

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

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

BigTable

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

Storage

MapReduce

Processing data

BigTable

Database management

Hadoop ecosystem

**An ecosystem of Open source softwares
based on these architectures**

3 seminal papers

Hadoop ecosystem

Google File System
Storage

HDFS



MapReduce
Processing data

Hadoop MapReduce



BigTable
Database management

HBase



Hadoop ecosystem



Hadoop

HADOOP

**is a distributed computing framework
developed and maintained by
THE APACHE SOFTWARE FOUNDATION
written in Java**

Hadoop

HDFS

MapReduce

A file system to
manage the
storage of data

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

Name node

One of the nodes acts
as the master node

This node
manages the
overall file system

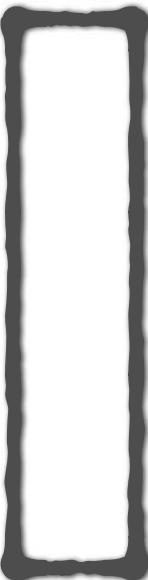
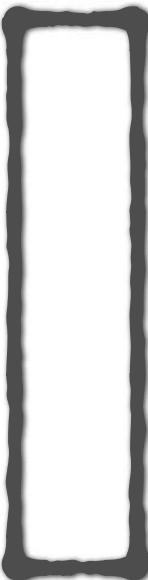
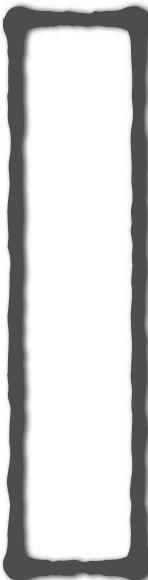
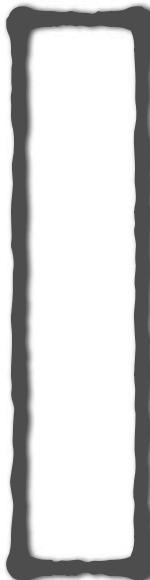
HDFS

The name node stores

1. The directory structure

2. Metadata for all the files

Name node



HDFS

Other nodes are called data nodes

The data is physically stored on these nodes

Name node

Data node 1

Data node 2

Data node 3

Data node 4

HDFS

Here is a large text file

Text Up Previous 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.

Figure 4.5: An example of distributed indexing with MapReduce. Adapted from Dean and Ghemawat (2004).
Introducing parts [width=11.5cm!]{http://mapreduce.info/images/part1.pdf} In general, MapReduce breaks a large computing problem into smaller parts by recasting it in terms of manipulation of key-value pairs. In distributed indexing, the mapping from terms to terms is also distributed. For indexing, a key-value pair has the form (termID, docID). A simple solution is to maintain a (perhaps precomputed) mapping for frequent terms that is copied to all nodes and use terms directly (instead of termIDs) for infrequent terms. We do not address this problem here and assume that all nodes have a consistent term \$NAME across term 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 BSB1 and SPMMI, 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 `libpq-{language}.lbox-p- $\#$ -segment`) in Figure 4.5.

For the reduce phase, we want all values for 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 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 \$1-\$3. Note that these key ranges for S response to contiguous terms (or terms). 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 $|S|$ segment files, where S is the number of parsers. For instance, Figure 4.5 shows segment files for 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 their reduce phase.

Let's see how
this file is
stored in HDFS

HDFS

first the file is
broken into
blocks of size
128 MB

Text: Dynamic indexing Up: Index construction Previous: Single-pass in-memory indexing Contents Index

Block 1

Distributed indexing

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 as a result of the construction process is a distributed index that is partitioned across several machines - either according to document ID or according to document content.

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). 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 machines or nodes that are built from standard hardware. Although hundreds or thousands of machines are available in such clusters, individual machines can fail at any time.

The map and reduce phases of MapReduce split up the computation into chunks that standard machines can process in a short time. The various steps of MapReduce are shown in Figure 4.5 and in Figure 4.6. The first step splits the input data, in our case a collection of web pages, into 128MB chunks where the size of the chunks 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 is arranged by the overhead of managing many small chunks). The second step consists of machine instances processing one segment at a time. One machine dies or becomes a laggard due to hardware problems, the segment it was working on is simply reassigned to another machine.

Figure 4.5: An example of distributed indexing with MapReduce (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.

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

use for intermediate tasks, like segment files (shown as `\texttt{VLOGX1a-VLOGX1f}, \texttt{VLOGX1g-VLOGX1j}, \texttt{VLOGX1k-VLOGX1l}) in Figure 4.3.`

For the reduce phase, we want all values for a given key to be close together, so that they can be read and processed quickly. This is achieved by partitioning the keys into \$s\$ term partitions, and by having the parsers write key-value pairs for each term partition into a separate segment file. In Figure 4.5, the term partitioning corresponds to first letter: a-f, g-h, q-z, and s-t. (We chose these random for ease of presentation.) The term partitions are

term partition. Each term partition thus corresponds to s segments files, where s is the number of parsers. For instance, Figure 4.5 shows three a-f segment files of the a-f partition, corresponding to three parsers shown in the figure.

Collecting all values (here: docIDs) for a given key (here `\texttt{C}`) is the task of the inverters in the reduce phase. The `\texttt{C}` shows three a-f segment files of the a-f partition, corresponding to three parsers shown in the figure.

Block 2

Block 3

Block 4

Block 5

Block 6

Block 7

Block 8

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

Text: 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 as a result of the distributed nature of the document. In this section, we describe distributed indexing for a partitioned index. Most large search engines prefer a document-partitioned index (which can be easily generated from a term-partitioned index).

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 machines or nodes that are built from standard hardware. Although hundreds or thousands of machines are available in such clusters, individual machines can fail at any time.

Block 2

The map and reduce phases of MapReduce split up the computation into chunks that standard machines can process in a short time. The various steps of MapReduce are shown in Figure 4.5 and in Figure 4.6. First, the input data, in our case a collection of web pages, is split into splits where the size of each 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 is arranged by the user). Second, instances processing one split are assigned one machine. One machine processes one split. If a machine dies or becomes a laggard due to hardware problems, the split it was working on is simply reassigned to another machine.

Figure 4.5: An example of distributed indexing with MapReduce (from Gan and Ghemawat, (2004)).

Figure 4.6: An example of distributed indexing with MapReduce (width=11.5cm)[start/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 terms directly (instead of term IDs) for infrequent terms. We do not address this problem here and assume that all nodes share a consistent term S .

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 faced in intermediate tasks, the segment files (shown as `\texttt{VLOGX1-VLOGX2}, \texttt{VLOGX3-VLOGX4}, \texttt{VLOGX5-VLOGX6}` in Figure 4.3).

For the reduce phase, we want all values for a given key b to be close together, so that they can be read and processed quickly. This is achieved by partitioning the keys into S term partitions, and by having the parsers write key-value pairs for each term partition into a separate segment file. In Figure 4.5, the term partitioning corresponds to first letter: a-f, g-h, q-z, and S=3S. (We chose these random for ease of presentation.) The term partitions are term partition. Each term partition thus corresponds to S segments files, where S is the number of parsers. For instance, Figure 4.5 shows three a-f segment files of the a-f partition, corresponding to three parsers shown in the figure.

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

HDFS

Text: Dynamic indexing Up: Index construction Previous: Single-pass in-memory indexing Contents Index

Block 1 Distributed indexing

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 as a result of the nature of the document. In this section, we describe distributed indexing partitioned across several machines – either according to term or document.

Most large search engines prefer a document-partitioned index (which can be easily generated from a term-partitioned index).

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 machines or nodes that are built from standard hardware. Although hundreds or thousands of machines are available in such clusters, individual machines can fail at any time

Block 2
These blocks are then stored across the data nodes

The map and reduce phases of MapReduce split up the computation into chunks that standard machines can process in a short time. The various steps of MapReduce are shown in Figure 4.5 and in Figure 4.6. First, the input data, in our case a collection of web pages, is split into S splits where the size of each 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 is arranged by the user). Each machine processes one split. One machine processes one segment, one machine processes one term, and so on.

Figure 4.5: An example of distributed indexing with MapReduce (width=11.5cm)[start/mapreduce2.eps].

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

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 faced in intermediate tasks, like segment files (shown as `\texttt{VLOGX1a-VLOGX1f}, \texttt{VLOGX1g-VLOGX1j}, \texttt{VLOGX1k-VLOGX1l}) in Figure 4.3.`

For the reduce phase, we want all values for a given key to be close together, so that they can be read and processed quickly. This is achieved by partitioning the keys into S term partitions and assigning the parsers for each term partition to a separate segment file. In Figure 4.5, the term partition $a-f$ corresponds to first letter: $a-f$, $g-h$, $i-j$, and $k-l$. (We chose these random for ease of explanation. In general, key names need not correspond to continuous terms or segments.) The term partitions are term partition. Each term partition thus corresponds to s segments files, where s is the number of parsers. For instance, Figure 4.5 shows three $a-f$ segment files of the $a-f$ partition, corresponding to three parsers shown in the figure.

Collecting all values (here: docIDs) for a given key (here `\texttt{C}`) is the task of the inverters in the reduce phase. The **Block 8** part of the figure shows the inverters for the $a-f$ partition. The inverters collect all values for a given key and then store them in memory. The inverters then send the collected values to the reduce phase. The **Block 9** part of the figure shows the reduce phase. The reduce phase takes the collected values and performs the reduce operation. The reduce operation is typically a simple aggregation operation, such as summing up the values for a given key. The reduce phase then outputs the final results, which are then stored in the HDFS system.

HDFS

Name node

The name
node stores
metadata

Data node 1

Block 1

Block 2

Data node 2

Block 3

Block 4

Data node 3

Block 5

Block 6

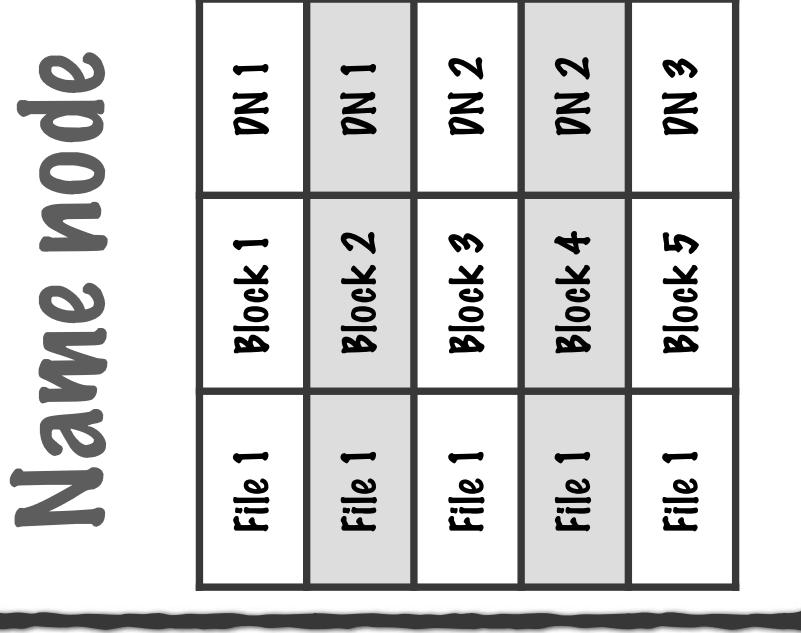
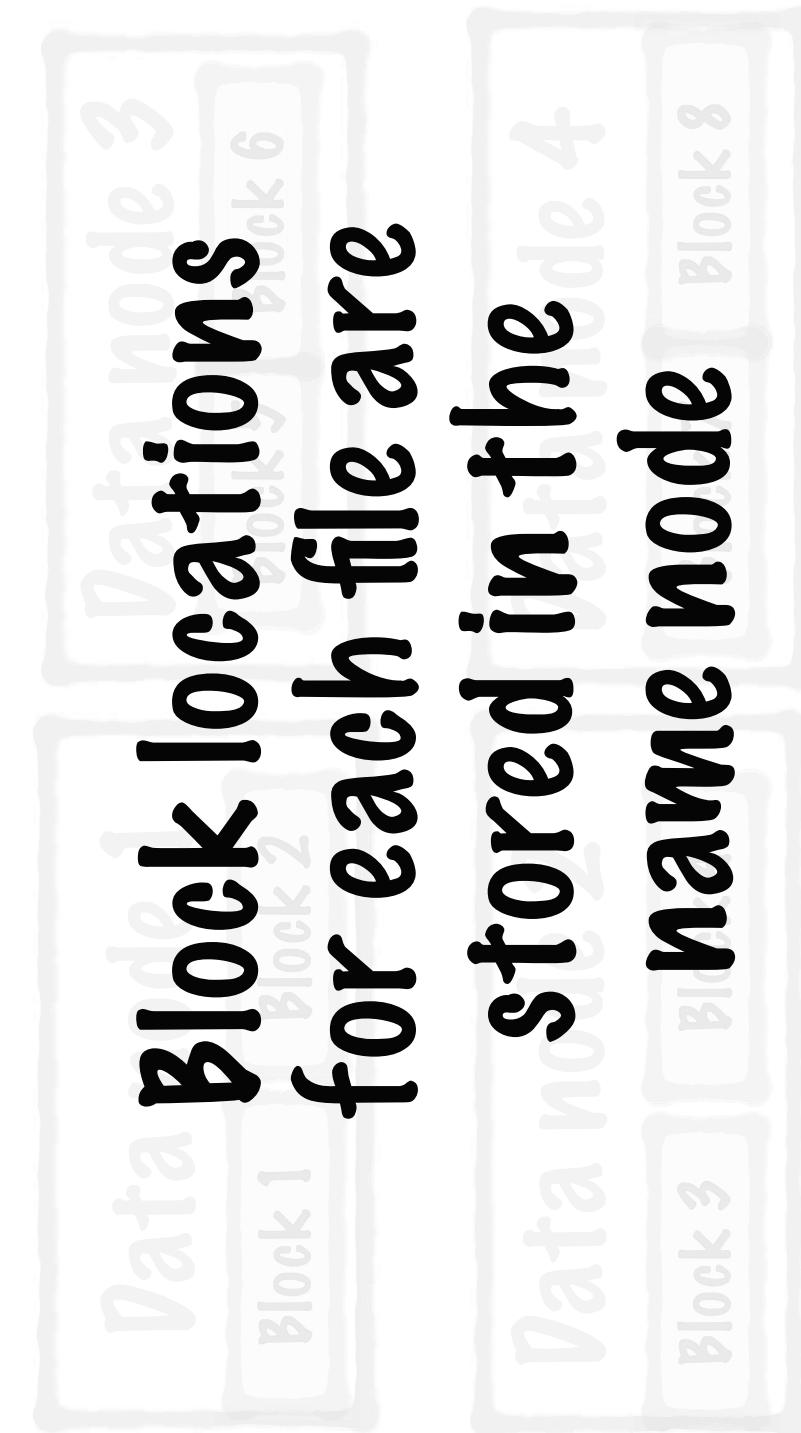
Data node 4

Block 7

Block 8

HDFS

Block locations for each file are stored in the name node



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

Block 7 Block 8

Block 3 Block 4

Block 3 Block 4

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

Data node 3

Block 6

Block 5

Data node 1

Block 2

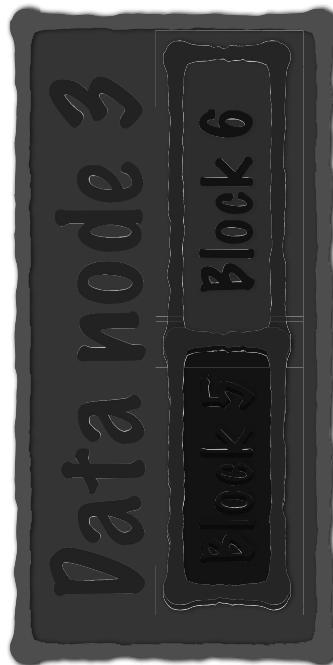
Block 1

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

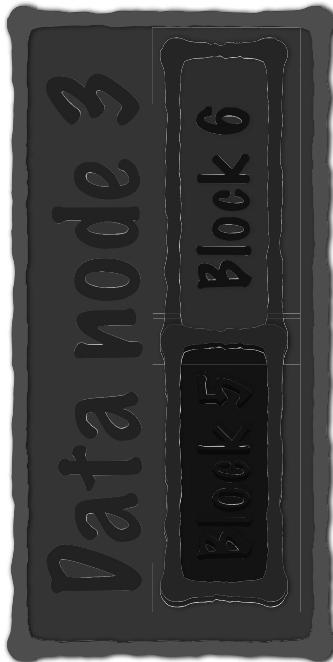


Or one of the data
nodes crashes?

HDFS

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

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

Data node 3

Block 6

Data node 1

Block 1

Data node 2

Block 4

Data node 4

Block 3

Data node 7

Block 8

HDFS

Name node

Data node 1

Block 1 Block 2

Data node 3

Block 5 Block 6

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

Data node 2

Block 3 Block 4

Block 1 Block 2

HDFS

Name node

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

Data node 3

Block 5 Block 6

Data node 1

Block 1 Block 2

The replica locations
are also stored in the
name node

Hadoop

HDFS

A file system to
manage the
storage of data

MapReduce

A framework to
process data across
multiple servers

Hadoop

works

MapReduce

A file system to
manage the
storage of data

A framework to
process data across
multiple servers

MapReduce

**MapReduce is a way
to parallelize a data
processing task**

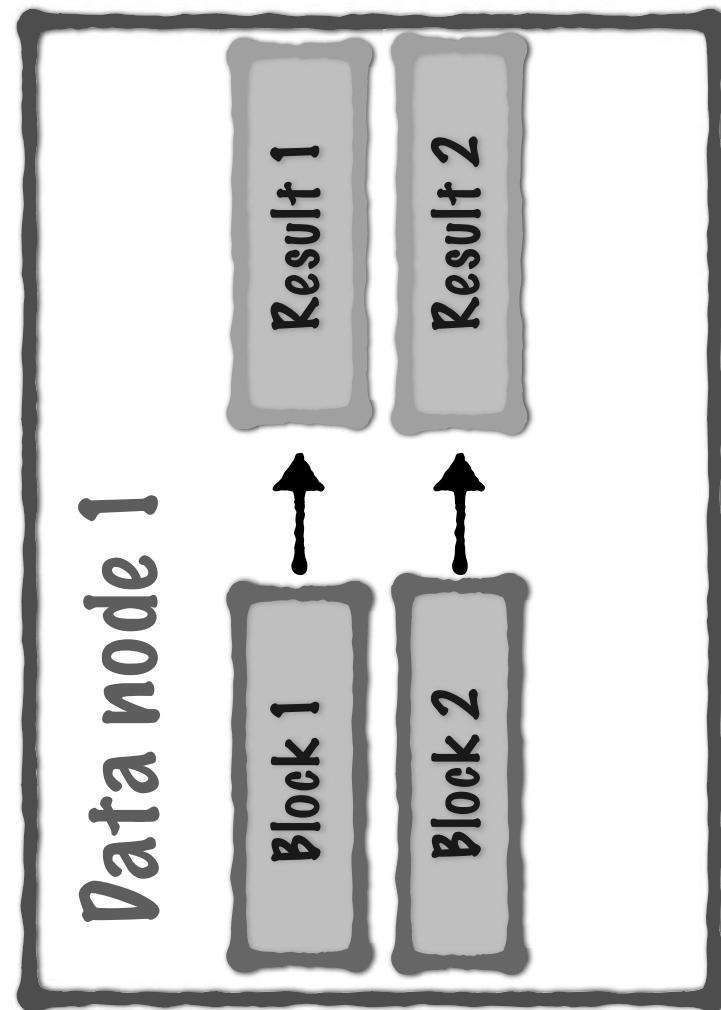
MapReduce

MapReduce tasks
have 2 phases

MapReduce

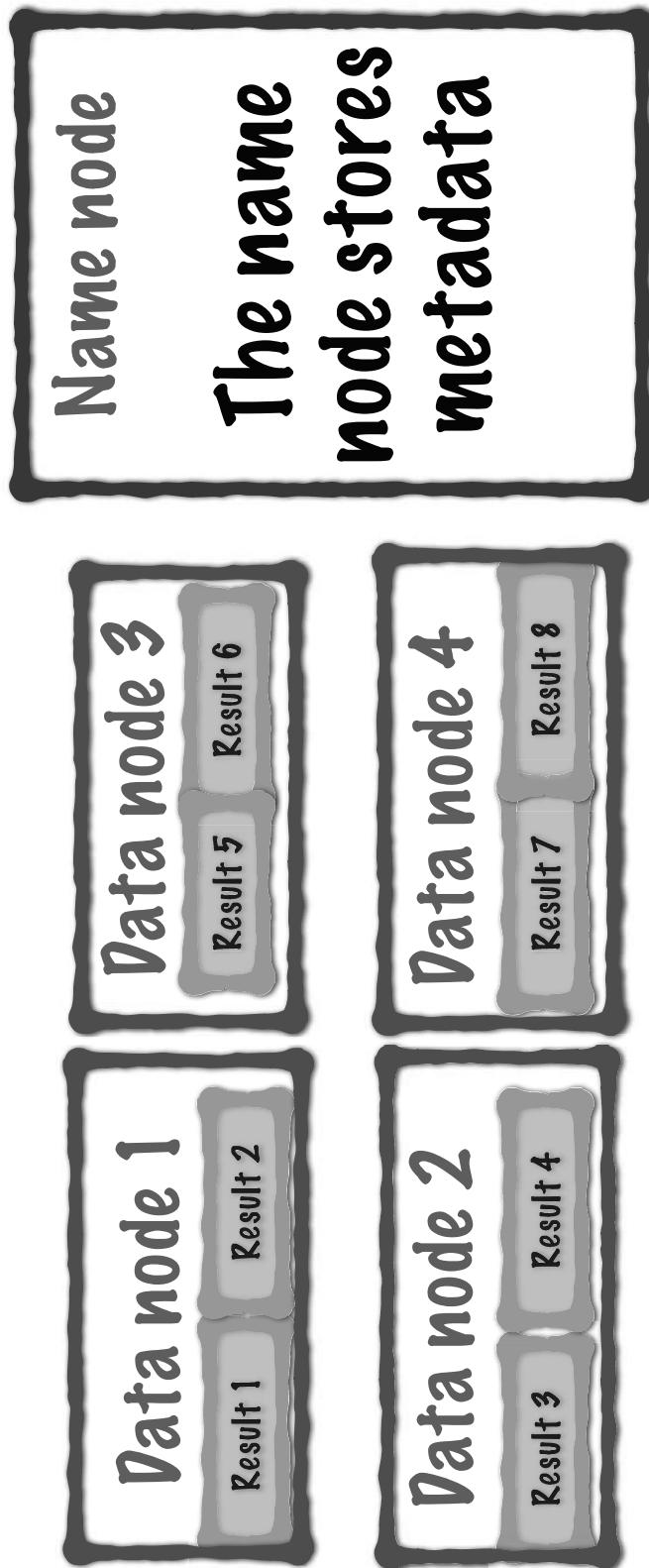
1. Process each block in the node it is stored in

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

Name node

The name
node stores
metadata

Result 1

Result 2

Result 3

Result 4

Result 5

Result 6

Result 7

Result 8

Reduce phase

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 for the map and reduce phases

The rest is taken care of by Hadoop

Hadoop

works

MapReduce

A file system to
manage the
storage of data

A framework to
process data across
multiple servers

Hadoop ecosystem

With Hadoop, you can

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

HDFS

Hadoop MapReduce

HBase