3 seminal papers

Google File System

MapReduce

BigTable

3 seminal papers

Google File System

MapReduce

BigTable

These are all technologies built to power Google Search

3 seminal papers

Google File System

MapReduce

BigTable

Each of these papers proposed an architecture for an important distributed computing problem

3 seminal papers

proposed an architecture for

Google File System — Storage

MapReduce — Processing data

BigTable — Database management

# 3 seminal papers

| | |
|---|---|
| Google File System | Storage |
| MapReduce | Processing data |
| BigTable | Database management |

**All of these architectures abstract programmers from the complexity of distributed computing**

# 3 seminal papers

Google File System

MapReduce

BigTable

Storage

Processing data

Database management

# Hadoop ecosystem

An ecosystem of Open source softwares based on these architectures

| 3 seminal papers | | Hadoop ecosystem |
| --- | --- | --- |
| Google File System<br>Storage | → | HDFS |
| MapReduce<br>Processing data | → | Hadoop MapReduce |
| BigTable<br>Database management | → | HBase |

# Hadoop ecosystem


HBase

HADOOP

# HADOOP

is a distributed computing framework
developed and maintained by

## THE APACHE SOFTWARE FOUNDATION

written in Java

# HADOOP

## HDFS

A file system to manage the storage of data

## MapReduce

A framework to process data across multiple servers

# HADOOP

## HDFS

A file system to manage the storage of data

## MapReduce

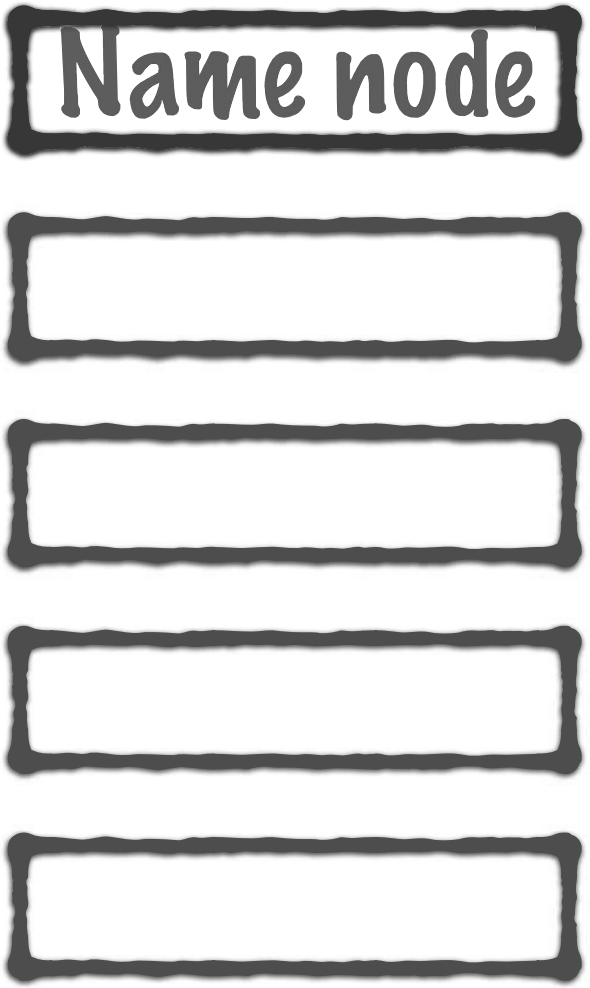
A framework to process data across multiple servers

# HDFS

## The Hadoop Distributed File System

Hadoop uses this to store data across multiple disks

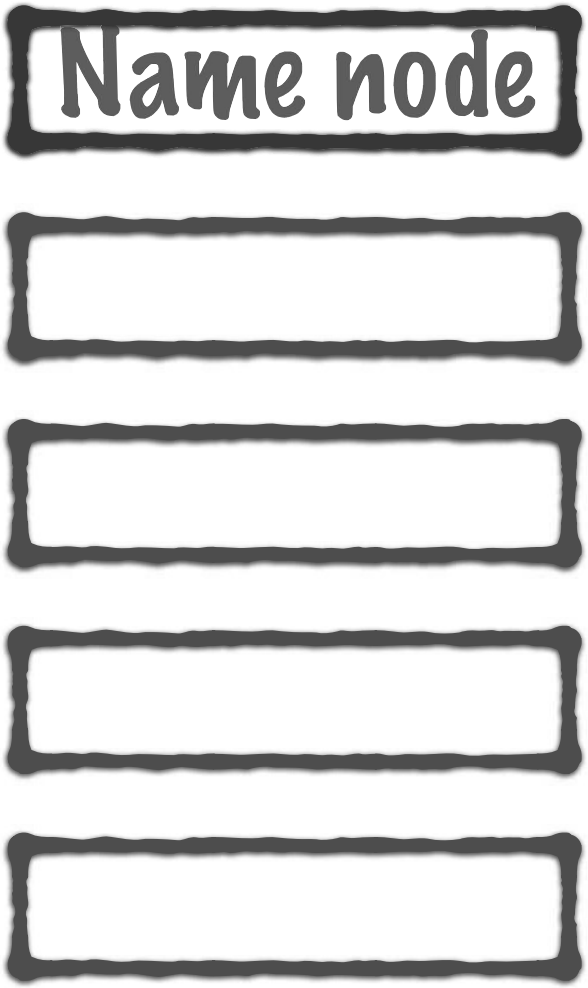
# HDFS




One of the nodes acts as the master node

This node manages the overall file system

# HDFS

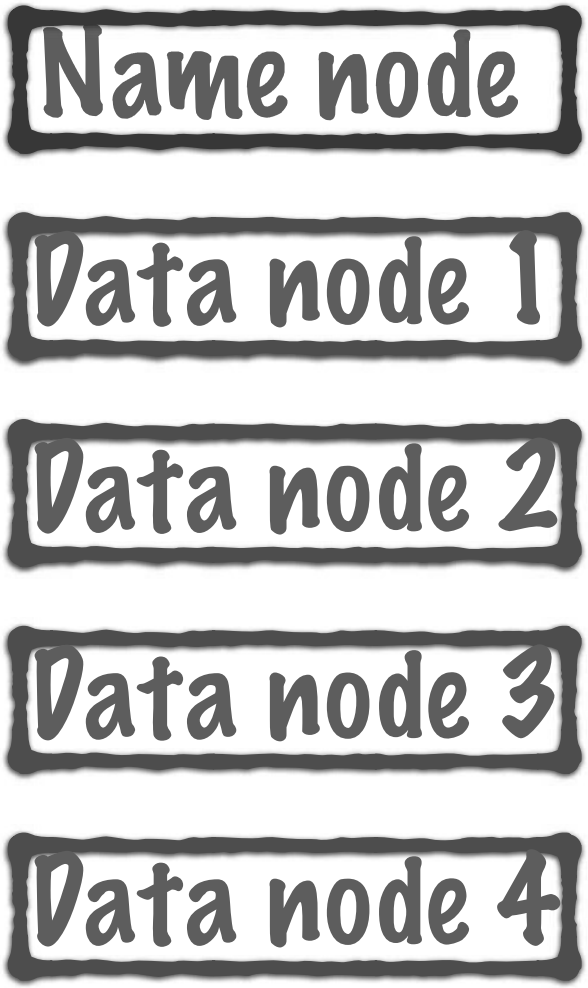



The name node stores

1. The directory structure

2. Metadata for all the files

# HDFS




Other nodes are called data nodes

The data is physically stored on these nodes

# 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.

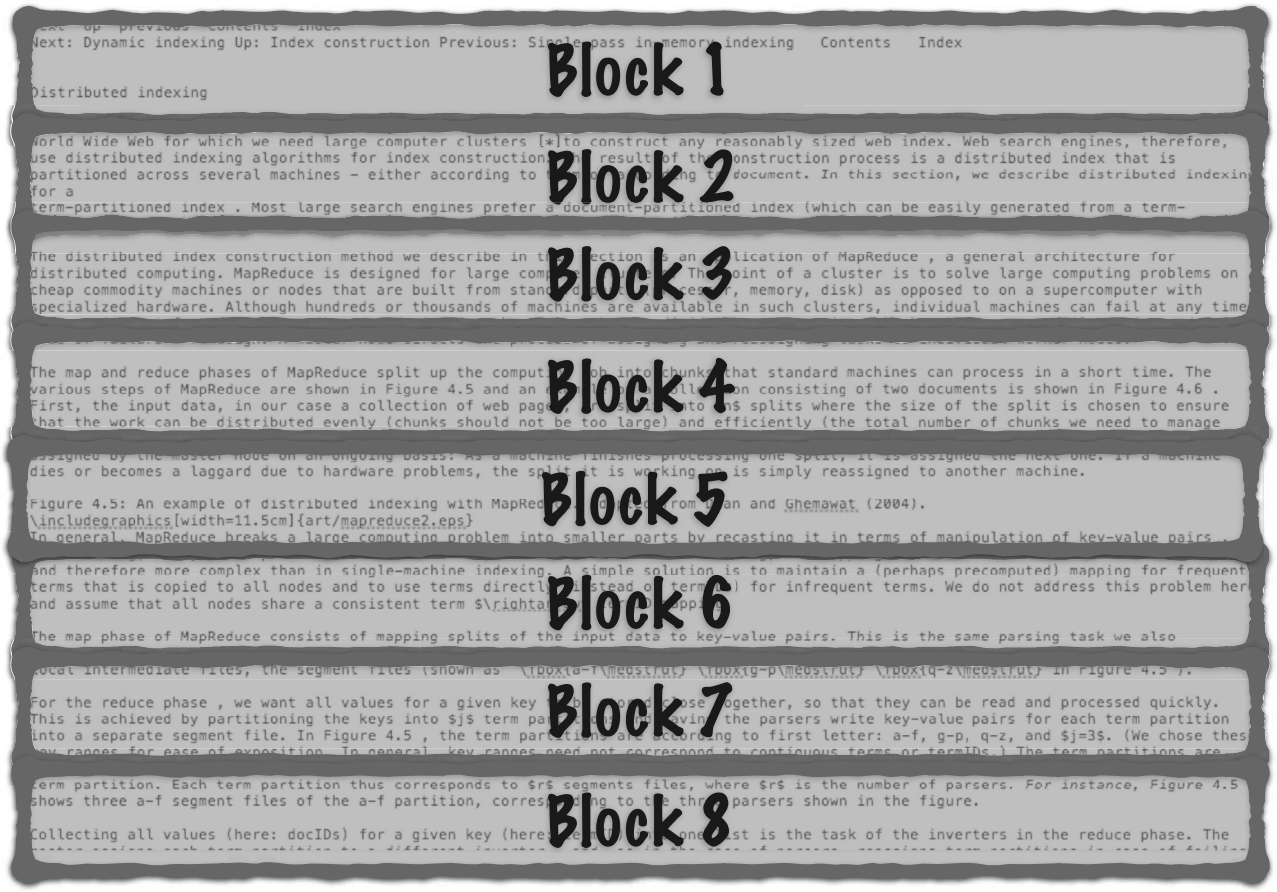
The map phase of MapReduce consists of mapping splits of the input data to key-value pairs. This is the same parsing task we also encountered in BSBI and SPIMI, and we therefore call the machines that execute the map phase parsers . Each parser writes its output to local intermediate files, the segment 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

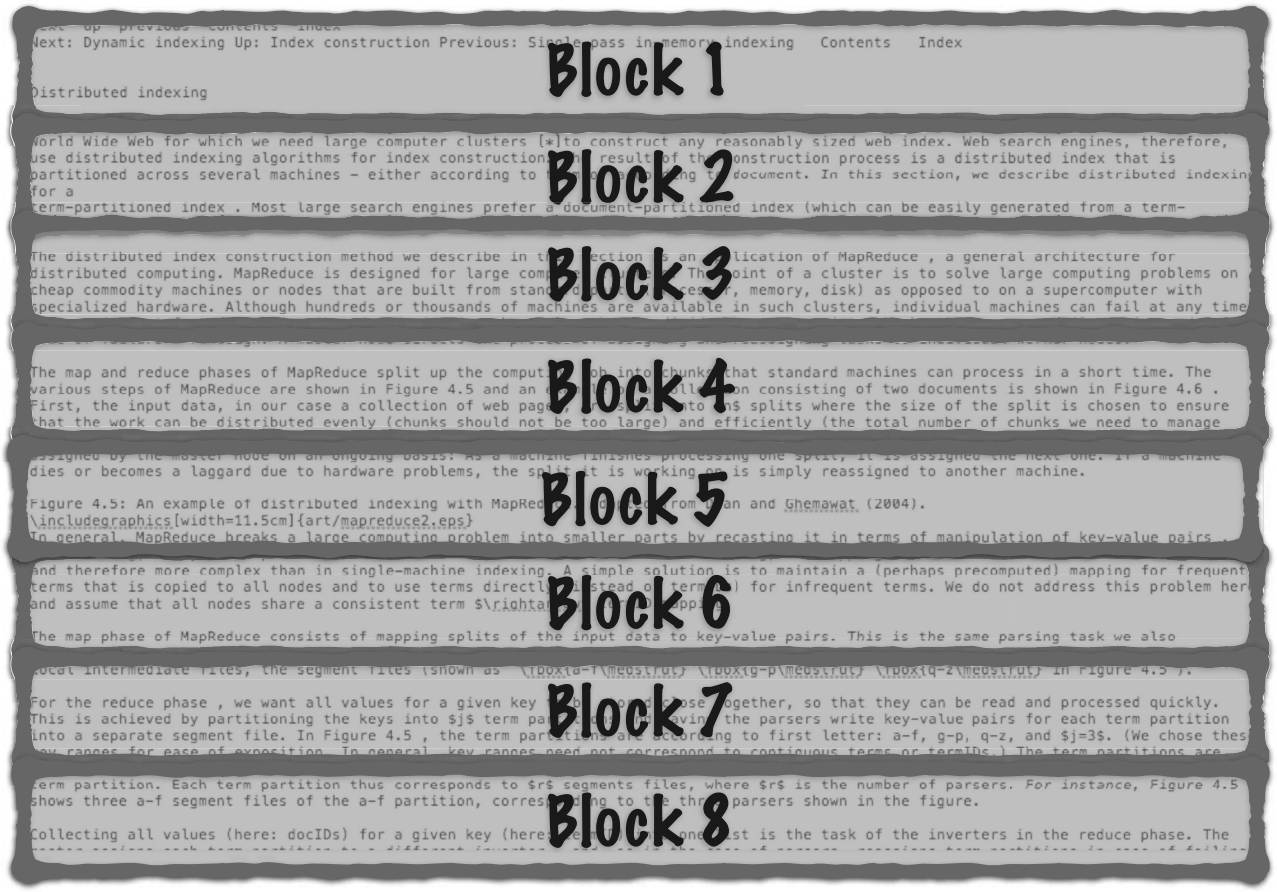
## Let's see how this file is stored in HDFS

# HDFS




First the file is broken into blocks of size 128 MB

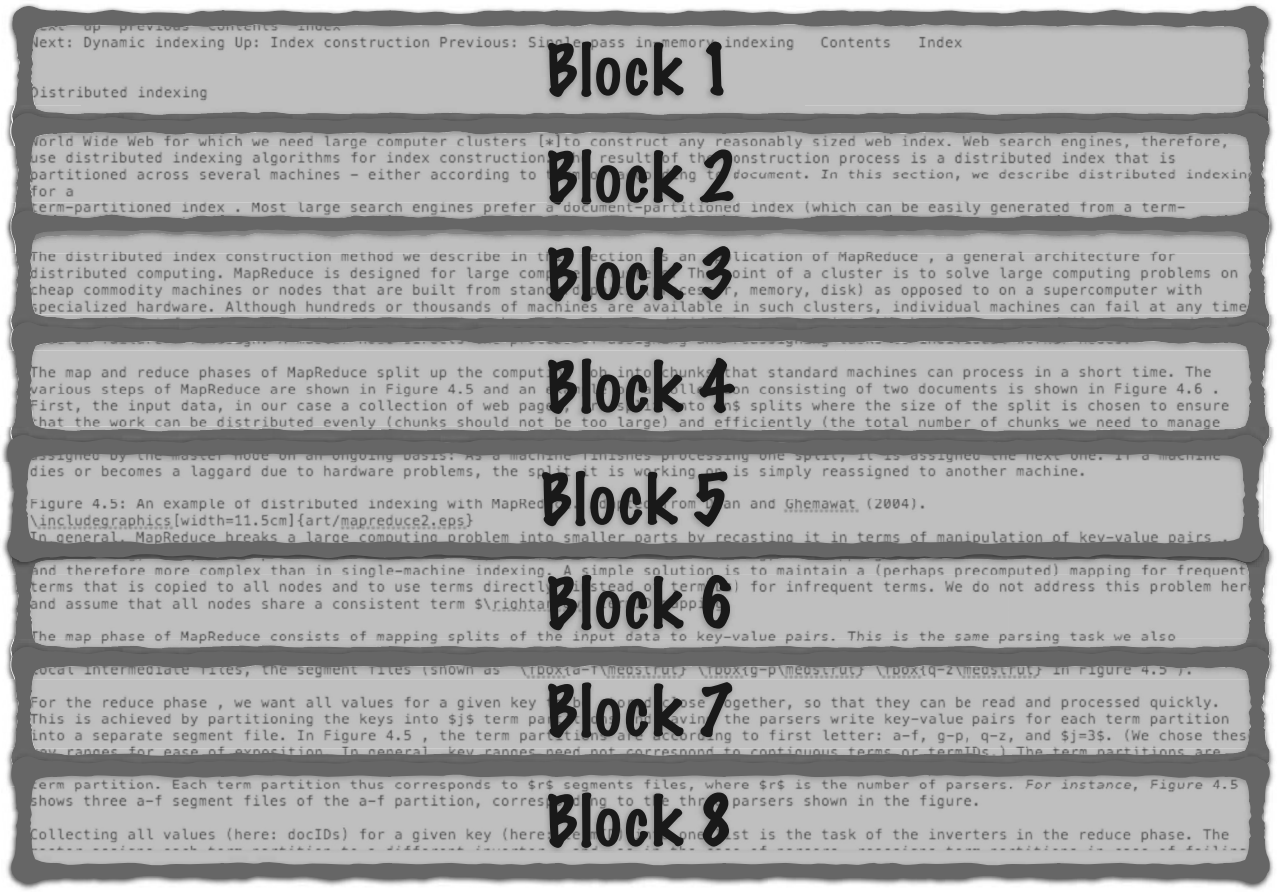
# HDFS




First the file is broken into blocks of size 128 MB

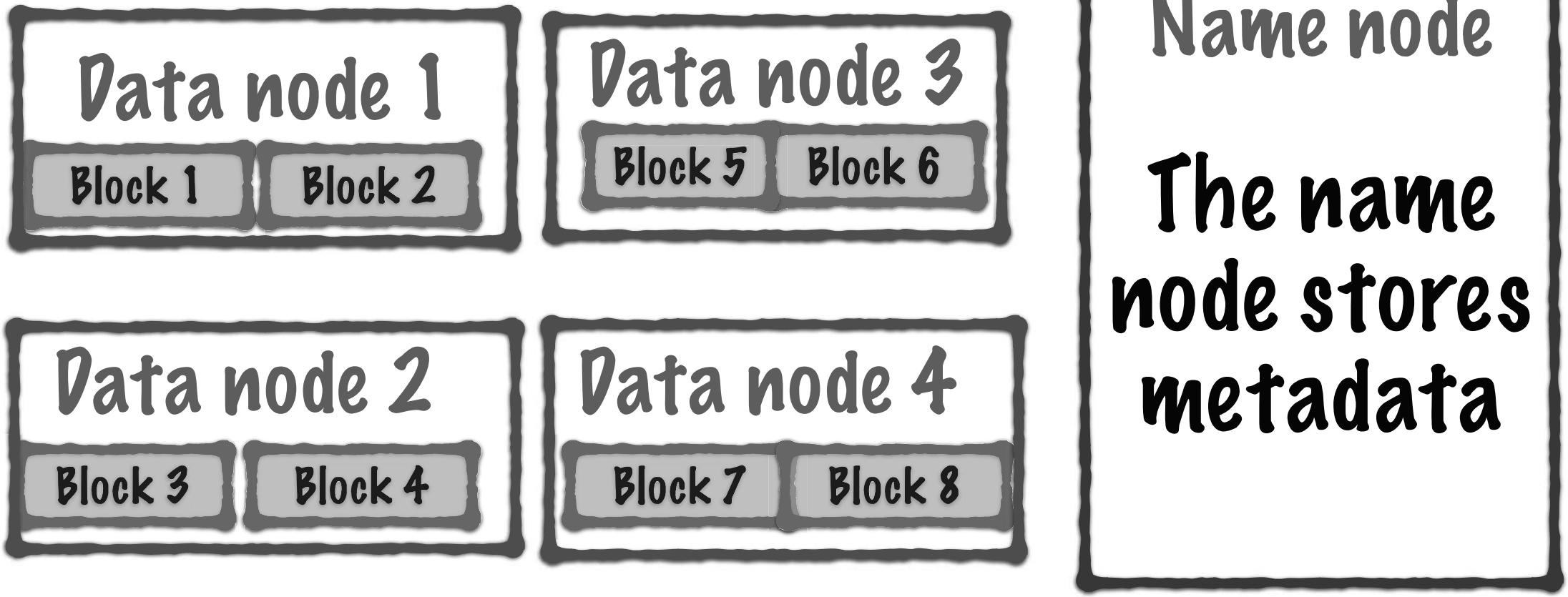
This size is chosen to minimize the time to seek to the block on the disk

# HDFS




These blocks are then stored across the data nodes

# 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

# 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 |

# 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 |

# HDFS

Data node 3

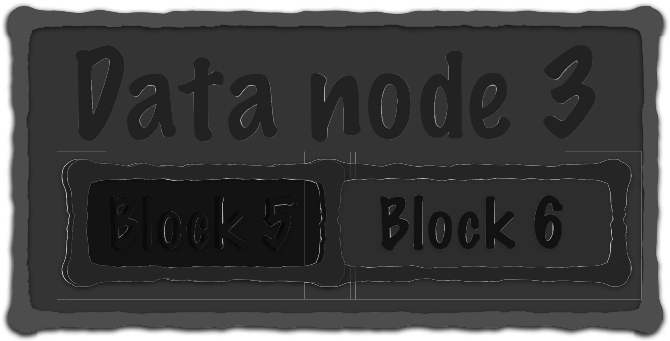
Block 5 | Block 6

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 |

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

# HDFS

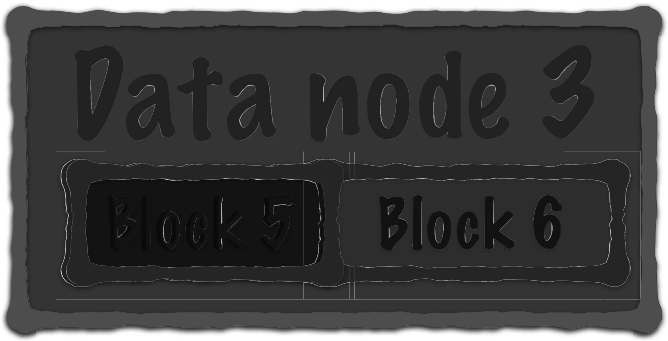


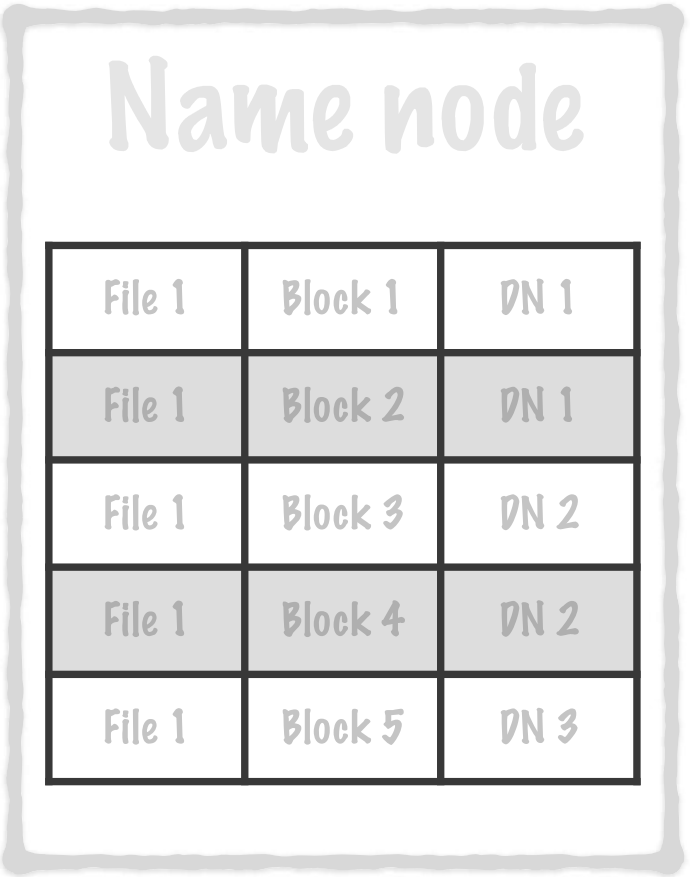

Name node

| 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?

# HDFS




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

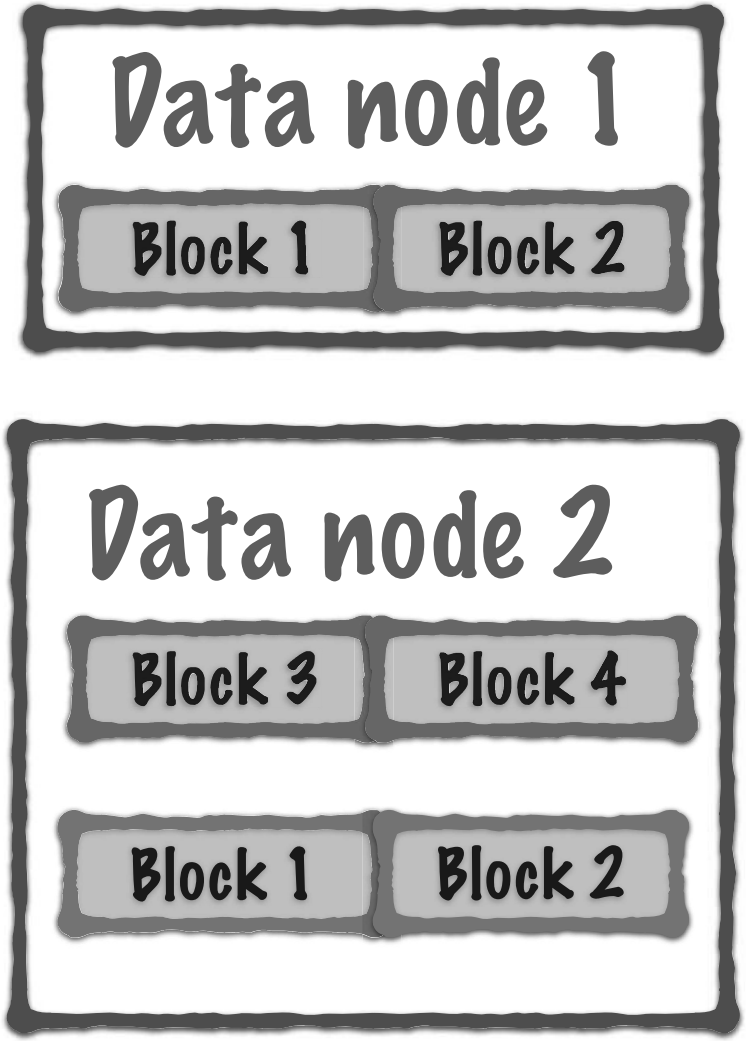
Block 8

## 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




Each block is replicated, and the replicas are stored in different data nodes

# HDFS


The replica locations are also stored in the name node

## Name node

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

# HADOOP

## HDFS

A file system to manage the storage of data

## MapReduce

A framework to process data across multiple servers

# HADOOP

## MapReduce

A framework to process data across multiple servers

## HDFS

A file system to manage the storage of data

# MapReduce

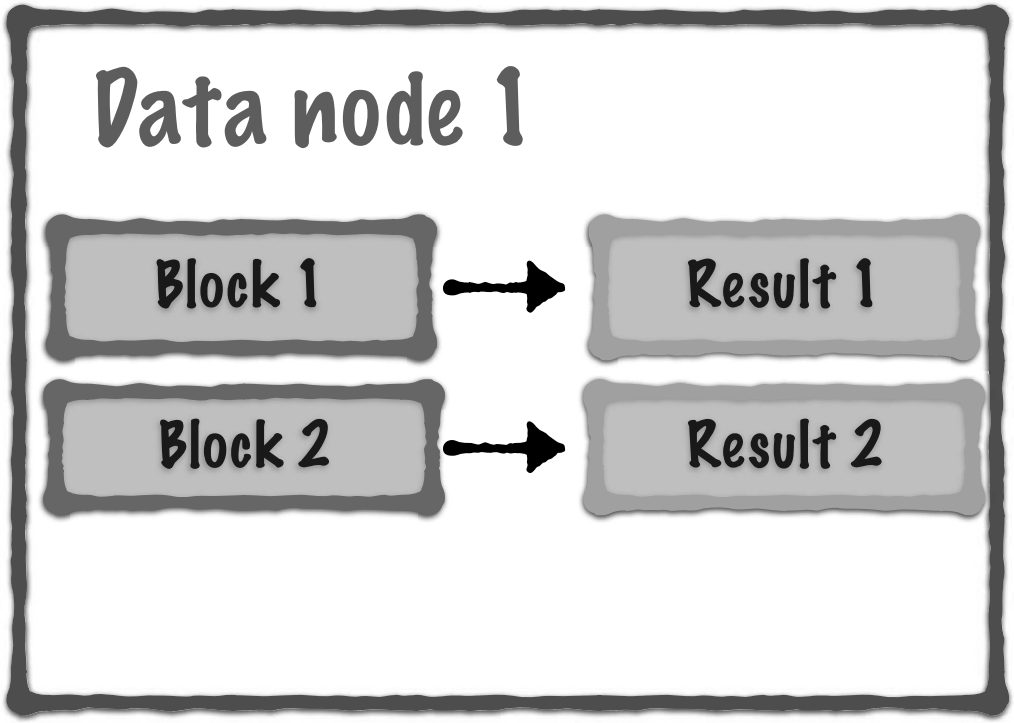
MapReduce is a way to parallelize a data processing task

# MapReduce

# MapReduce tasks have 2 phases

# MapReduce

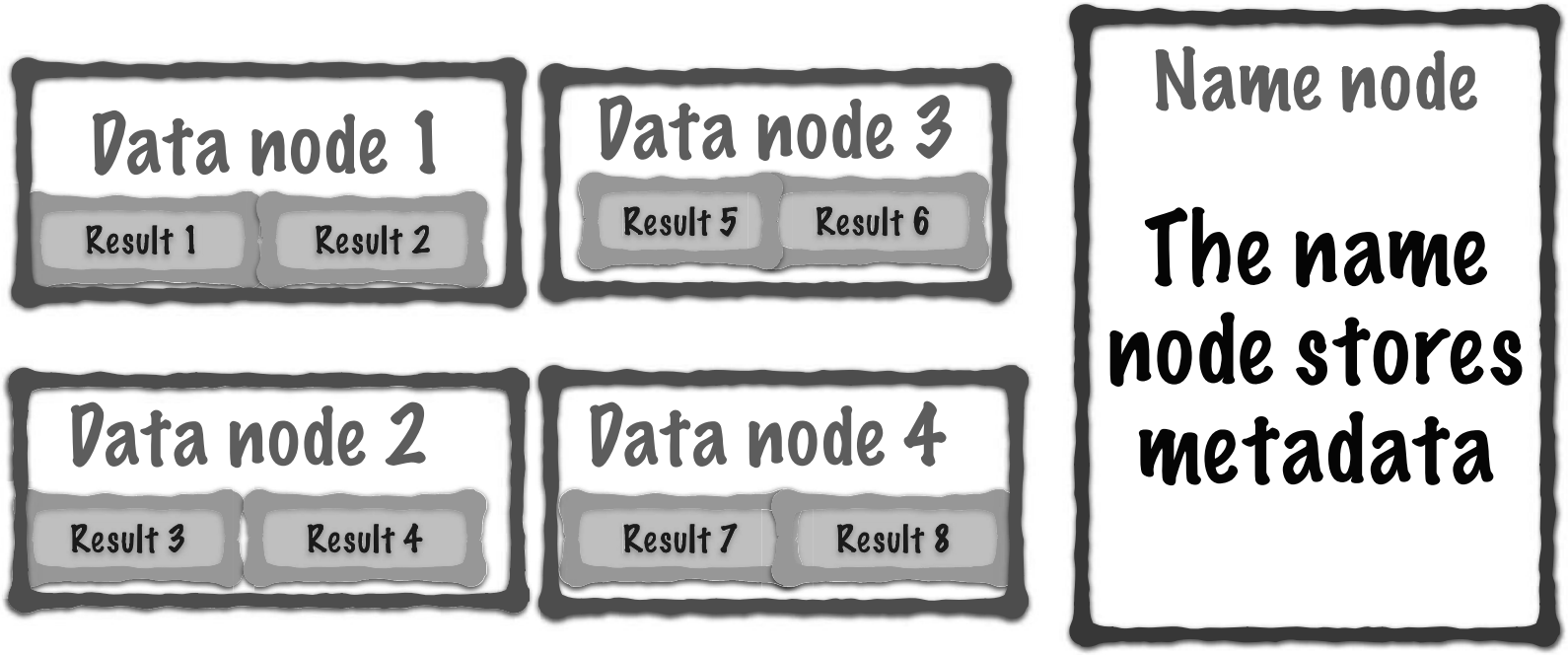
1. Process each block in the node it is stored in




## Map phase

# MapReduce

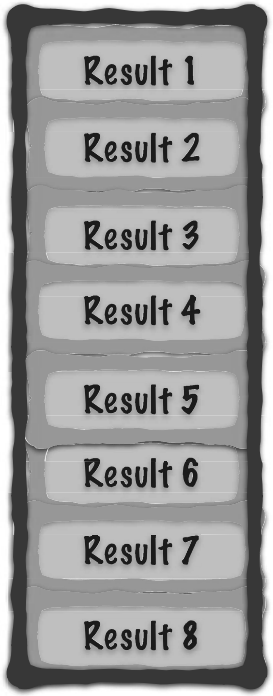
## 2. Take all the results to one node and combine them



# MapReduce

2. Take all the results to one node and combine them

## Reduce phase




Name node

The name node stores metadata

# MapReduce

**Any data processing task can be** expressed as a chain of map reduce operations

# MapReduce

The programmer just specifies the logic to be implemented the map and reduce phases

The rest is taken care of by Hadoop

# HADOOP

## MapReduce

A framework to process data across multiple servers

## HDFS

A file system to manage the storage of data

# Hadoop ecosystem


With Hadoop, you can

1. Store data in a cluster and
2. Process it