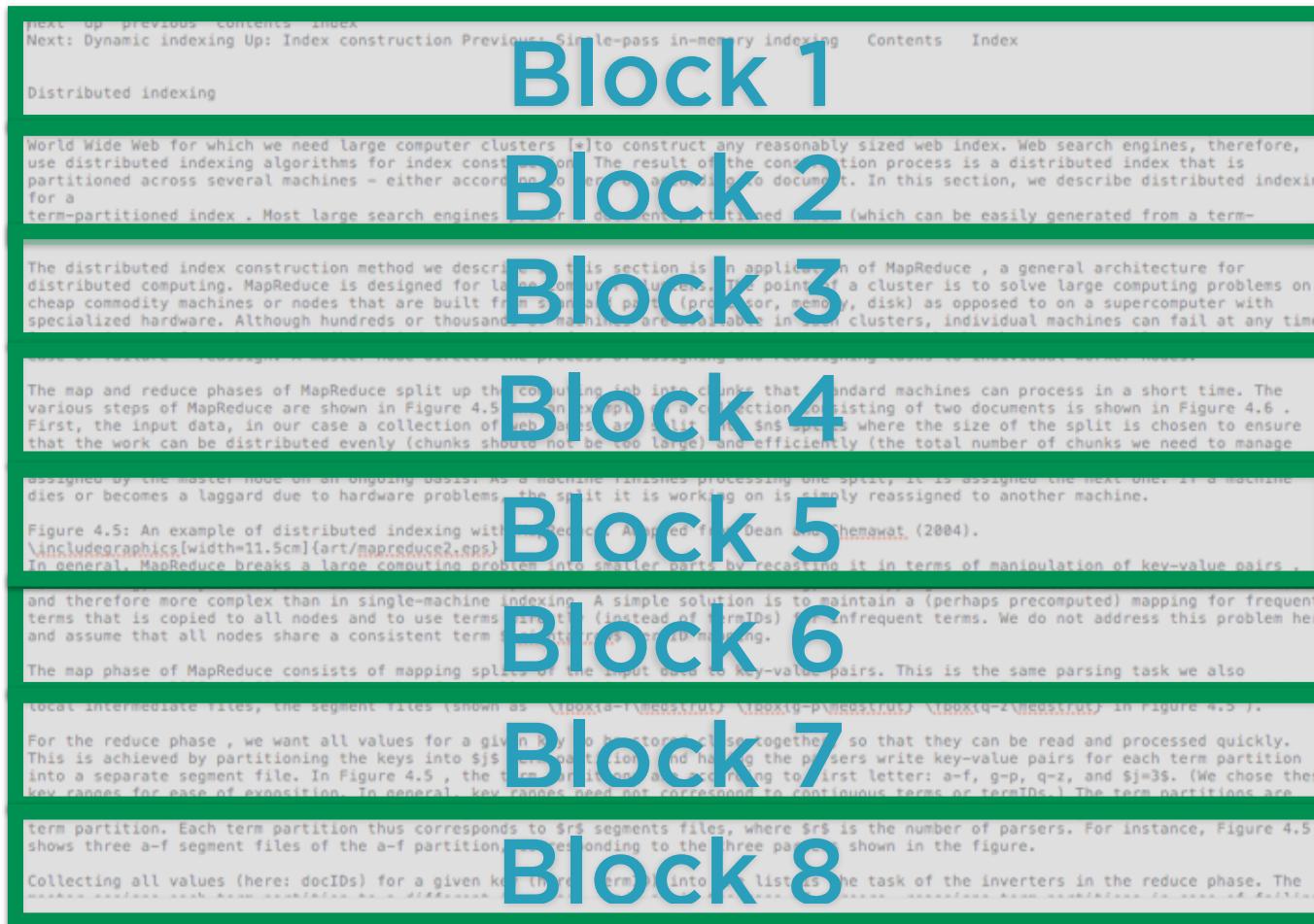
For the reduce phase , we want all values for a given key to be stored close together, so that they can be read and processed quickly. This is achieved by partitioning the keys into \$j\$ term partitions and having the parsers write key-value pairs for each term partition into a separate segment file. 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 continuous terms or termIDs.) The term partitions are

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

# Break the data into blocks

# Storing a File in HDFS



**Break the data  
into blocks**

**Different length files are  
treated the same way**

**Storage is simplified**

**Unit for replication and  
fault tolerance**

# Storing a File in HDFS

The screenshot shows a web page with a header containing links for 'Next', 'Up', 'previous', 'Contents', and 'Index'. Below this, there are links for 'Dynamic indexing', 'Index construction', 'Previous', 'Single-pass in-memory indexing', 'Contents', and 'Index'. The main content area is titled 'Distributed indexing' and contains the following text:

**Block 1**  
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 terms or according to document. In this section, we describe distributed indexing for a term-partitioned index. Most large search engines [+] use different partitioning schemes (which can be easily generated from a term-

**Block 2**  
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 commodity 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.

**Block 3**  
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. An example of 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 \$s\$ 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 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.

**Block 4**  
Figure 4.5: An example of distributed indexing with MapReduce. Adapted from Dean and Ghemawat (2004).  
In general, MapReduce breaks a large computing problem into smaller parts by recasting it in terms of manipulation of key-value pairs.

**Block 5**  
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 term IDs (instead of termIDs) for infrequent terms. We do not address this problem here and assume that all nodes share a consistent term to termID mapping.

**Block 6**  
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

**Block 7**  
local intermediate files, the segment files (shown as `TERMID=1:DOC1,DOC2,DOC3`, `TERMID=2:DOC1,DOC2,DOC3`, `TERMID=4:DOC1,DOC2,DOC3` in Figure 4.5).

**Block 8**  
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 \$s\$ 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 continuous terms or termIDs.) The term partitions are

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 partitions shown in the figure.

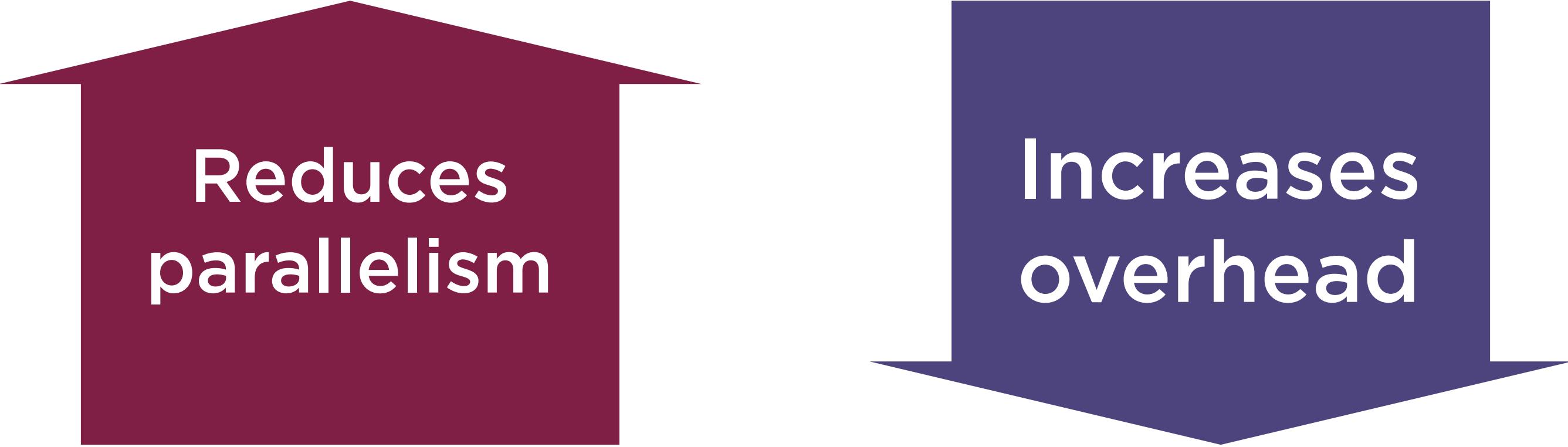
Collecting all values (here: docIDs) for a given key (here termID) into a list is the task of the inverters in the reduce phase. The

The blocks are  
of size 128 MB

# Storing a File in HDFS

**size 128 MB**

**Block size is a trade off**



Reduces parallelism

Increases overhead

# Storing a File in HDFS

**size 128 MB**

This size helps minimize  
the time taken to seek  
to the block on the disk

# Storing a File in HDFS

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

Distributed indexing

## Block 1

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 terms or according to document. In this section, we describe distributed indexing for a term-partitioned index. Most large search engines store their documents partitioned by term (which can be easily generated from a term-

## Block 2

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 commodity 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.

## Block 3

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. An example of 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 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

## Block 4

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.

## Block 5

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, 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 term IDs (instead of termIDs) for infrequent terms. We do not address this problem here and assume that all nodes share a consistent term frequency in term mapping.

## Block 6

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

## Block 7

local intermediate files, the segment files (shown as `TERMID=1@DOC1.DAT`, `TERMID=2@DOC1.DAT`, `TERMID=4@DOC1.DAT` 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\$ 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 continuous terms or termIDs.) The term partitions are

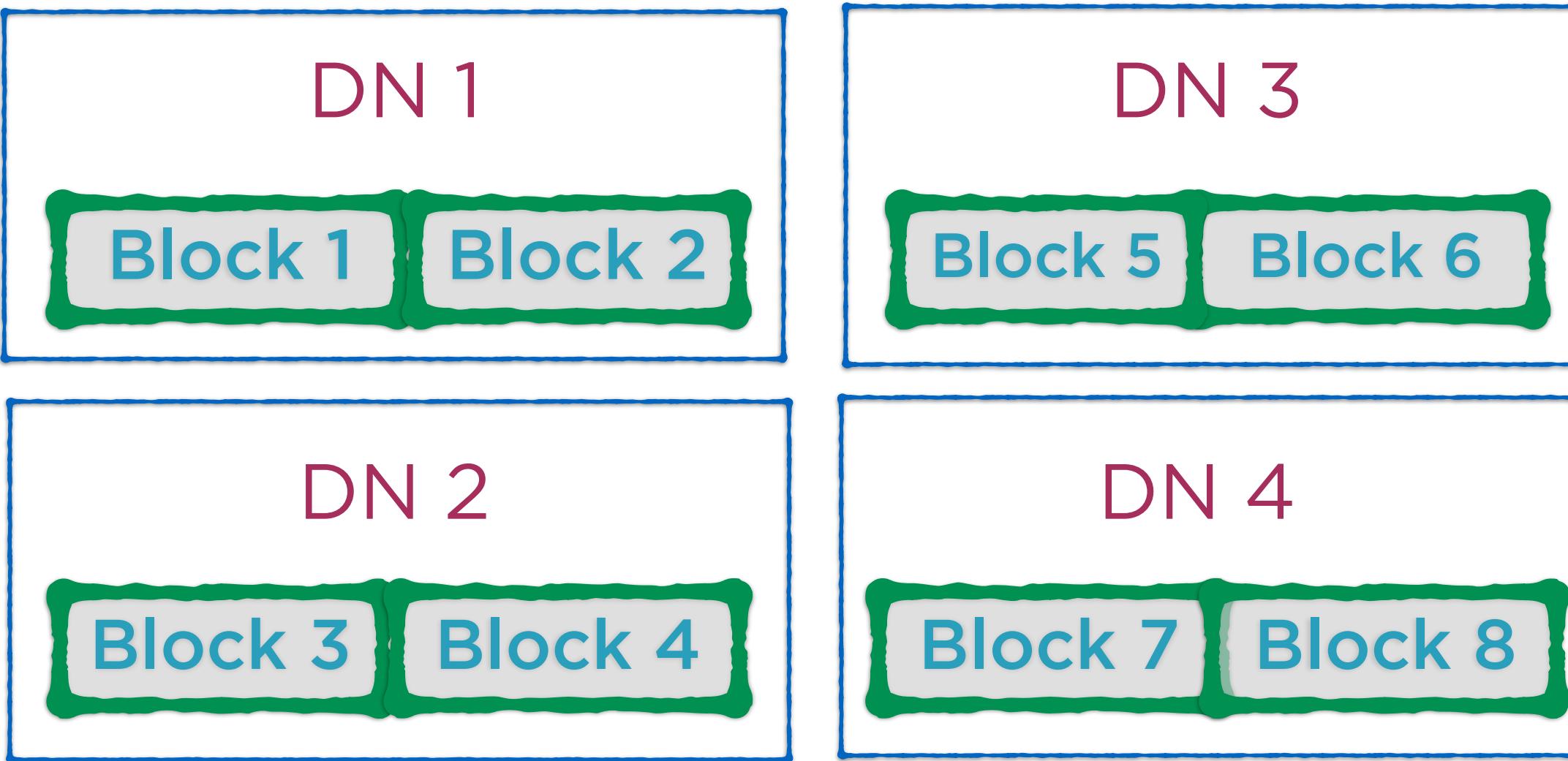
## Block 8

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 partitions shown in the figure.

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

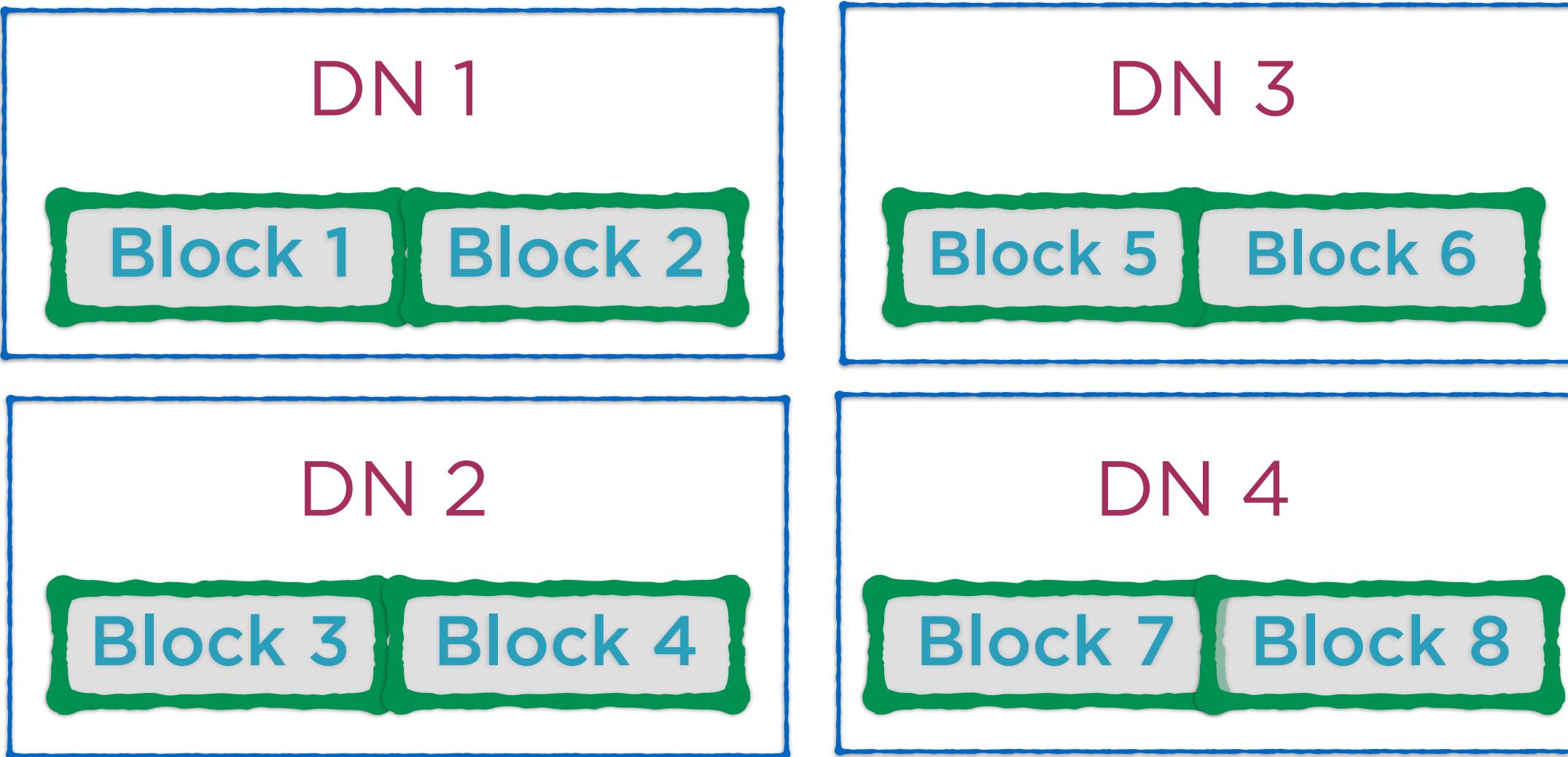
Store the  
blocks across  
the data nodes

# Storing a File in HDFS



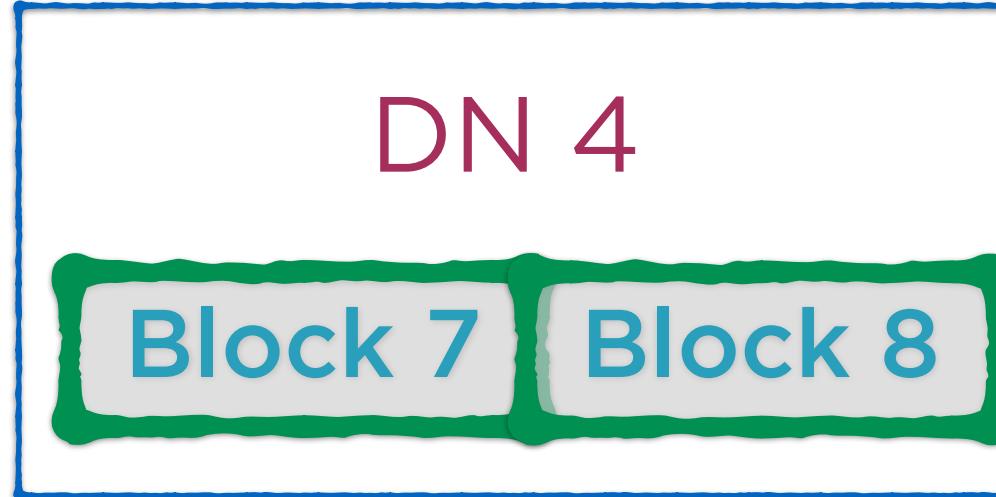
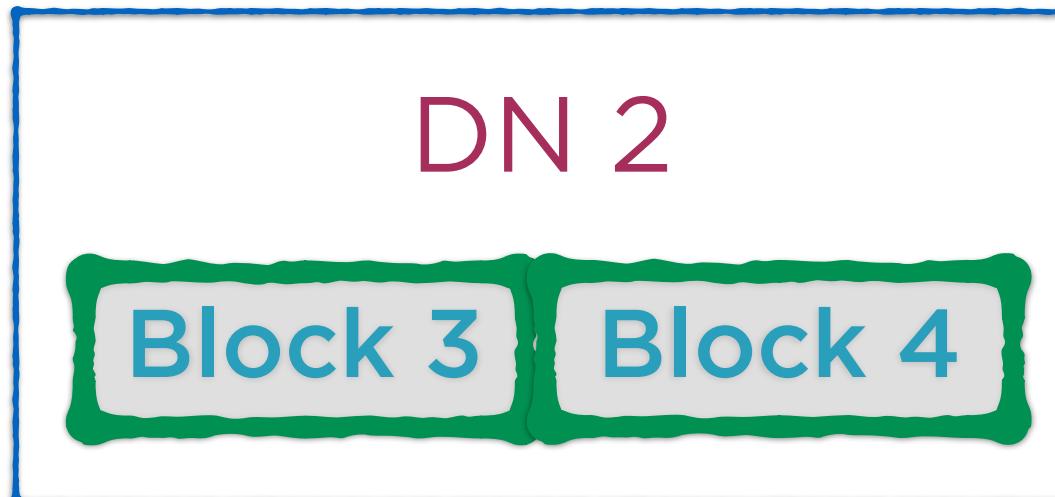
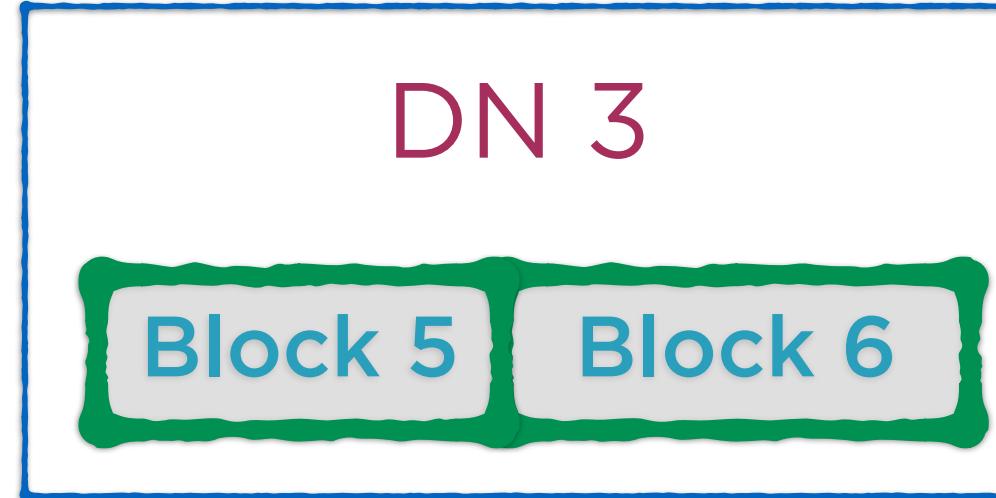
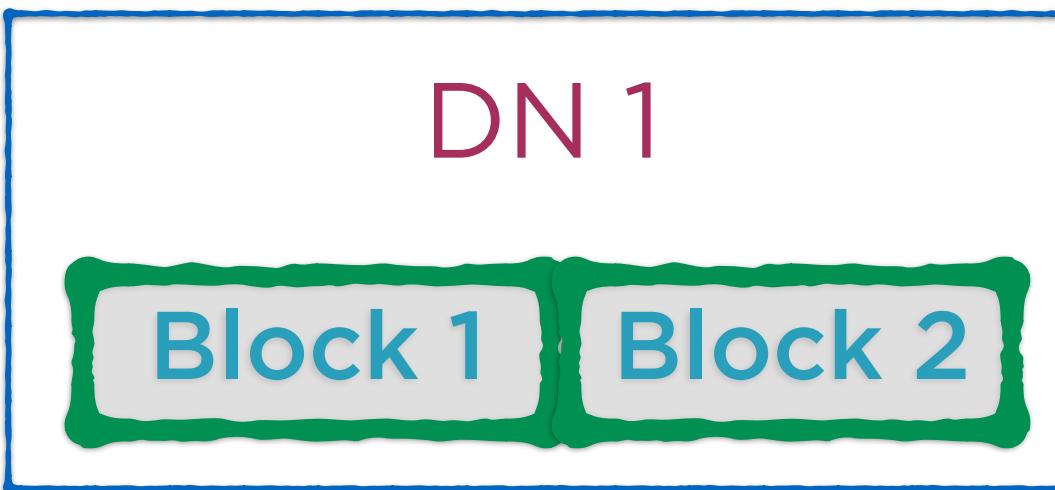
**Each node contains a partition or a split of data**

# Storing a File in HDFS



**How do we know where the splits of a particular file are?**

# Storing a File in HDFS



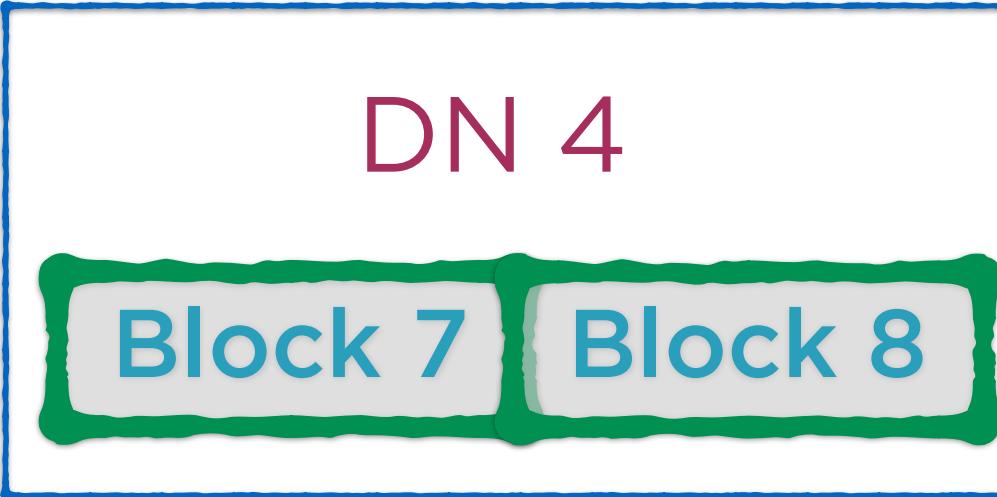
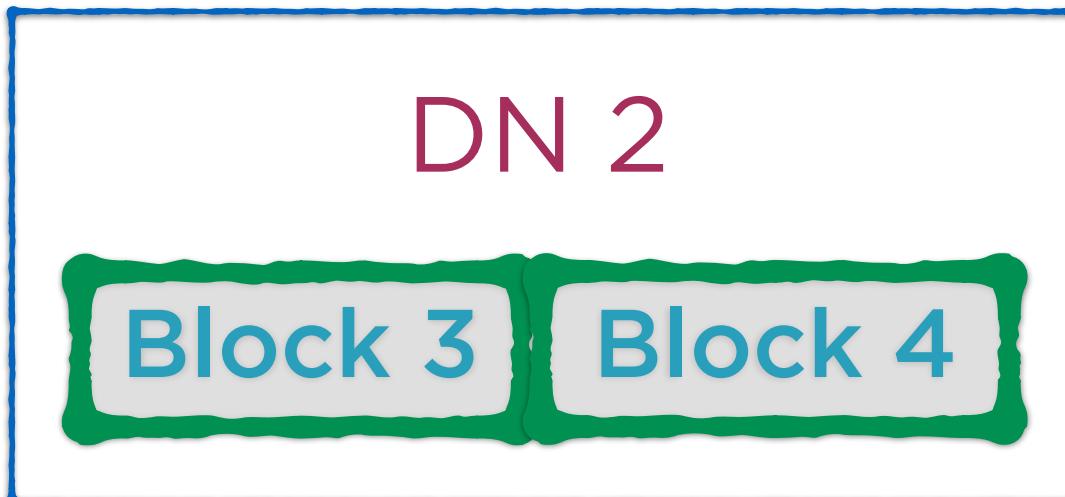
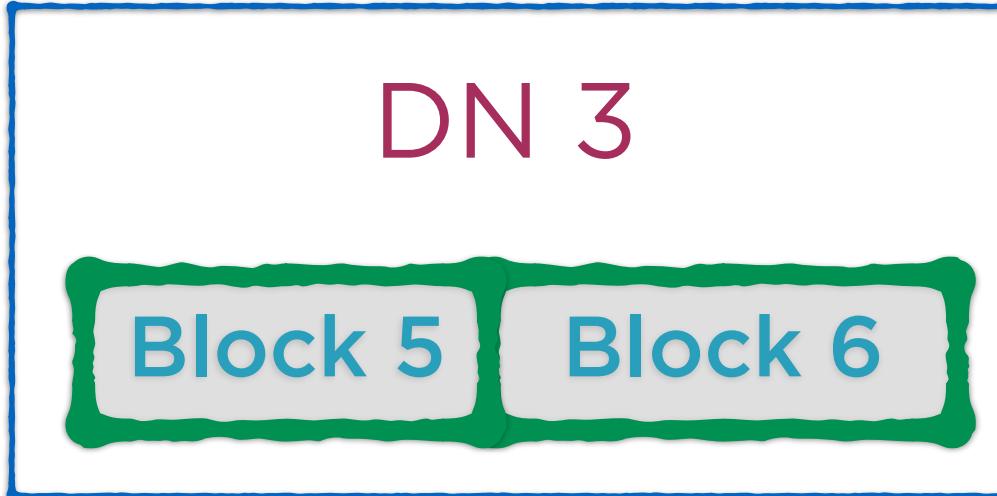
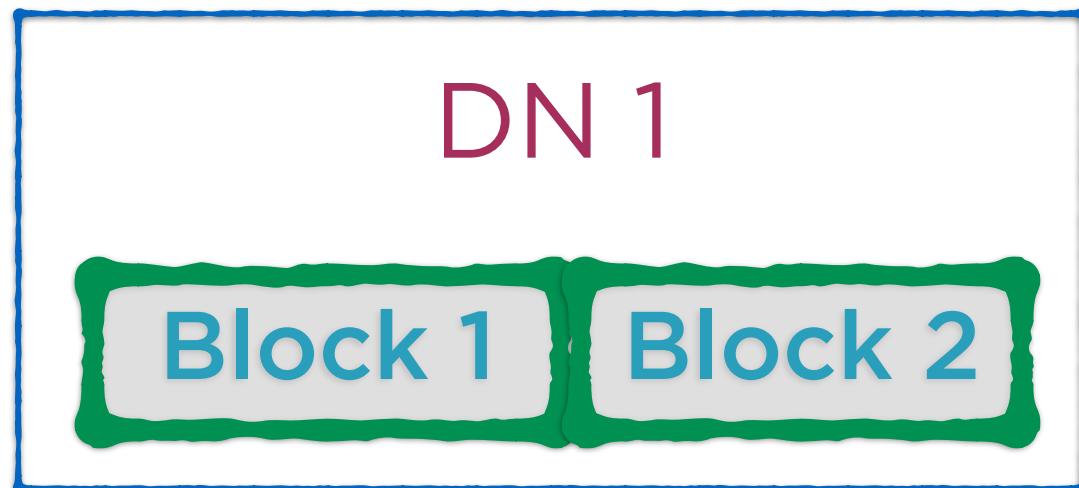
Name node

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

# Reading a File in HDFS

1. Use metadata in the name node to look up block locations
2. Read the blocks from respective locations

# Reading a File in HDFS

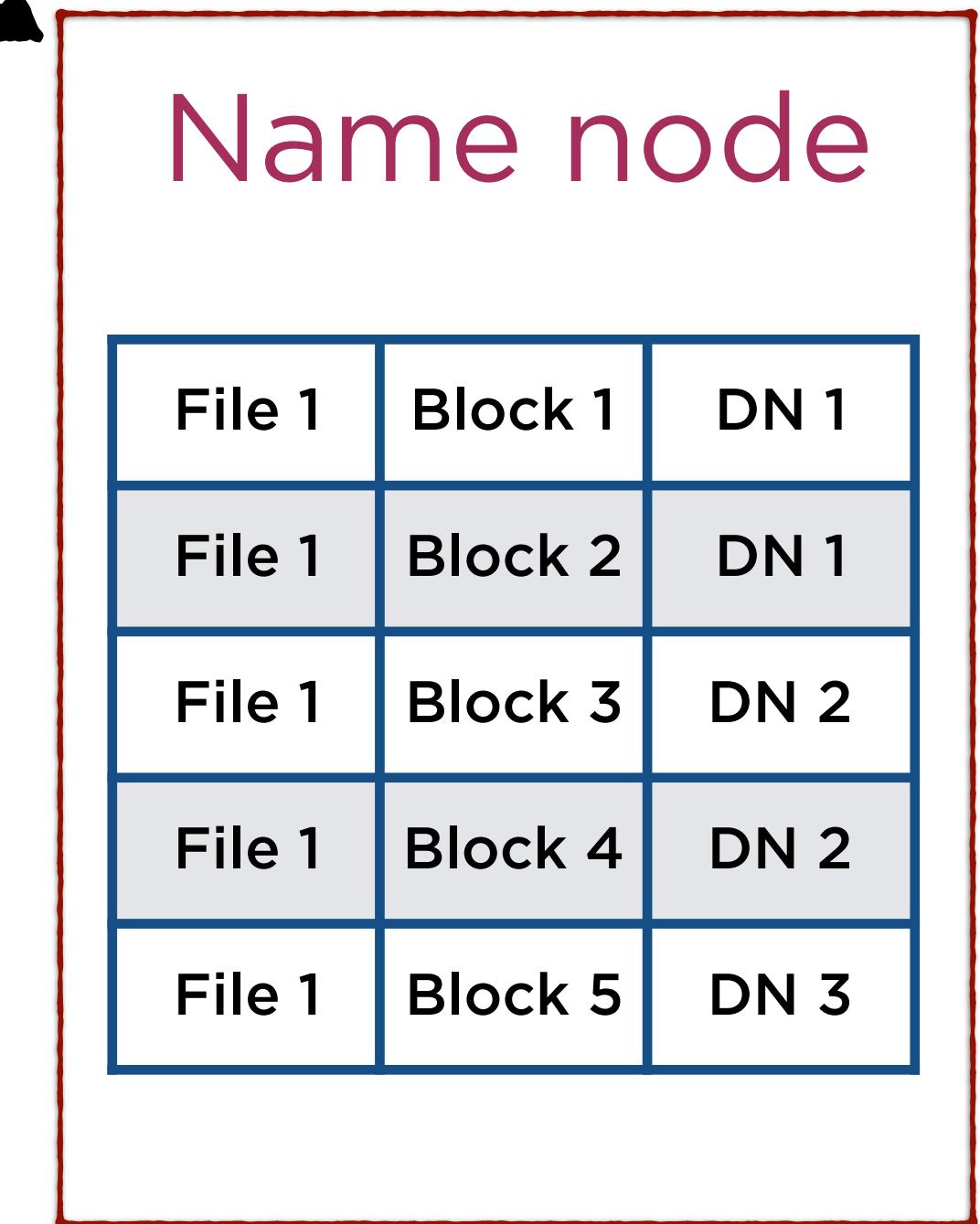
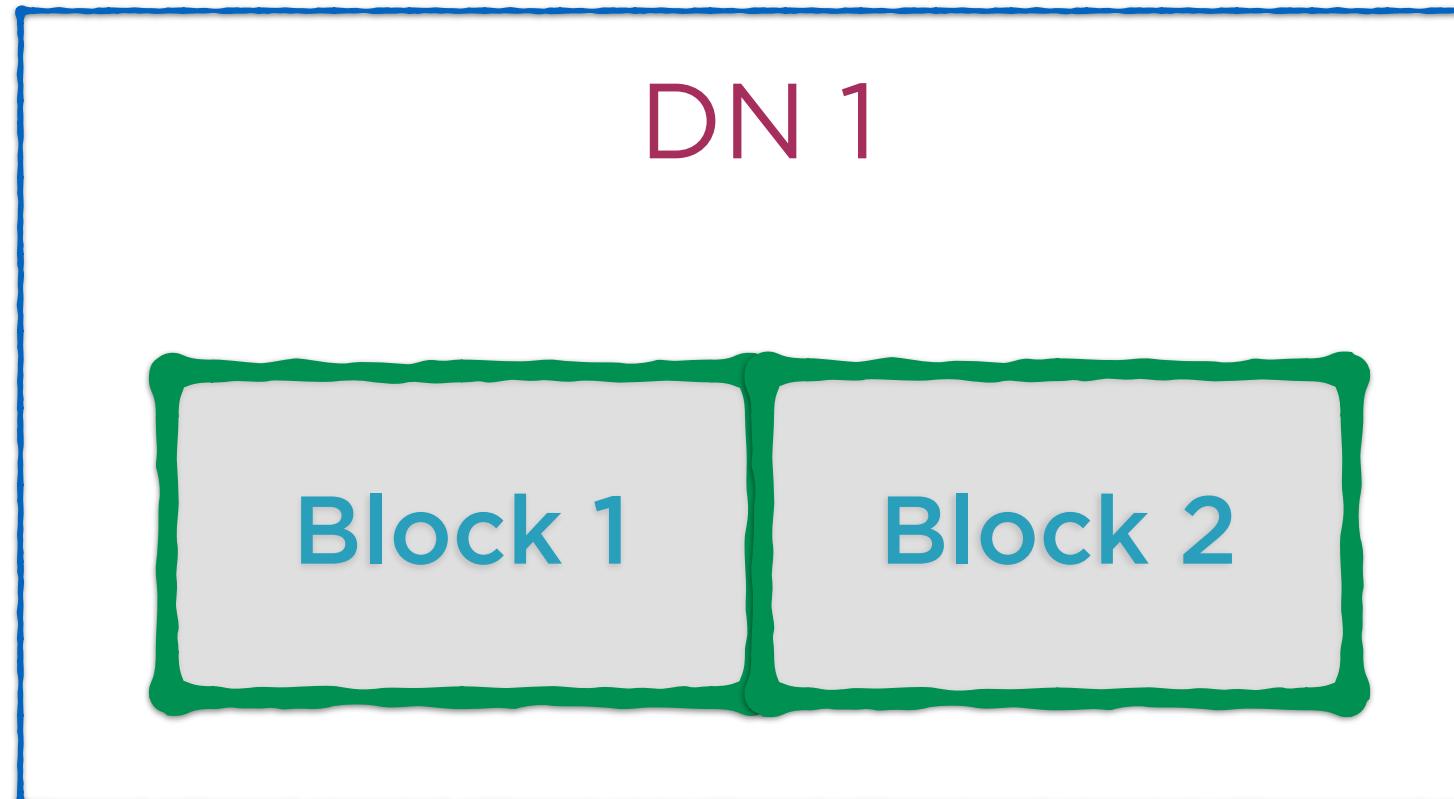


Name node

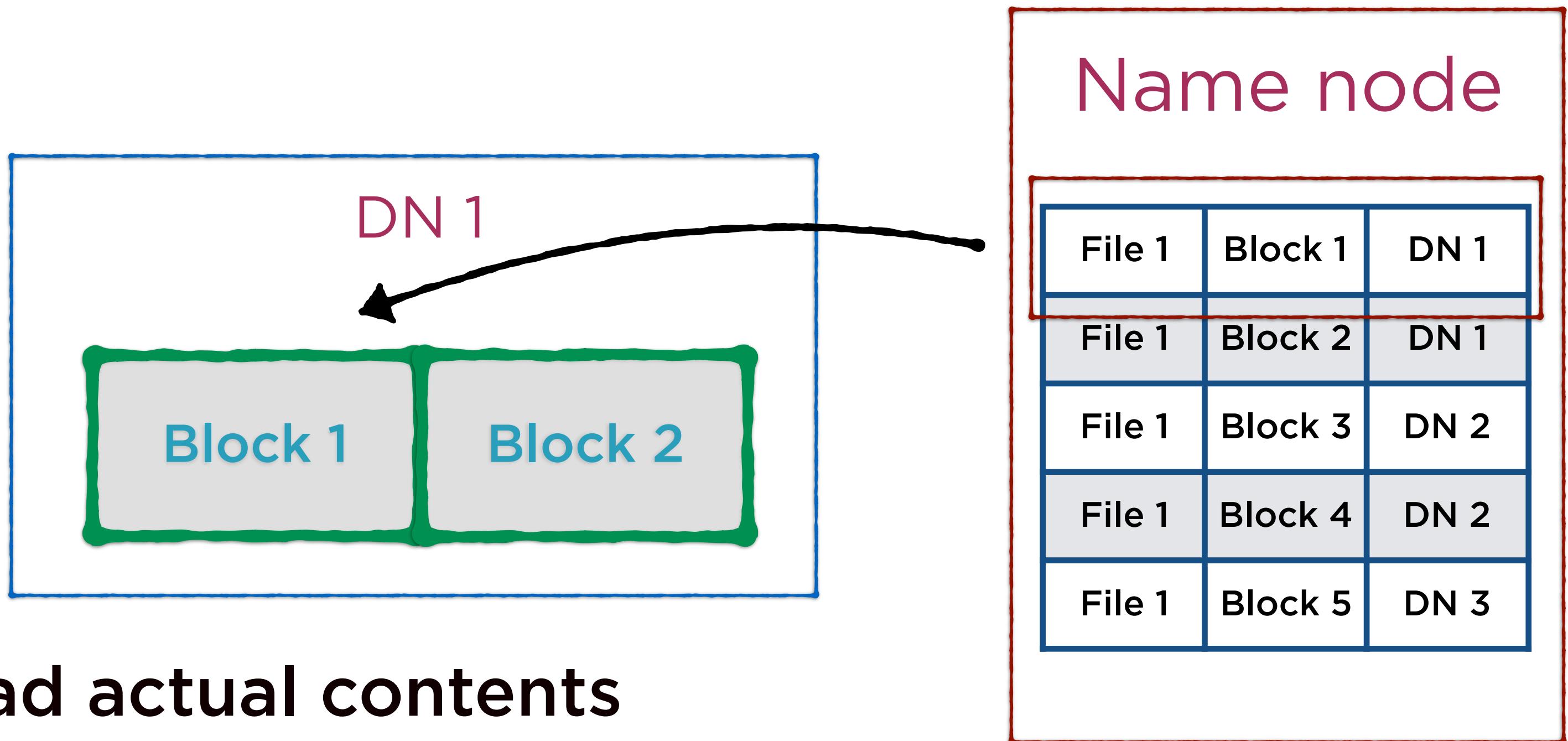
File 1	Block 1	DN 1
File 1	Block 2	DN 1
File 1	Block 3	DN 2
File 1	Block 4	DN 2
File 1	Block 5	DN 3

# Reading a File in HDFS

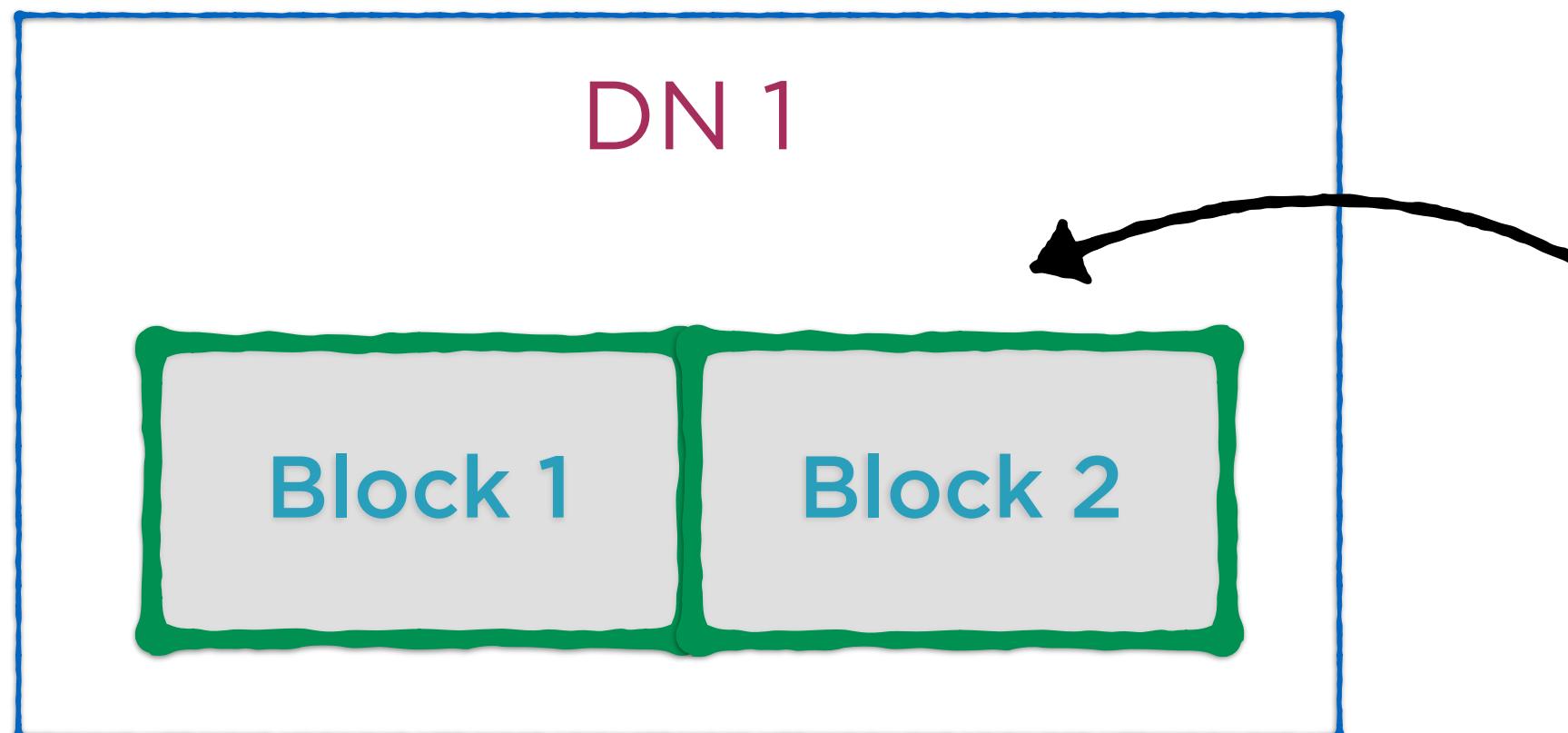
Request to the name node



# Reading a File in HDFS



# Reading a File in HDFS



Name node

Read actual contents  
from the block

# Demo

## **Interacting with the HDFS command line interface**

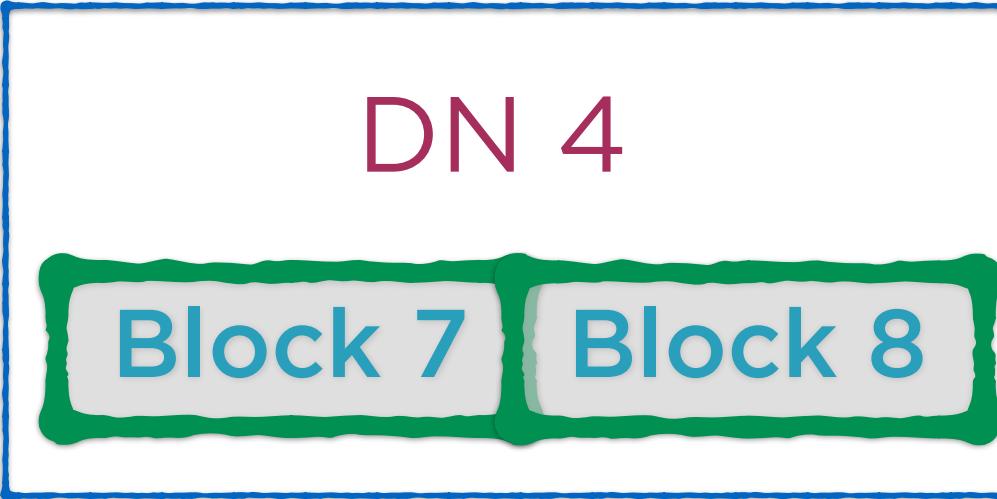
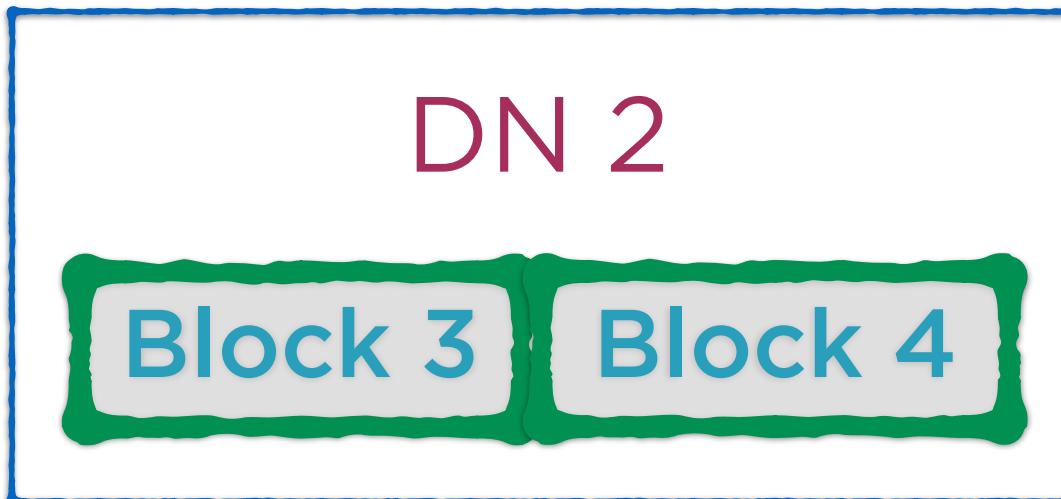
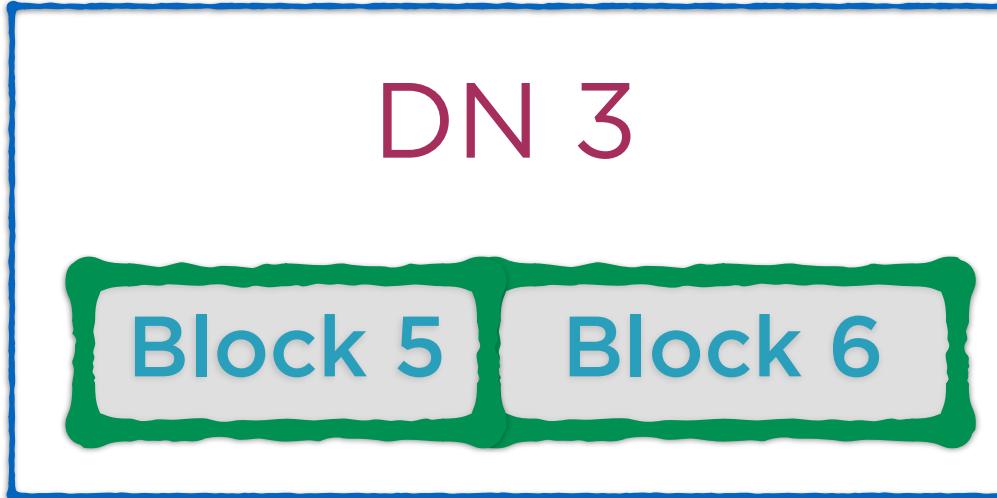
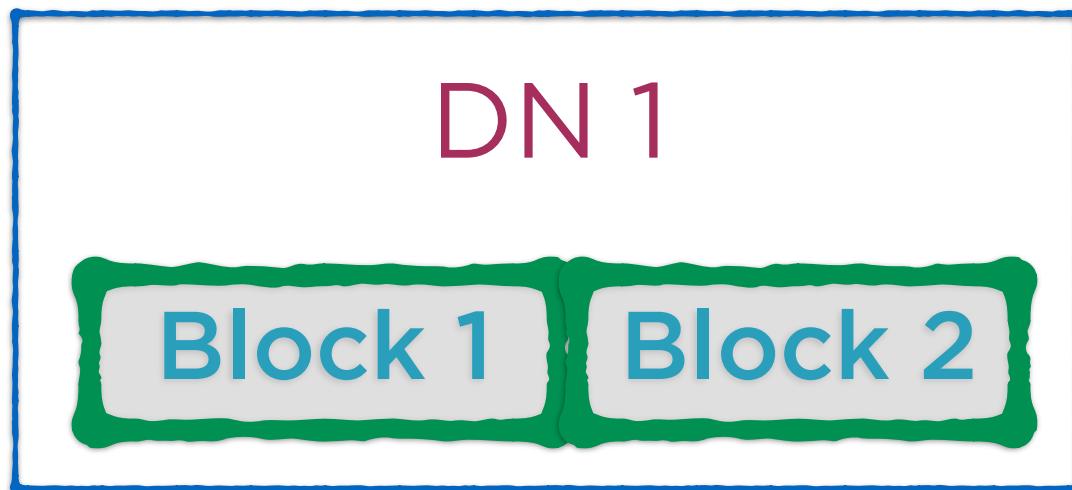
- Creating new directories
- Listing files and directories

# Demo

**Moving files from the local file system  
to HDFS and vice versa**

- copyFromLocal and copyToLocal
- put and get
- Viewing the contents of a file

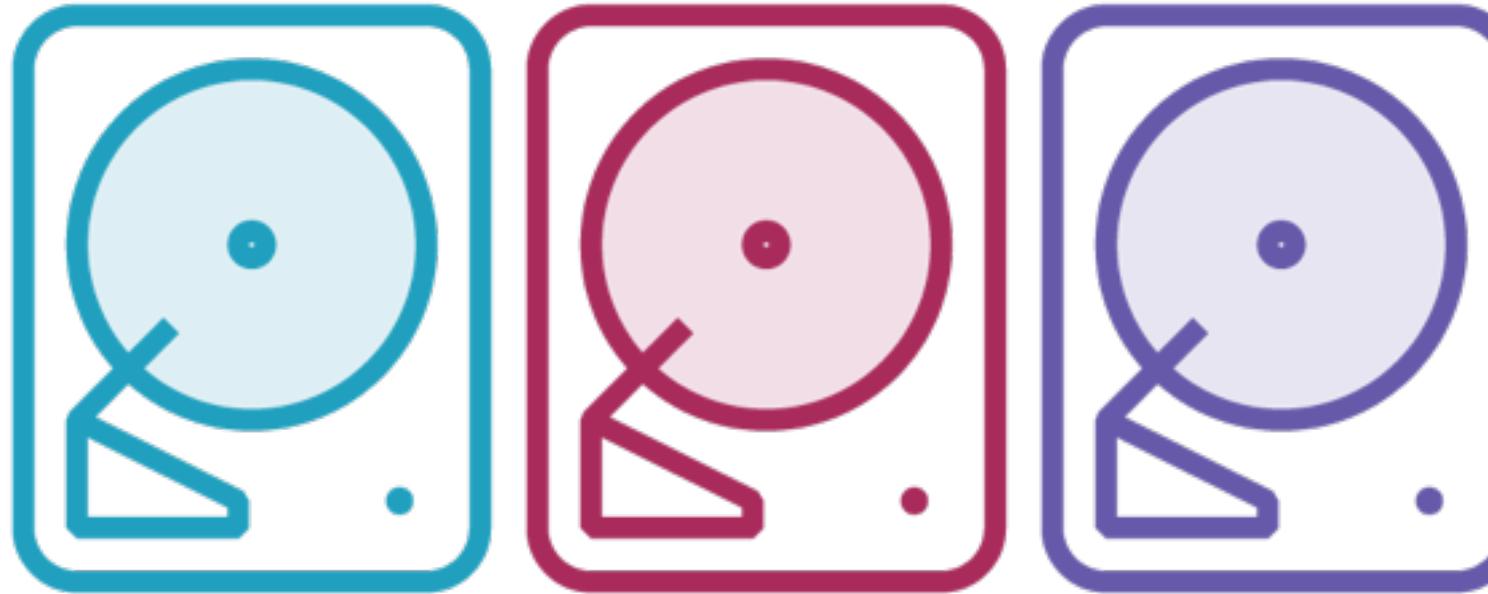
# A File Stored in HDFS



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

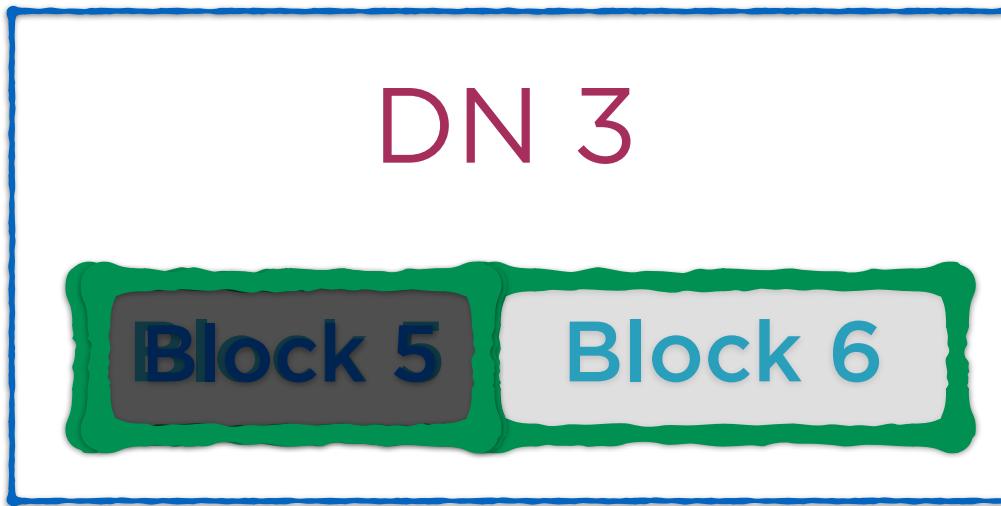
# Challenges of Distributed Storage



**Failure management  
in the data nodes**

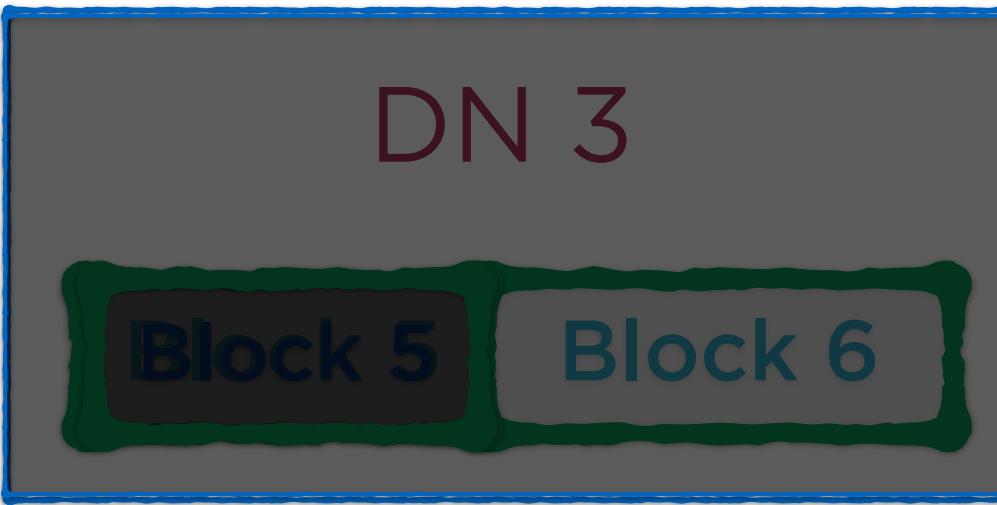
**Failure management  
for the name node**

# A File Stored in HDFS



**What if one of the  
blocks gets corrupted?**

# A File Stored in HDFS



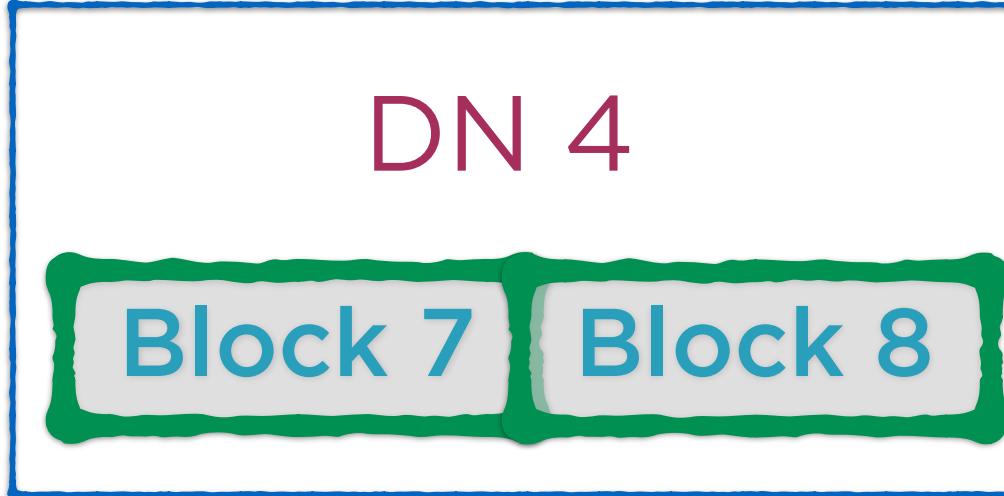
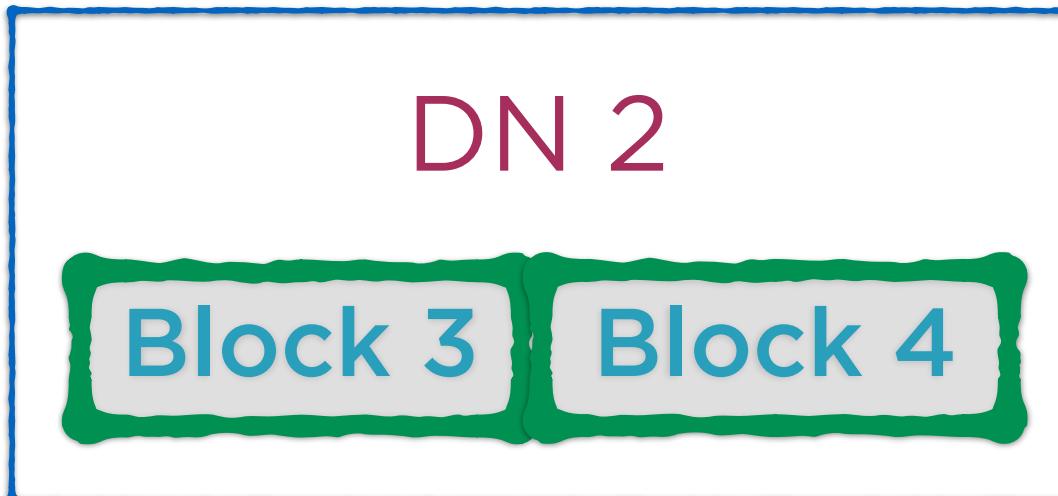
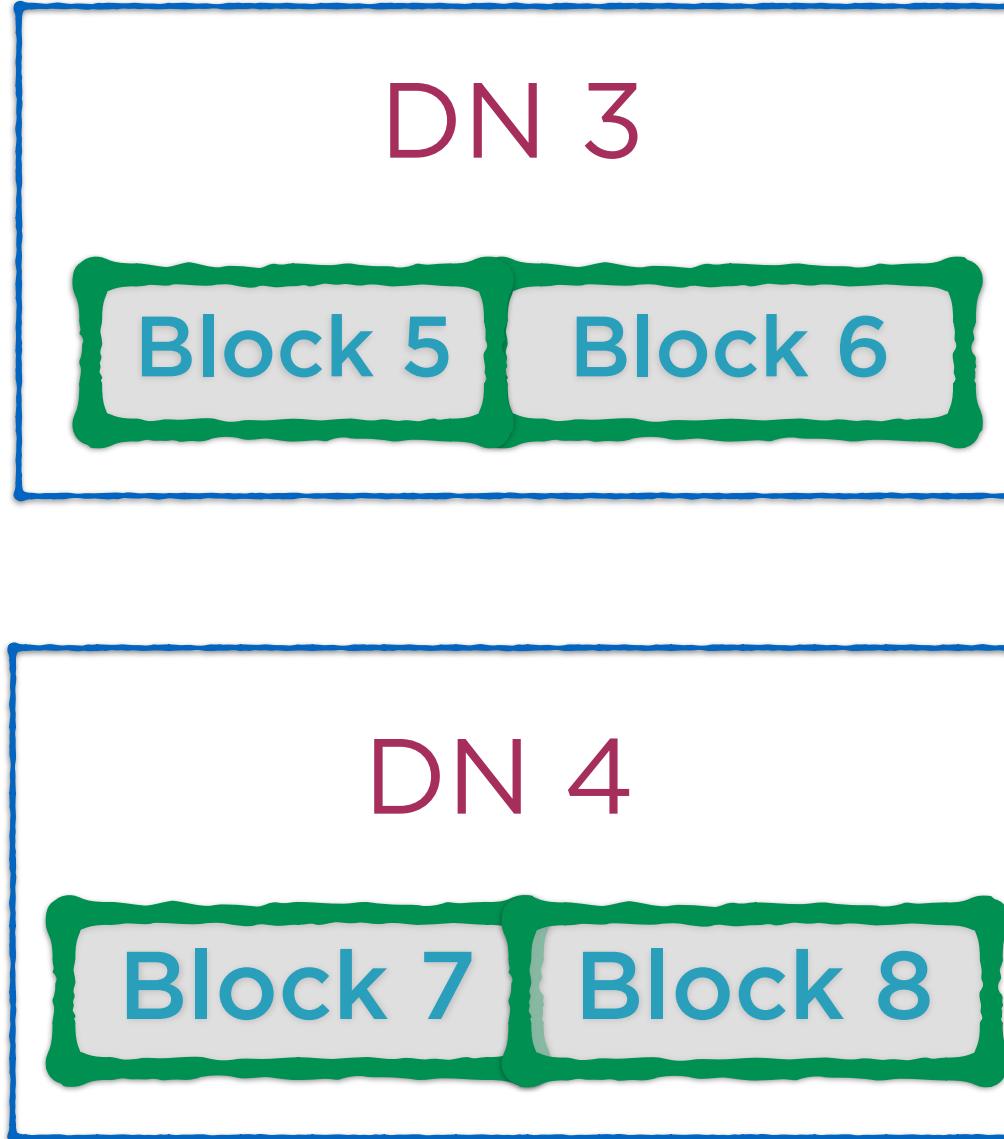
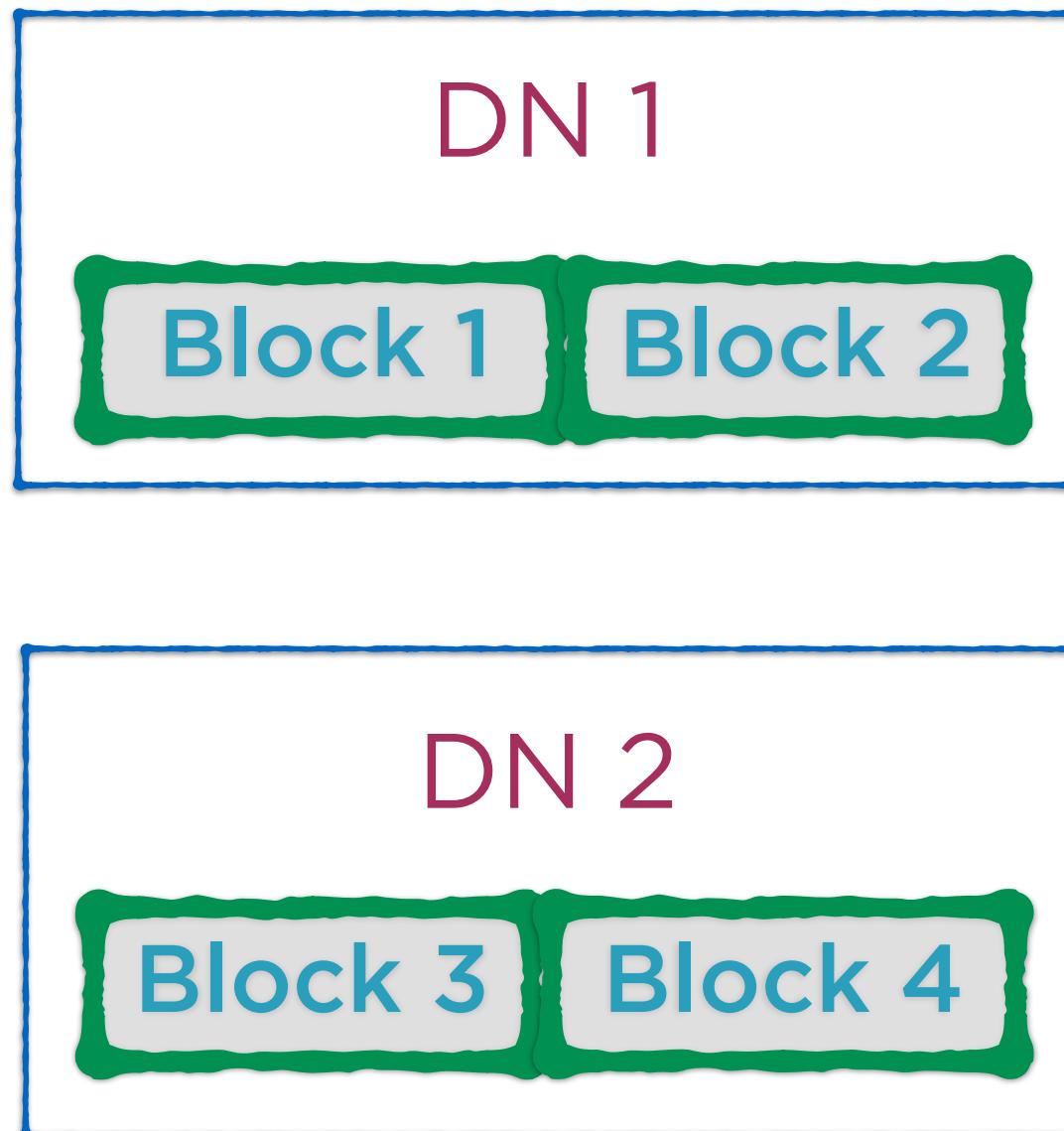
**What if a data node  
containing some blocks  
crashes?**

# Managing Failures in Data Nodes



**Define a  
replication factor**

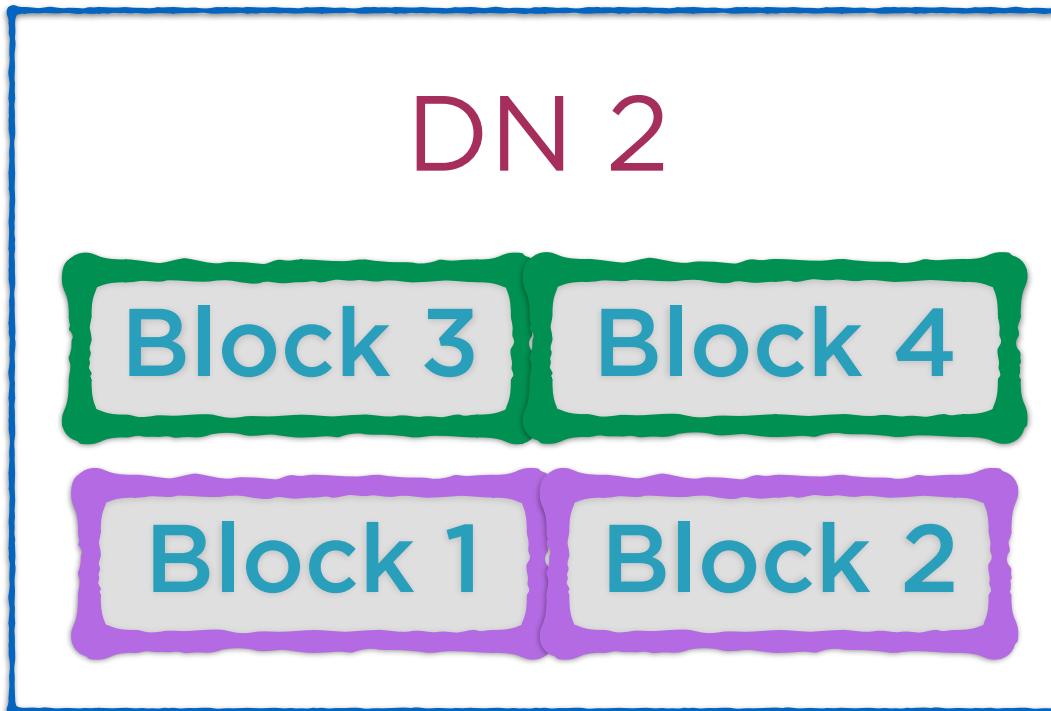
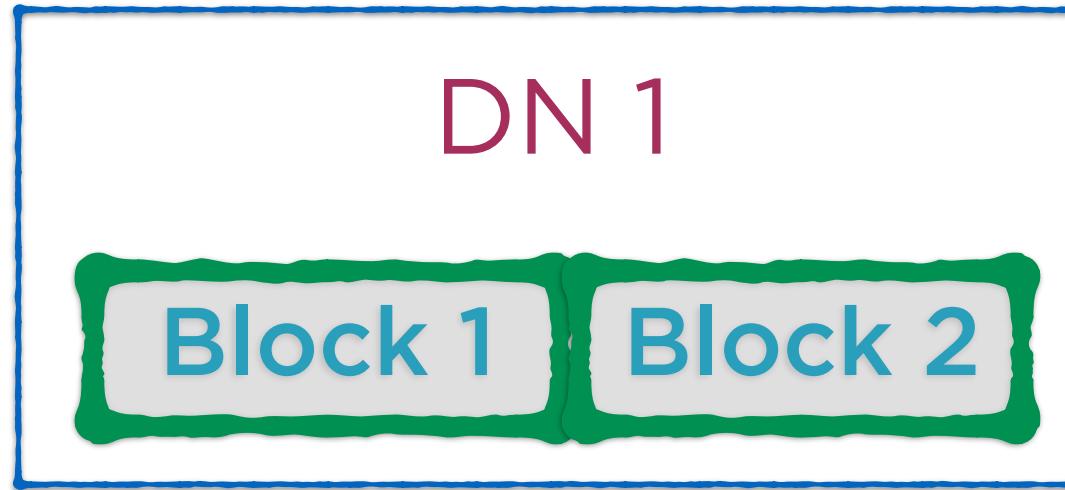
# A File Stored in HDFS



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

# Replication



1. Replicate blocks based on the replication factor
2. Store replicas in different locations

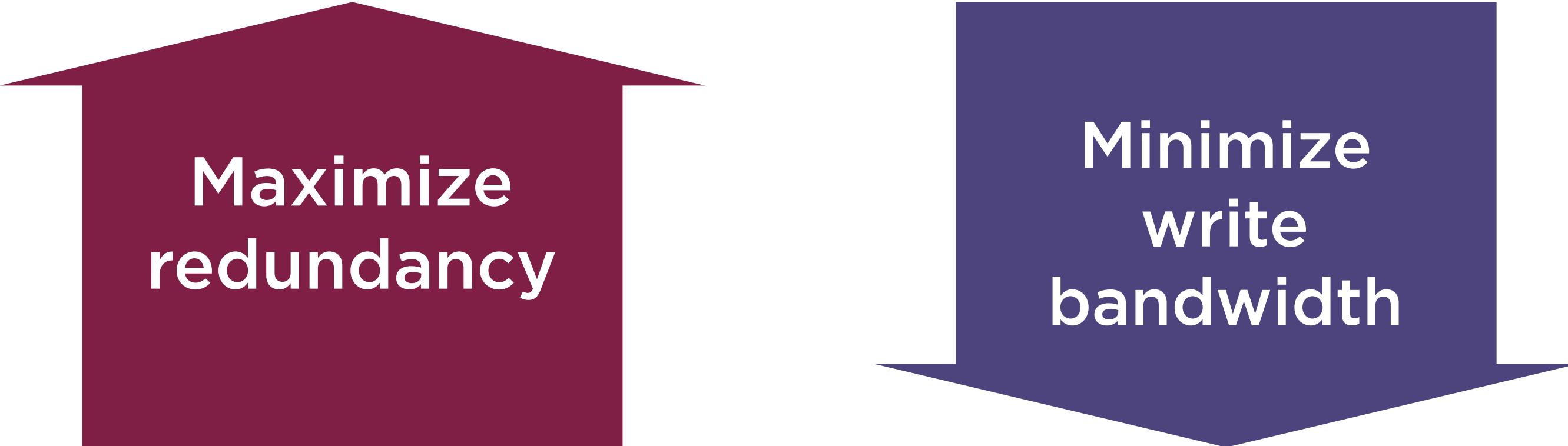
# Replication

The replica locations are also 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
File 1	Block 1	DN 2
File 1	Block 2	DN 2

# Choosing Replica Locations

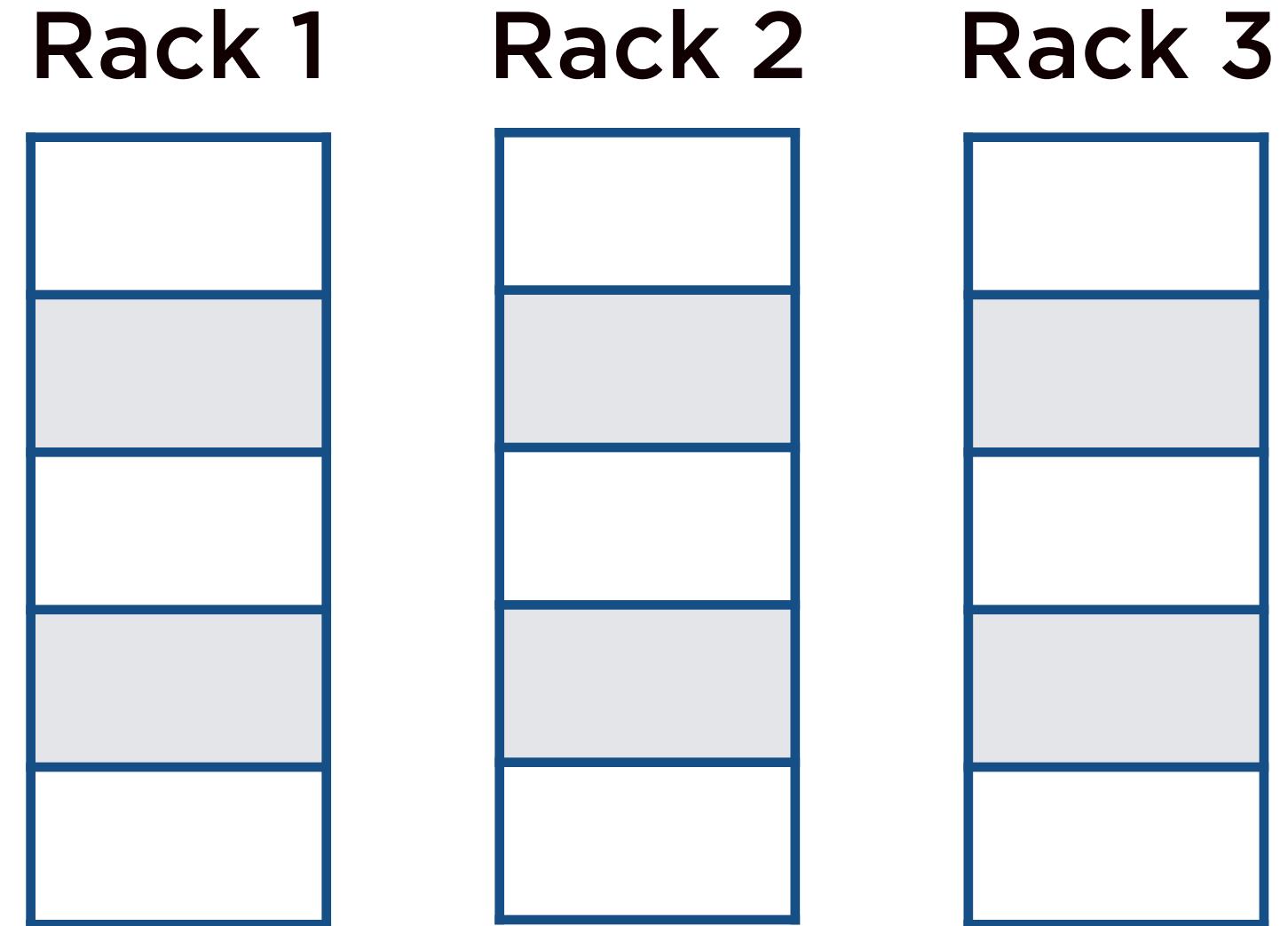


Maximize  
redundancy

Minimize  
write  
bandwidth



**Maximize  
redundancy**

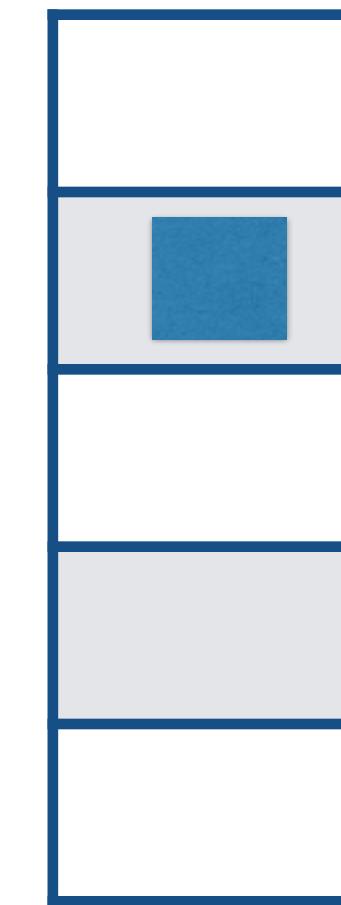
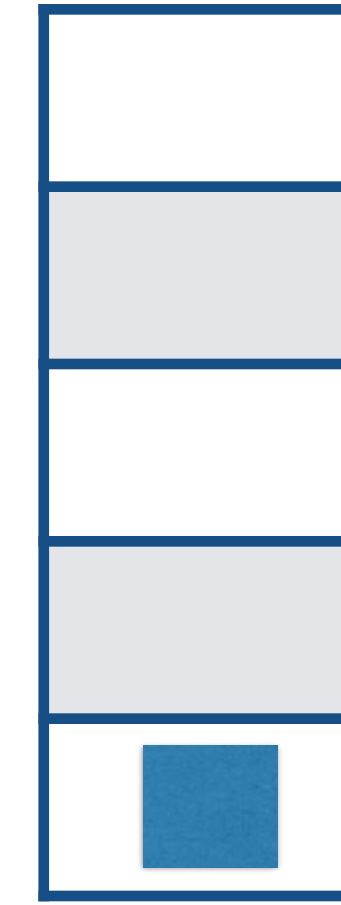
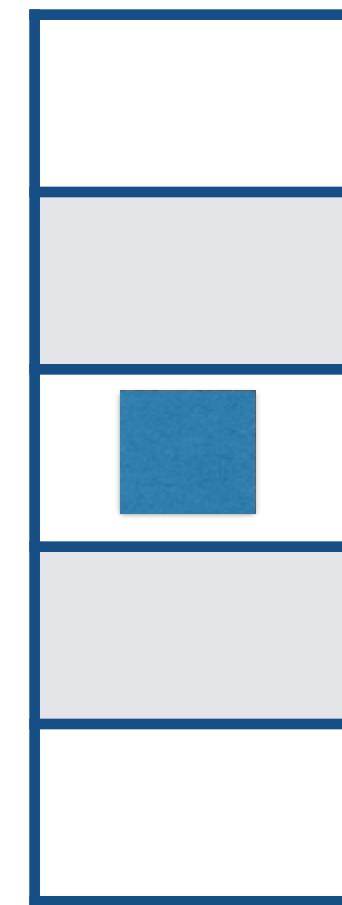


**Servers in a  
data center**



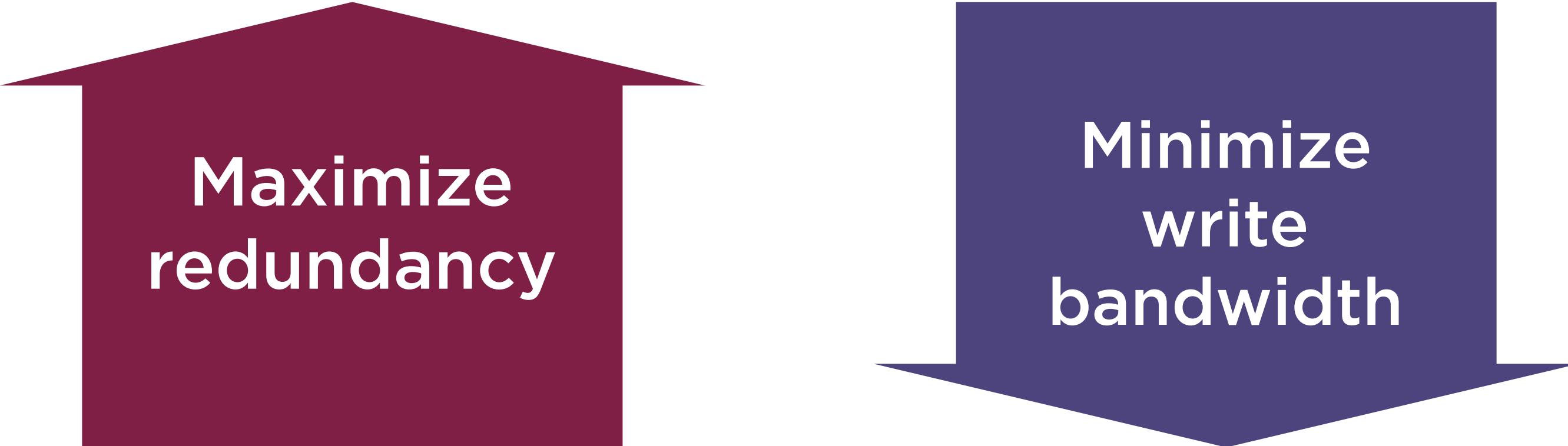
**Maximize  
redundancy**

## Rack 1      Rack 2      Rack 3



**Store replicas “far away”  
i.e. on different nodes**

# Choosing Replica Locations



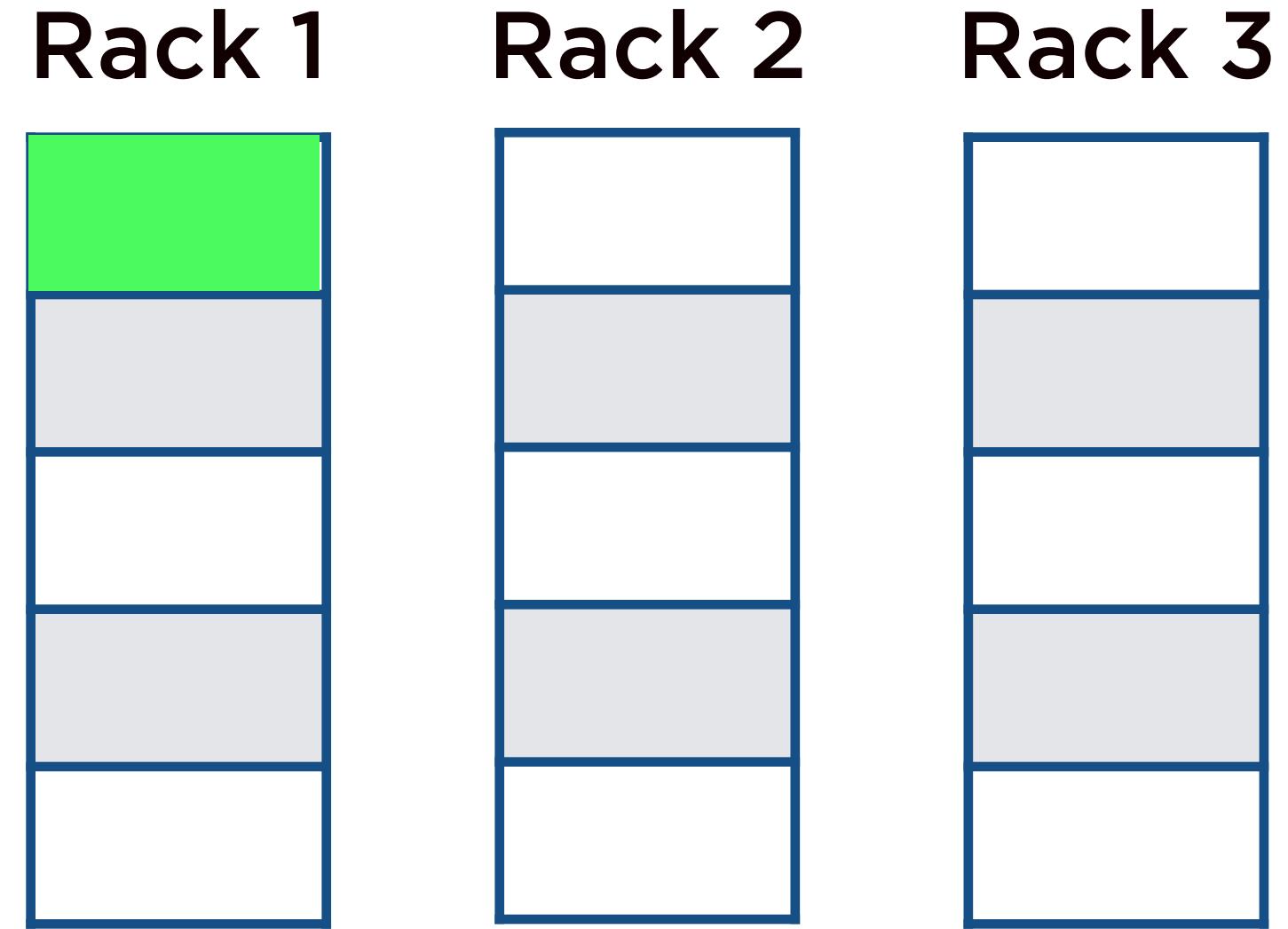
Maximize  
redundancy

Minimize  
write  
bandwidth

Minimize  
write  
bandwidth

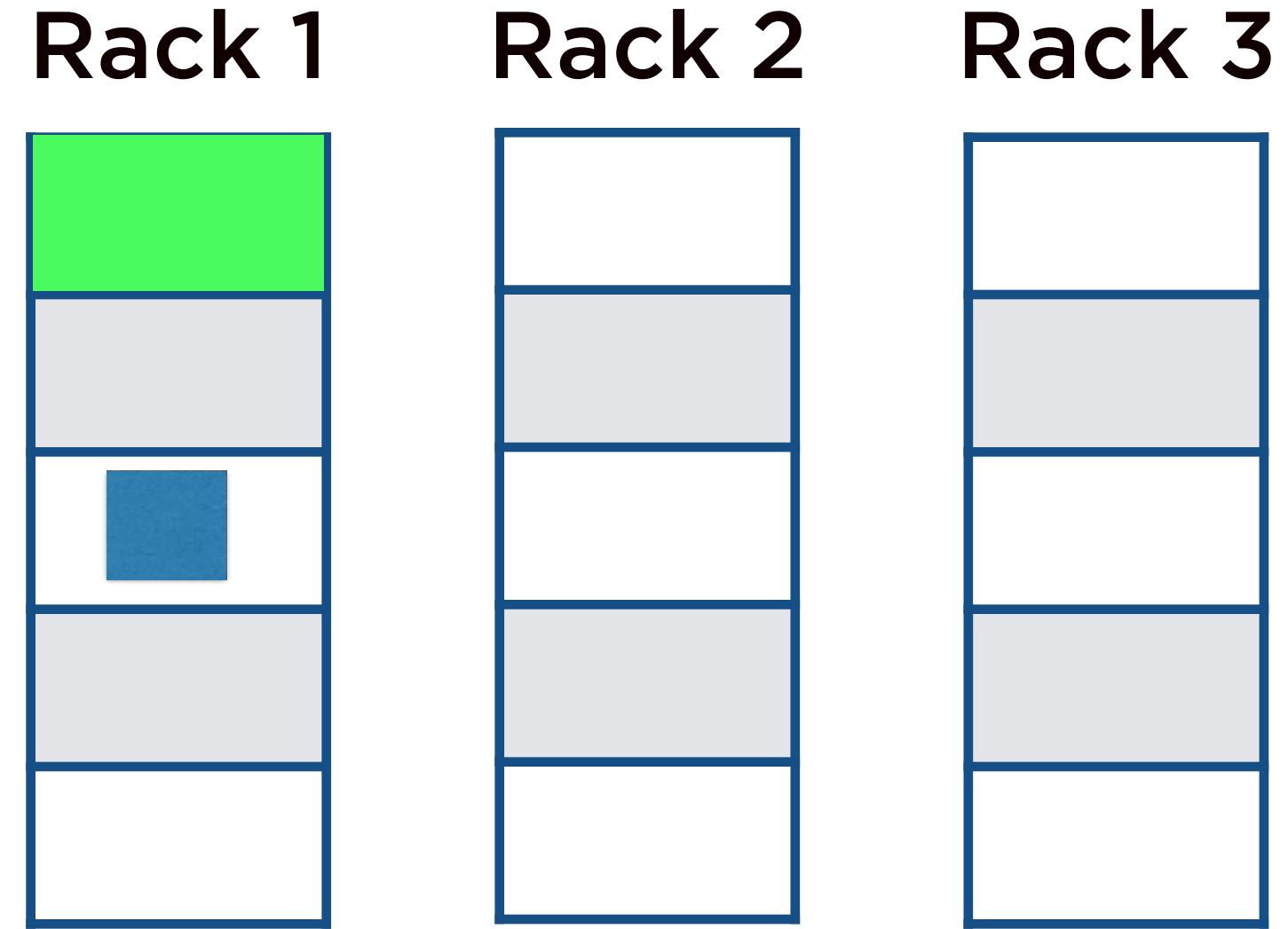
This requires that  
replicas be stored  
close to each other

Minimize  
write  
bandwidth



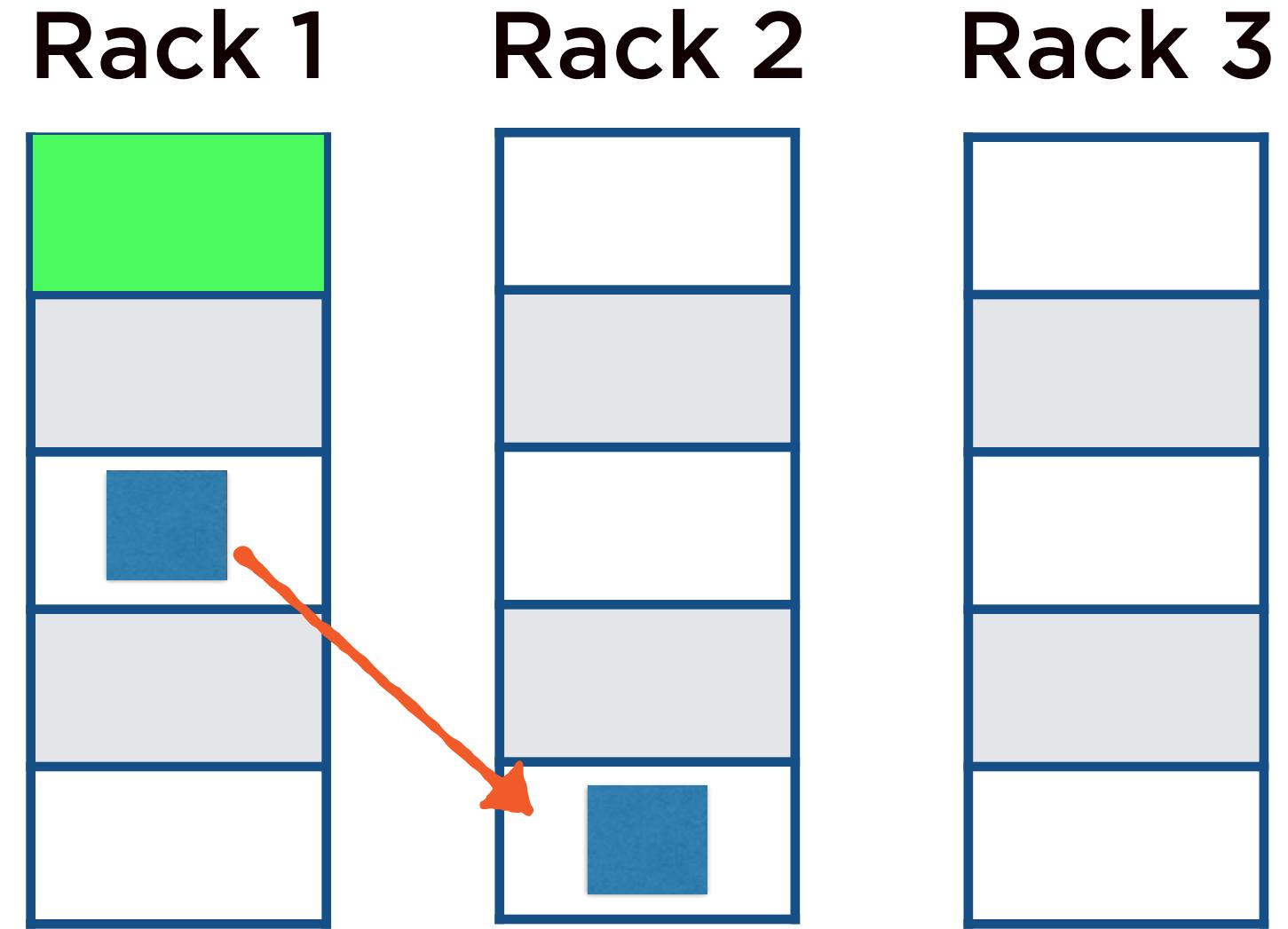
Node that runs the  
replication pipeline

Minimize  
write  
bandwidth



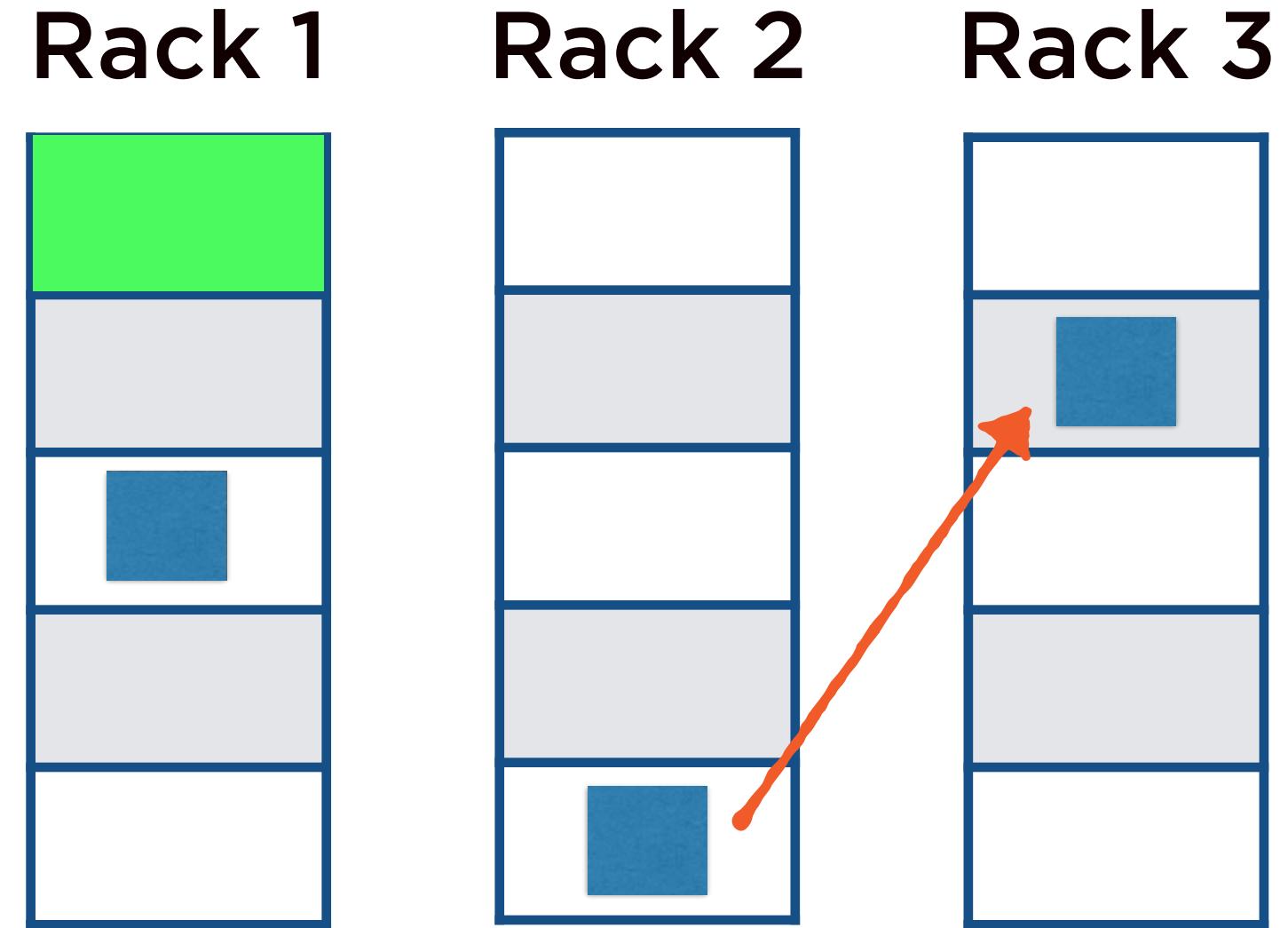
This node chooses  
the location for the  
first replica

Minimize  
write  
bandwidth



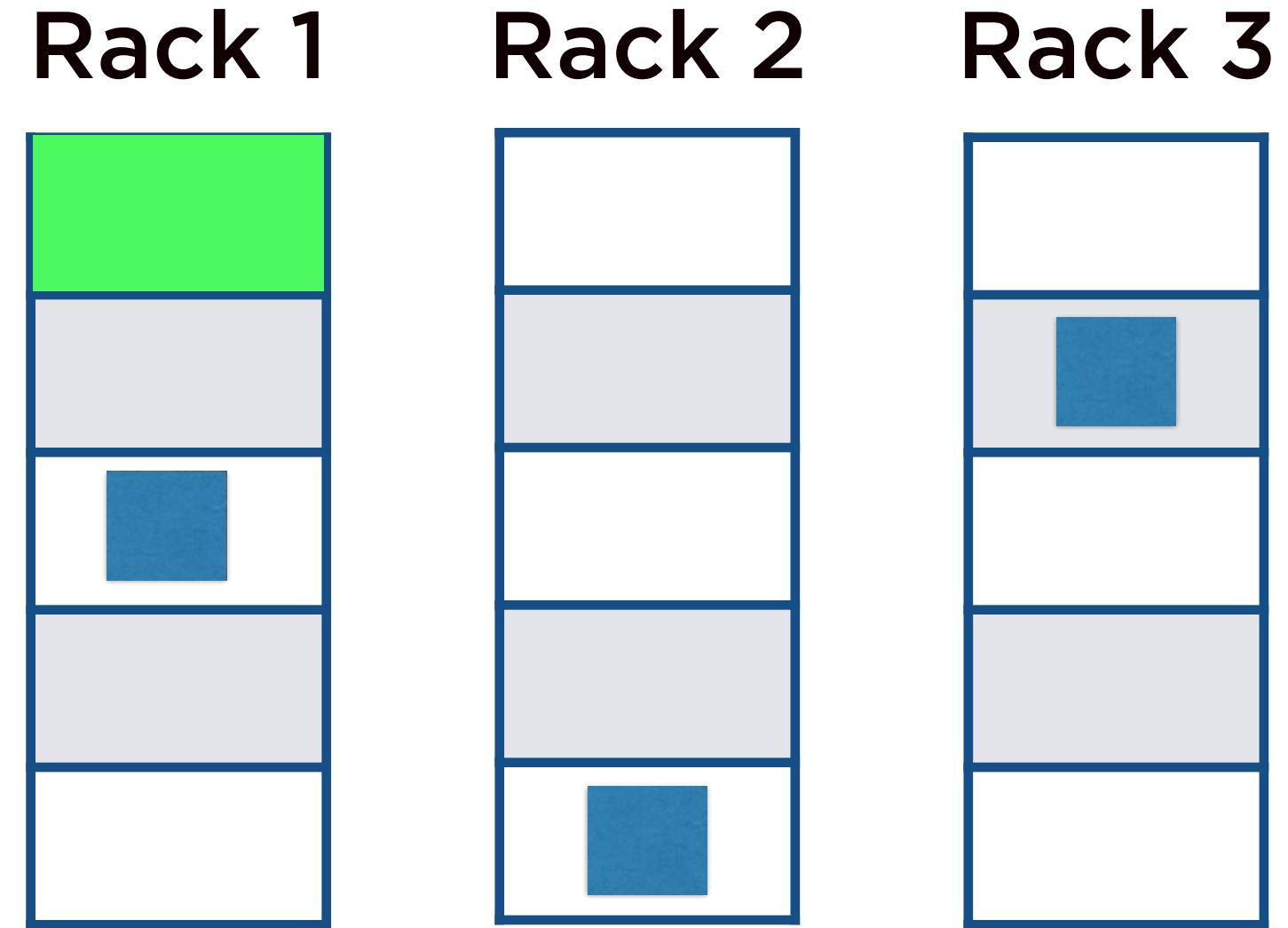
Data is forwarded  
from here to the  
next replica location

Minimize  
write  
bandwidth



Forwarded further  
to the next replica  
location

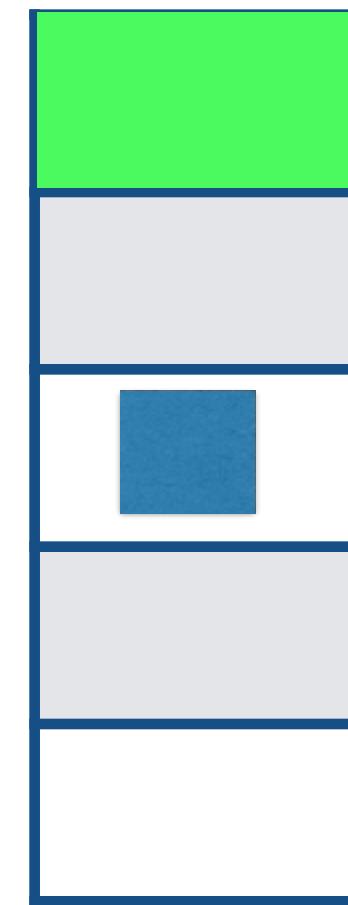
Minimize  
write  
bandwidth



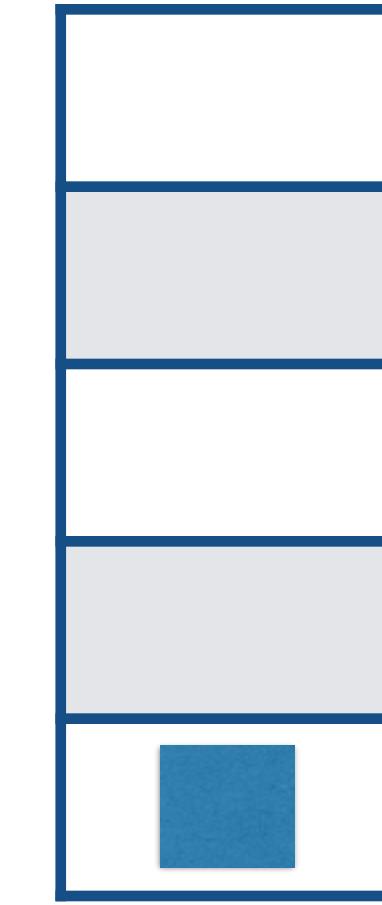
Forwarding requires  
a large amount of  
bandwidth

Minimize  
write  
bandwidth

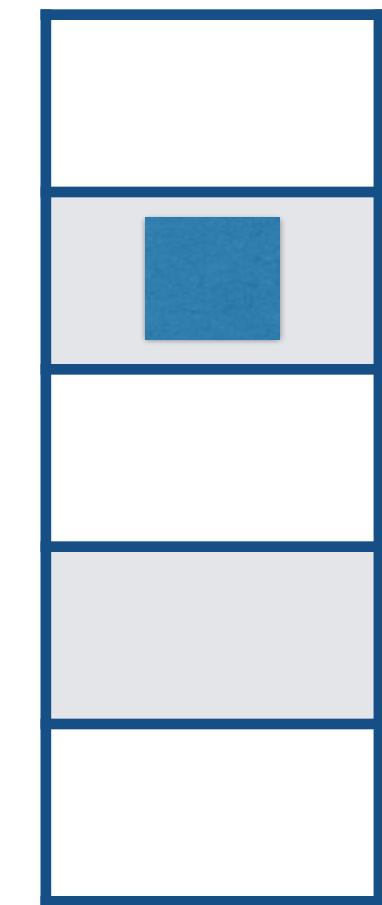
Rack 1



Rack 2

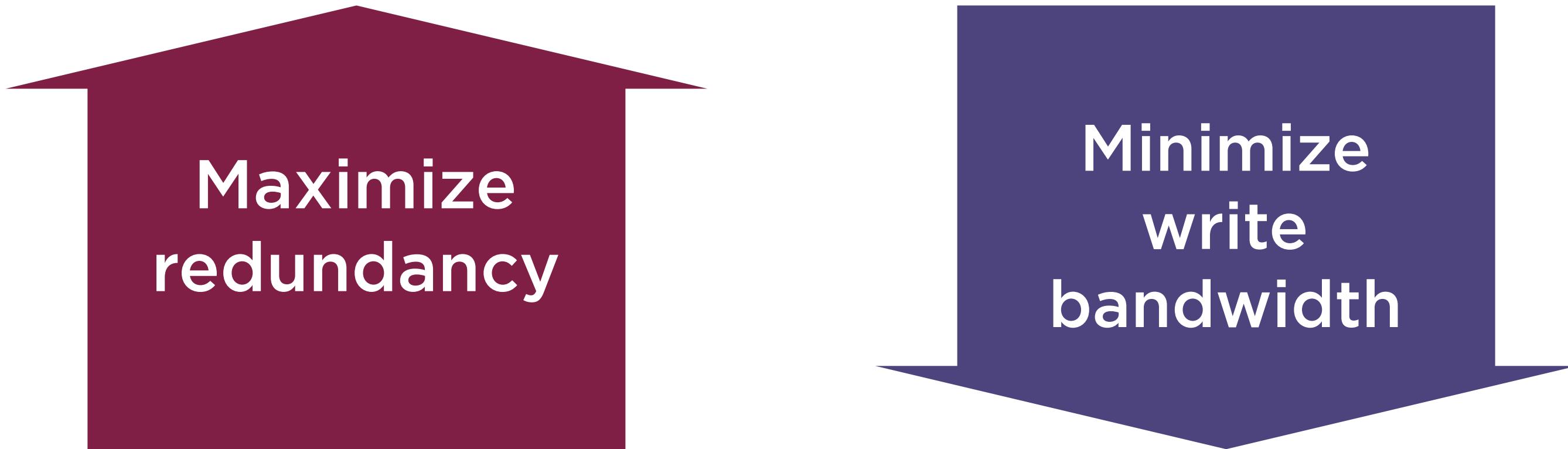


Rack 3



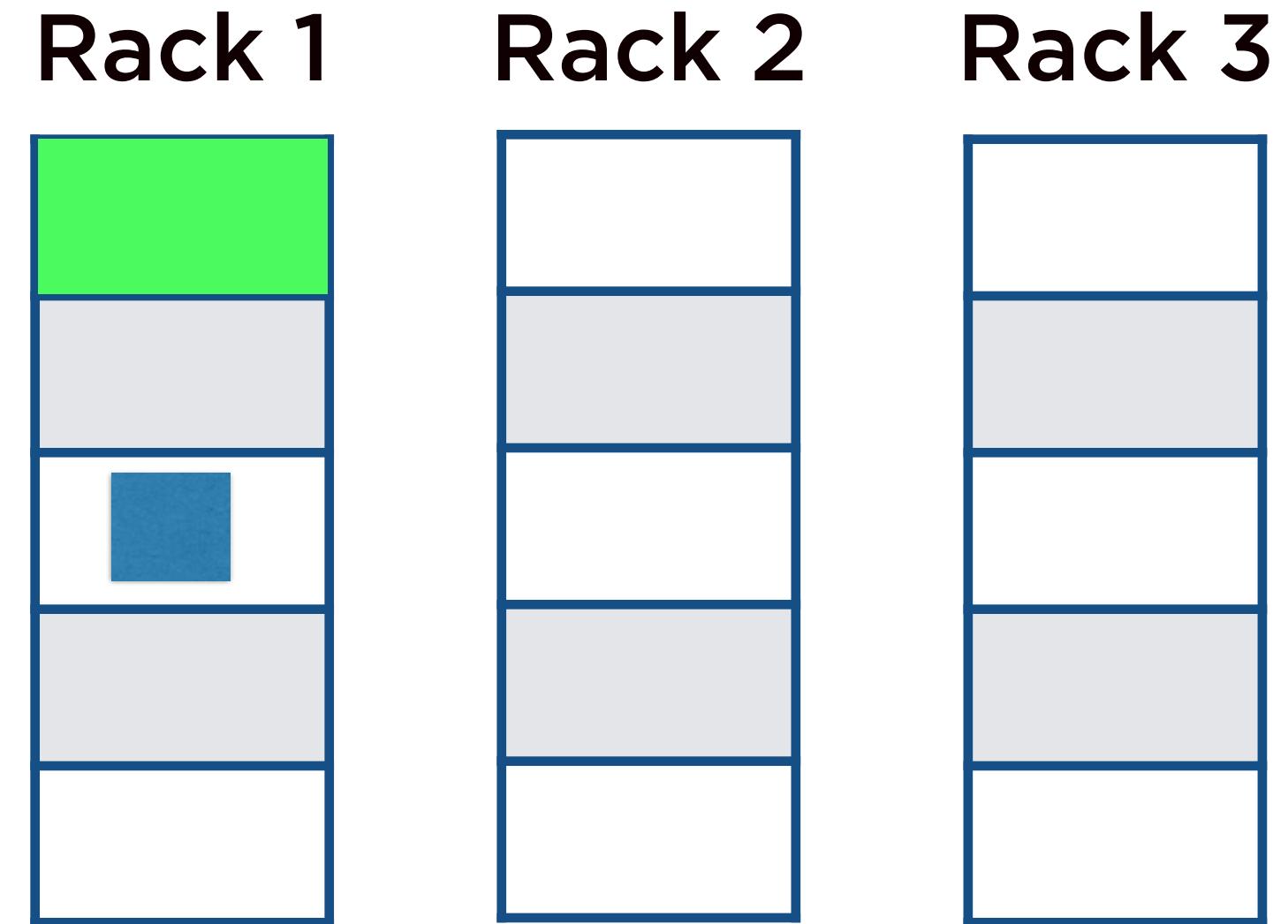
Increases the cost  
of write operations

# Default Hadoop Replication Strategy



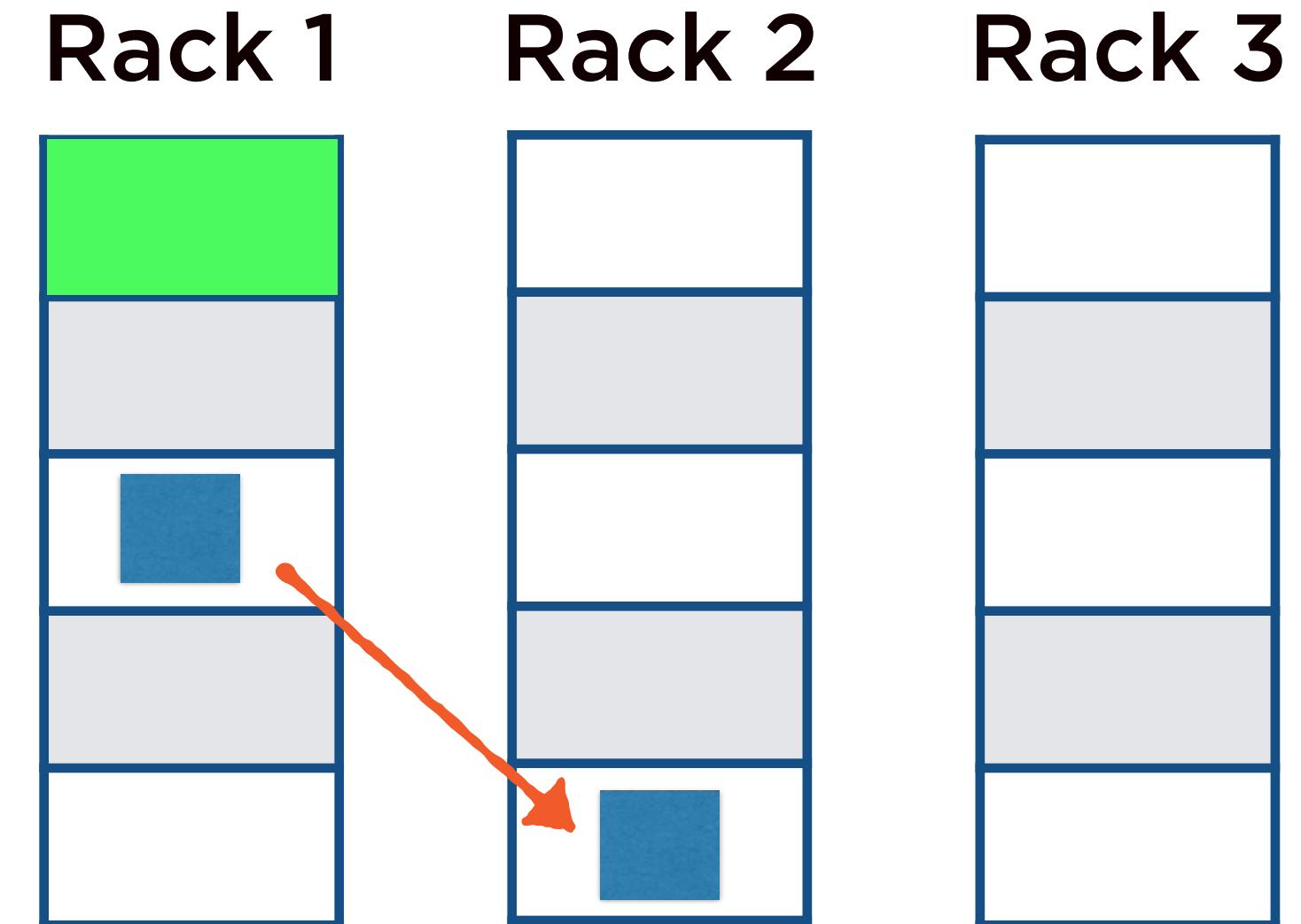
# Default Hadoop Replication Strategy

**First location  
chosen at random**



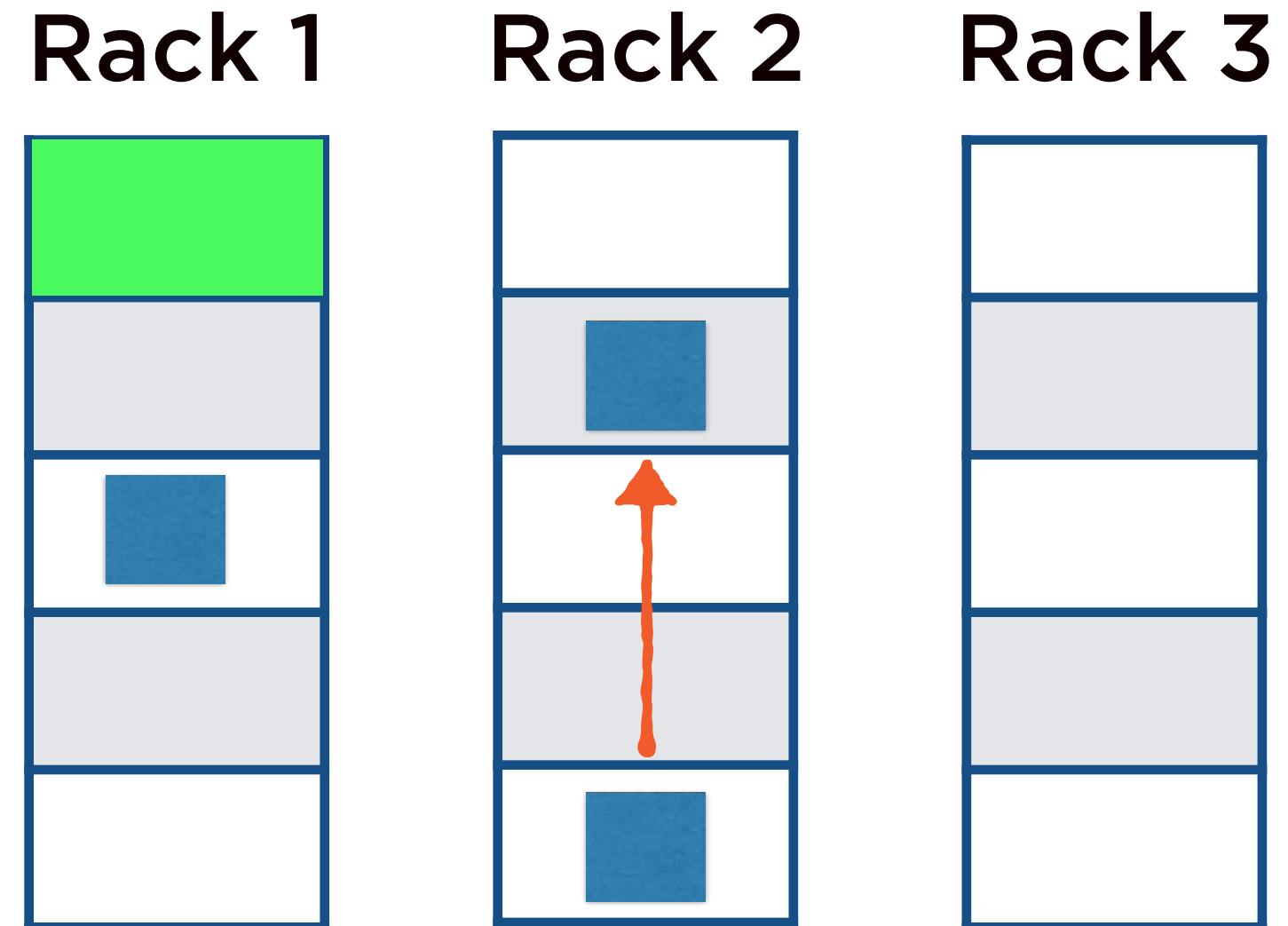
# Default Hadoop Replication Strategy

**Second location has  
to be on a different  
rack (if possible)**



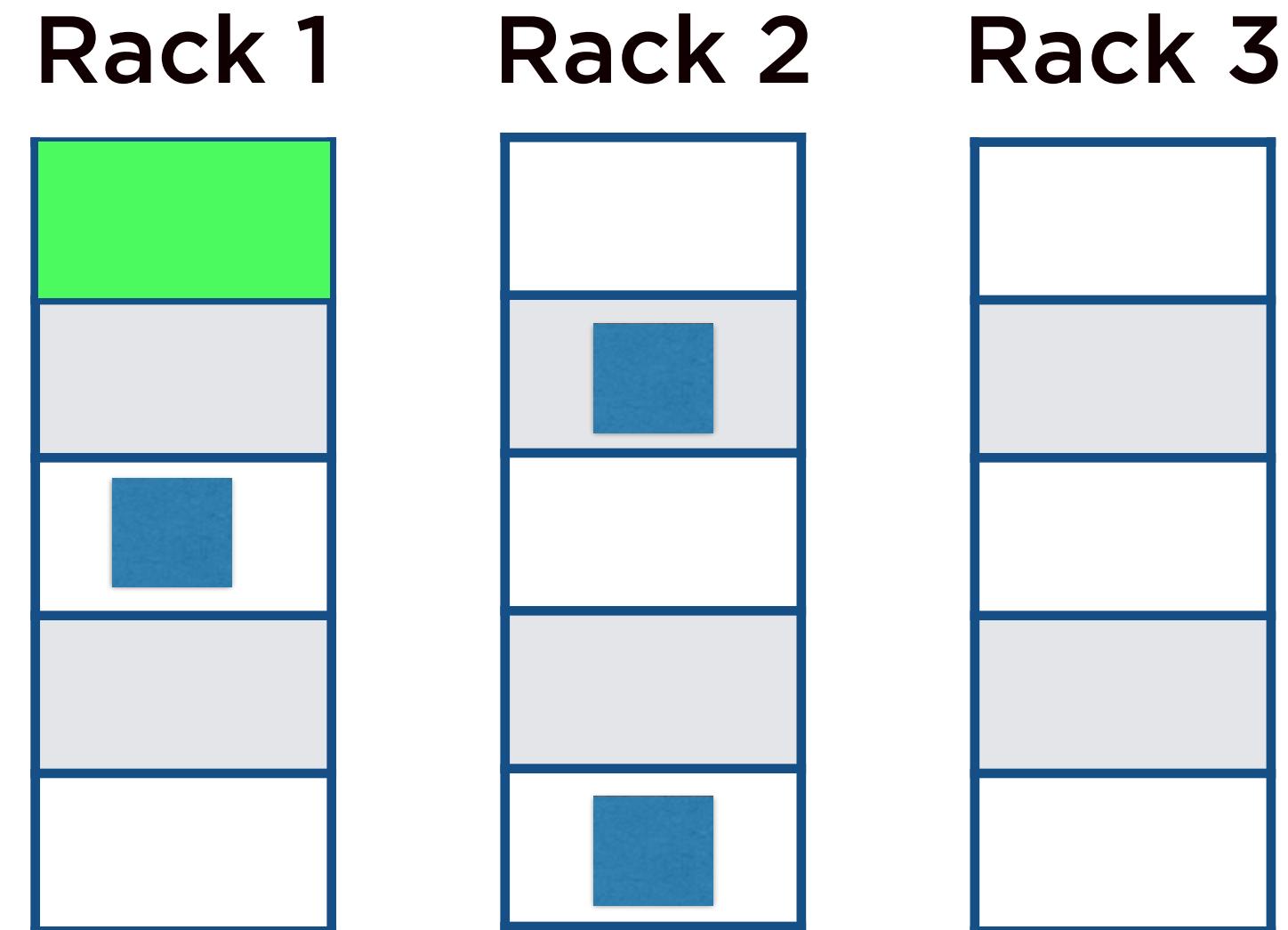
# Default Hadoop Replication Strategy

Third replica is on  
the same rack as  
the second but on  
different nodes



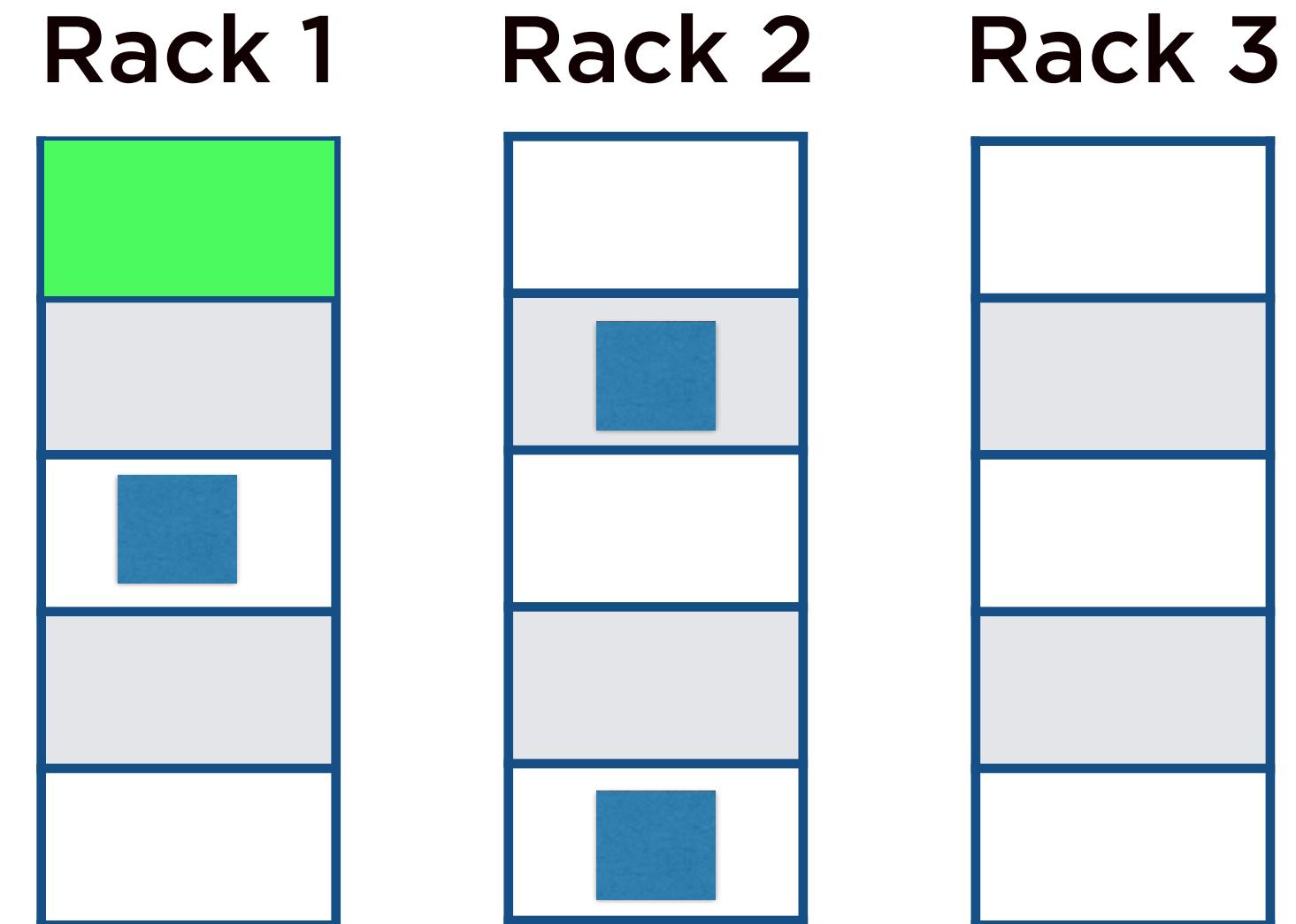
# Default Hadoop Replication Strategy

Reduces inter-rack traffic and improves write performance



# Default Hadoop Replication Strategy

Read operations  
are sent to the  
rack closest to  
the client



# Setting the Replication Factor

Defined in the configuration  
file `hdfs-site.xml`

# Setting the Replication Factor

**dfs.replication**

**3**

**This is the default in fully-distributed mode**

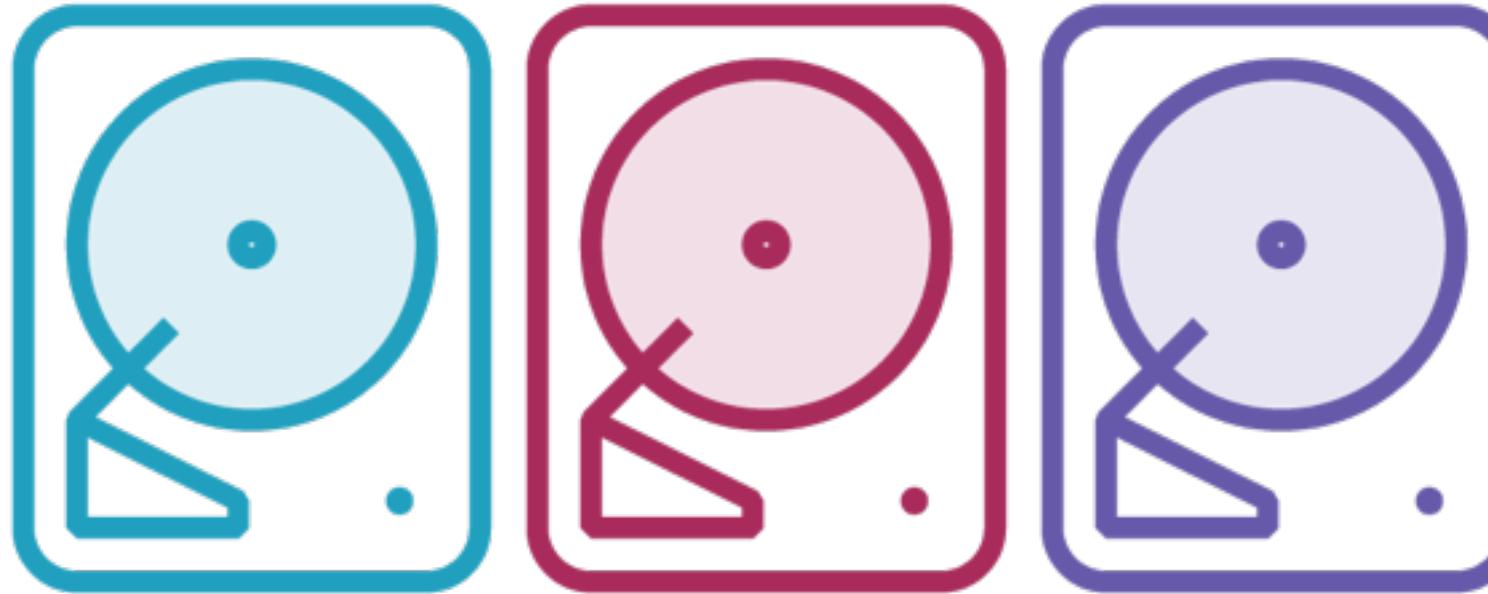
# Setting the Replication Factor

**dfs.replication**

**1**

**The pseudo-distributed mode has just one node so the replication factor cannot be >1**

# Challenges of Distributed Storage



**Failure management  
in the data nodes**

**Failure management  
for the name node**

# Name Node Failures

**The name node is  
the heart of HDFS**

Name node

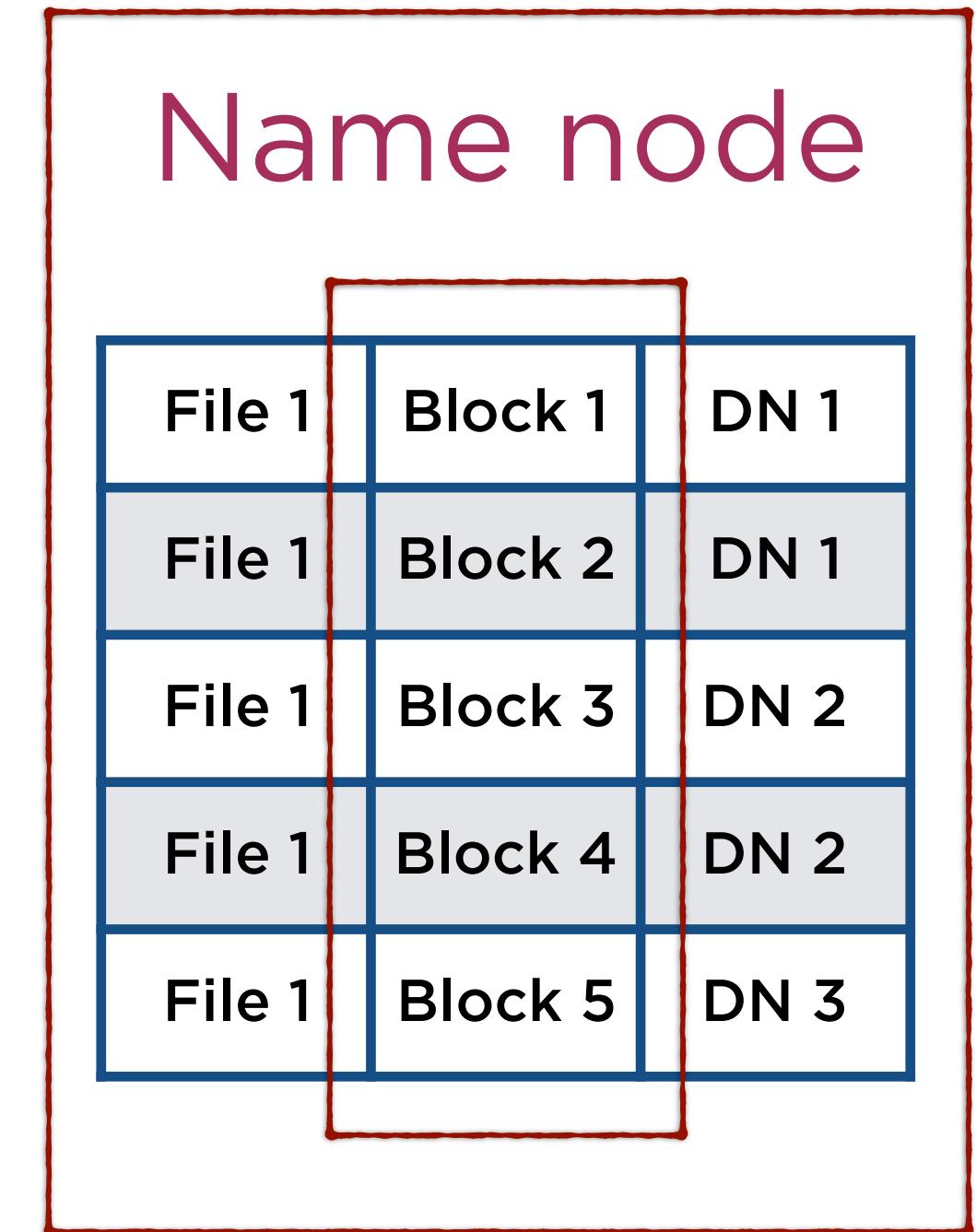
File 1	Block 1	DN 1
File 1	Block 2	DN 1
File 1	Block 3	DN 2
File 1	Block 4	DN 2
File 1	Block 5	DN 3

# Name Node Failures

**Block locations are  
not persistent**

i.e. they are stored  
in memory

**block caching**



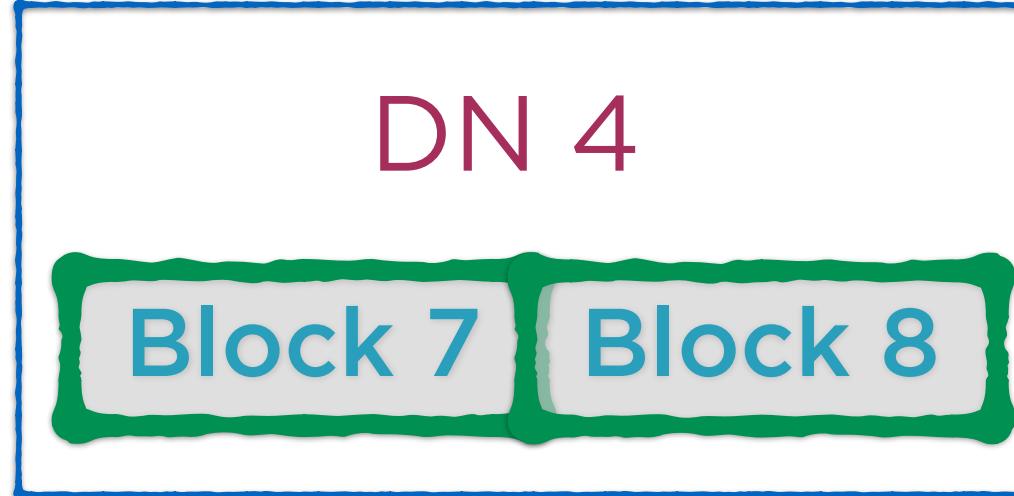
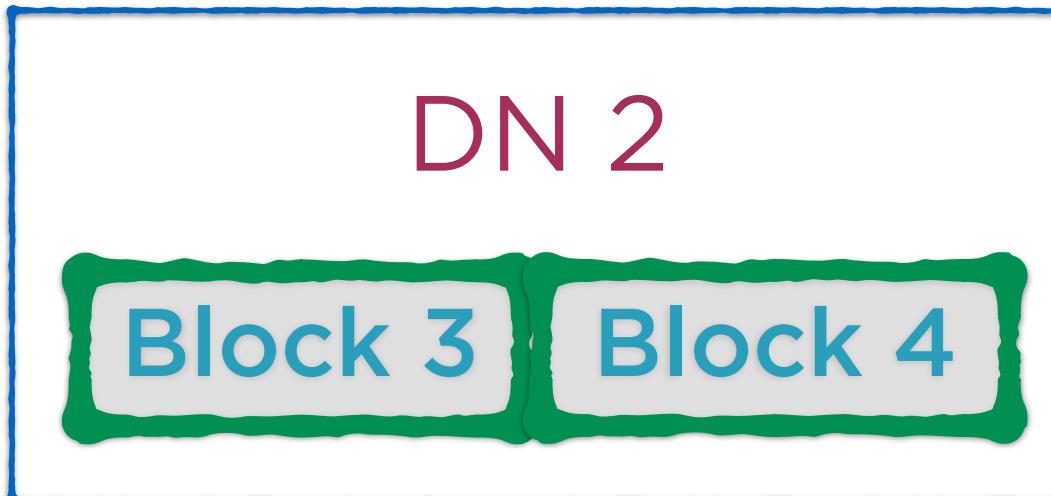
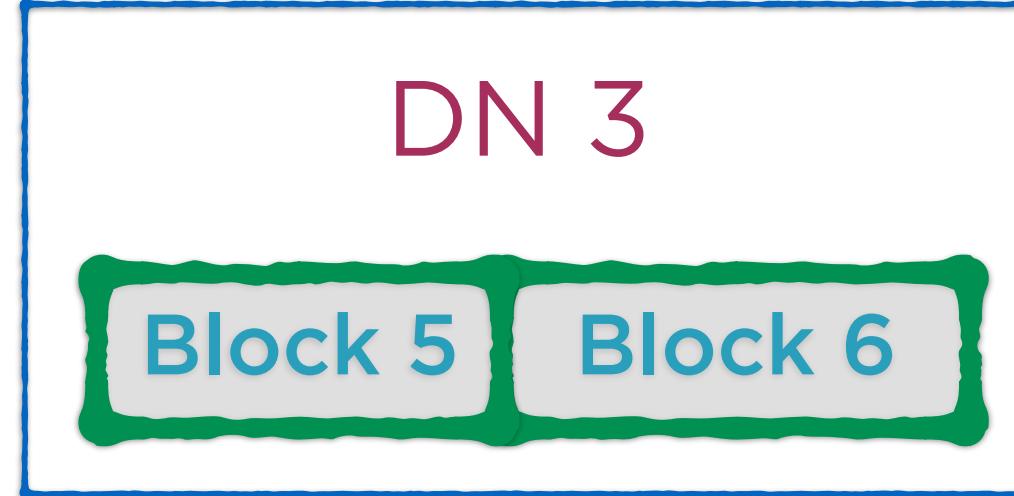
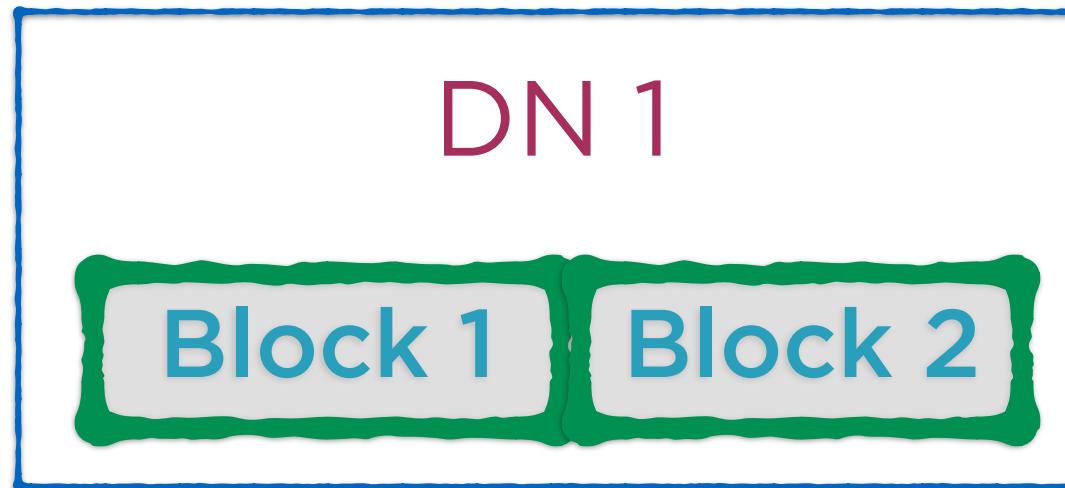
# Name Node Failures

If the name node fails

File-Block Location  
mapping is lost!

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

# Name Node Failures



This data is  
worthless  
without the  
name node

# Managing Name Node Failures

**Metadata Files**

**Secondary Name  
Node**



Metadata Files

# **fsimage edits**

**Two files that store  
the filesystem  
metadata**

# fsimage



A snapshot of the complete  
file system at start up

Loaded into memory

# edits



A log of all in-memory edits to the file system



Metadata Files

# **fsimage edits**

These together can  
help reconstruct a  
name node

# **fsimage edits**

**Default backup location**

**Name node  
local file system**

**Metadata Files**

# **fsimage edits**

**Metadata Files**

**Alternative backup location**

**A remote drive**

# Configuring the Backup Location

**Set the property**

**dfs.namenode.name.dir**

**in hdfs-site.xml**

# Configuring the Backup Location

**dfs.namenode.name.dir**

**\$PATH\_1,\$PATH\_2,\$PATH\_3**

**A comma separated list of paths**

# Configuring the Backup Location

**dfs.namenode.name.dir**

**\$PATH\_1,\$PATH\_2,\$PATH\_3**

**Each can be a different host**

# Configuring the Backup Location

**dfs.namenode.name.dir**

**\$PATH\_1,\$PATH\_2,\$PATH\_3**

**Metadata files will be backed up to each path**



Metadata Files

# fsimage edits

- Merging these 2 files is very compute heavy
- Bringing a system back online could take a long time

# Managing Name Node Failures

**Metadata Files**

**Secondary Name  
Node**

# Secondary Name Node

Name Node

**fsimage**

**edits**

Secondary Name  
Node

**fsimage**

**edits**

# Checkpointing

Name Node

**fsimage**

**edits**

Secondary Name  
Node

**fsimage**

**edits**

# Merge

# Checkpointing

Name Node

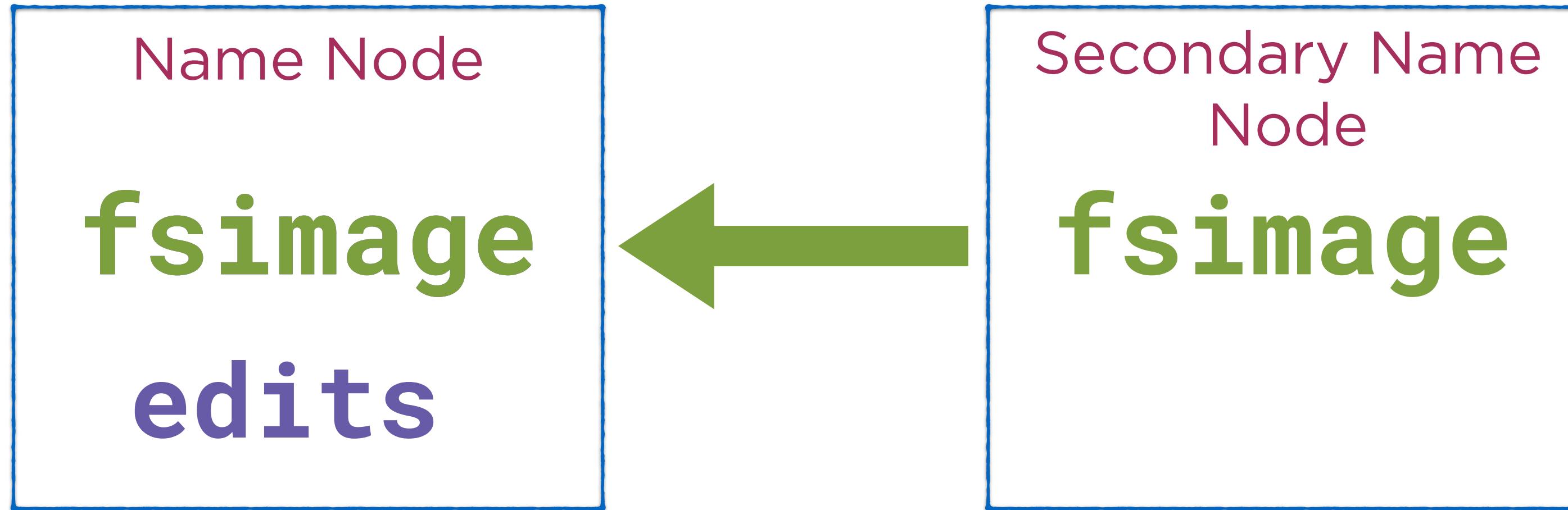
**fsimage**

**edits**

Secondary Name  
Node

**fsimage**

# Checkpointing



**Copy to name node**

# Checkpointing

Name Node

**fsimage**

**edits**

Secondary Name  
Node

**fsimage**

**edits**

**Reset the edits file to empty on  
both nodes**

# Checkpointing

Name Node

**fsimage**

**edits**

Secondary Name  
Node

**fsimage**

**edits**

**Checkpointing is done  
at a specified frequency**

# Failure Management

Name Node

**fsimage**

**edits**

If there is  
a failure

Secondary Name  
Node

**fsimage**

**edits**

This becomes  
the name node

# Configuring the Checkpoint Frequency

**Set properties in hdfs-site.xml**

# Configuring the Checkpoint Frequency

**dfs.namenode.checkpoint.period**

**The number of seconds between each checkpoint**

# Configuring the Checkpoint Frequency

**dfs.namenode.checkpoint.check.period**

The period when the secondary name node  
polls the name node for uncheckpointed  
transactions

# Configuring the Checkpoint Frequency

**dfs.namenode.checkpoint.txns**

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

# Summary

**Understood the components of HDFS**

**Moved files in and out of HDFS**

**Understood replication strategies for data nodes**

**Understood failure management strategies for the name node**