

Many technological
innovations are inspired
by nature

Take ants for example



Take ants for example



Ants are fascinating
in many ways

Individually,
each ant is
seemingly
inconsequential



Collectively, they work together to accomplish complicated things

They rarely come alone. They march single file through minuscule cracks around windows or under doors, looking for crumbs, water or a warm place to make a new home. Often you'll see them trooping up your walls or across your counter, organized and on a mission. You have an ant invasion.

MARY JO DILONARDO, "What kind of ants are in my house?", *Mother Nature Network*, August 10, 2015

Together, ants behave
like a single entity

In pursuit of a
common goal

a single entity
common goal

Take one ant down;
Another comes up to
take it's place!

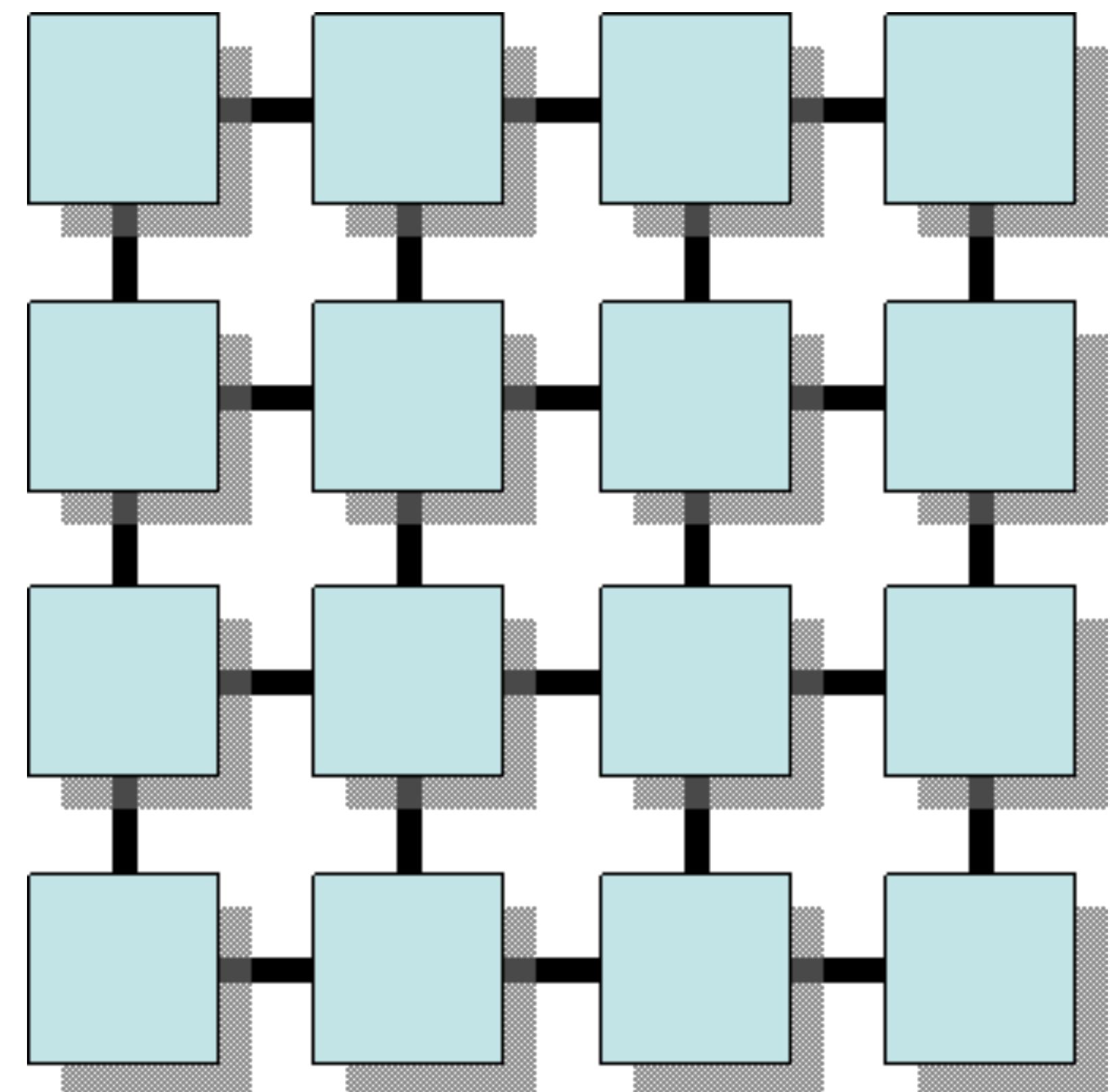
Distributed computing

is the idea of putting
many small and cheap
computers together

Distributed computing

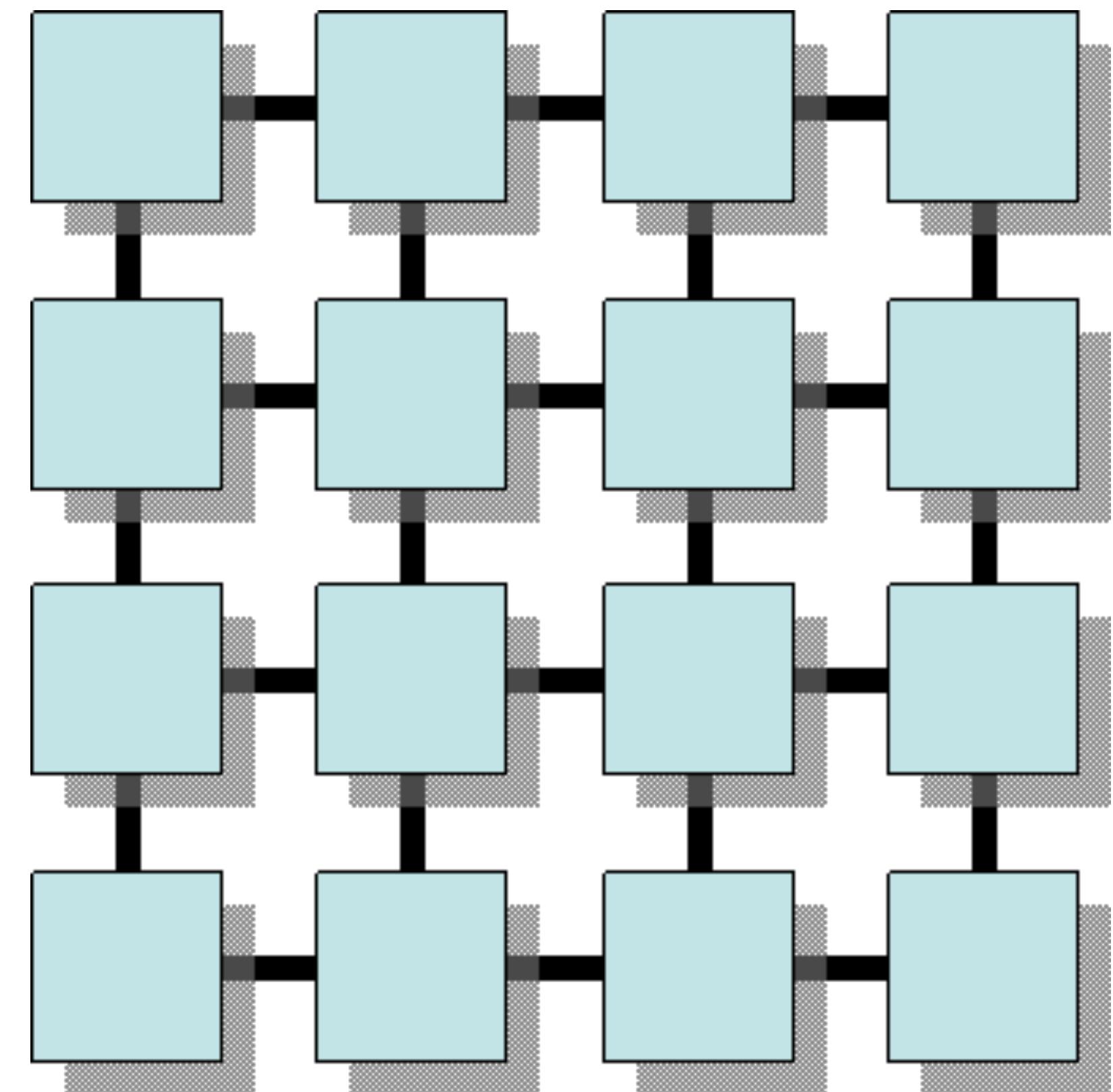
is the idea of putting many small
and cheap computers together

to accomplish
really complex
tasks



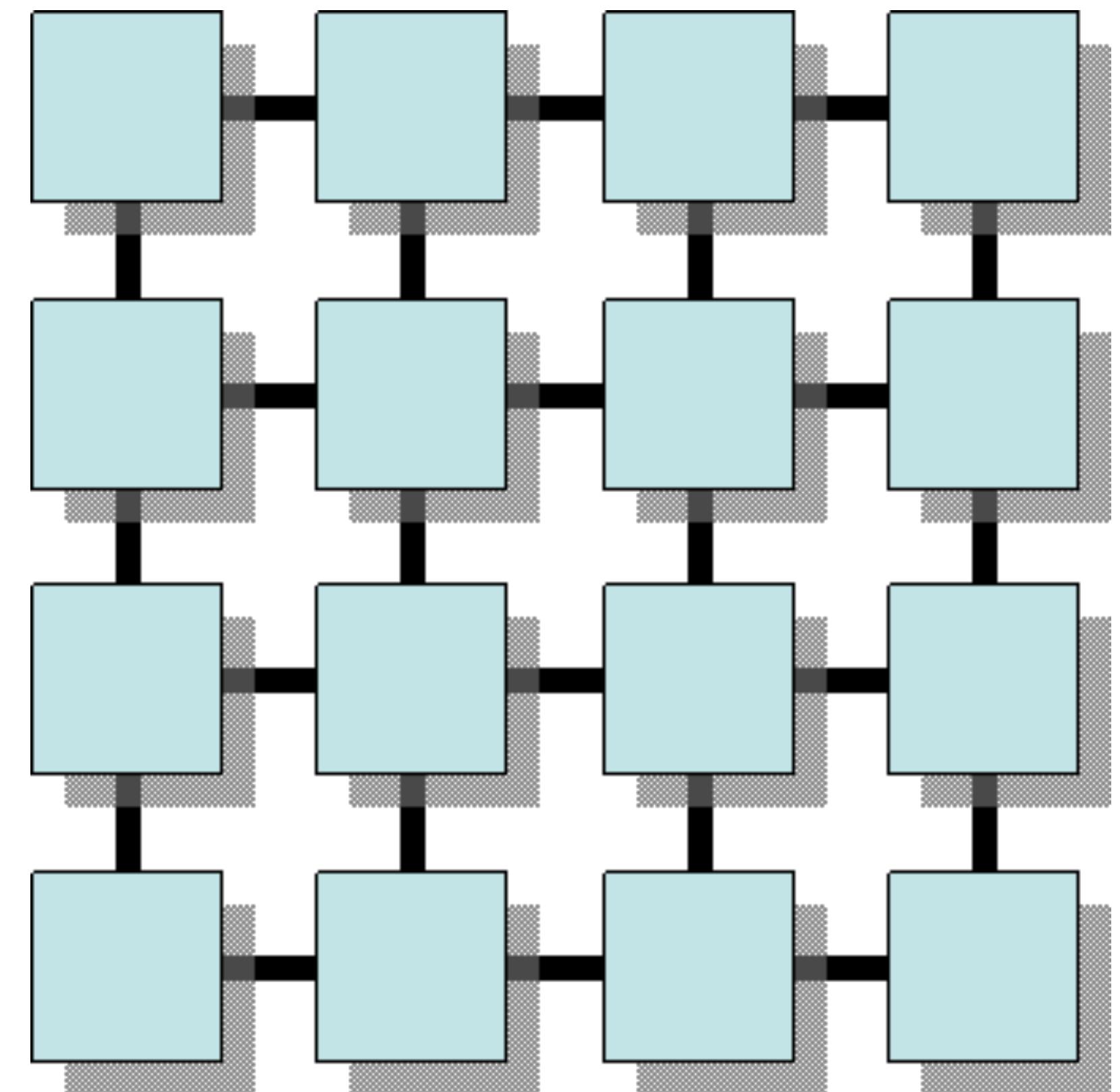
Distributed computing

Each individual computer is called a node



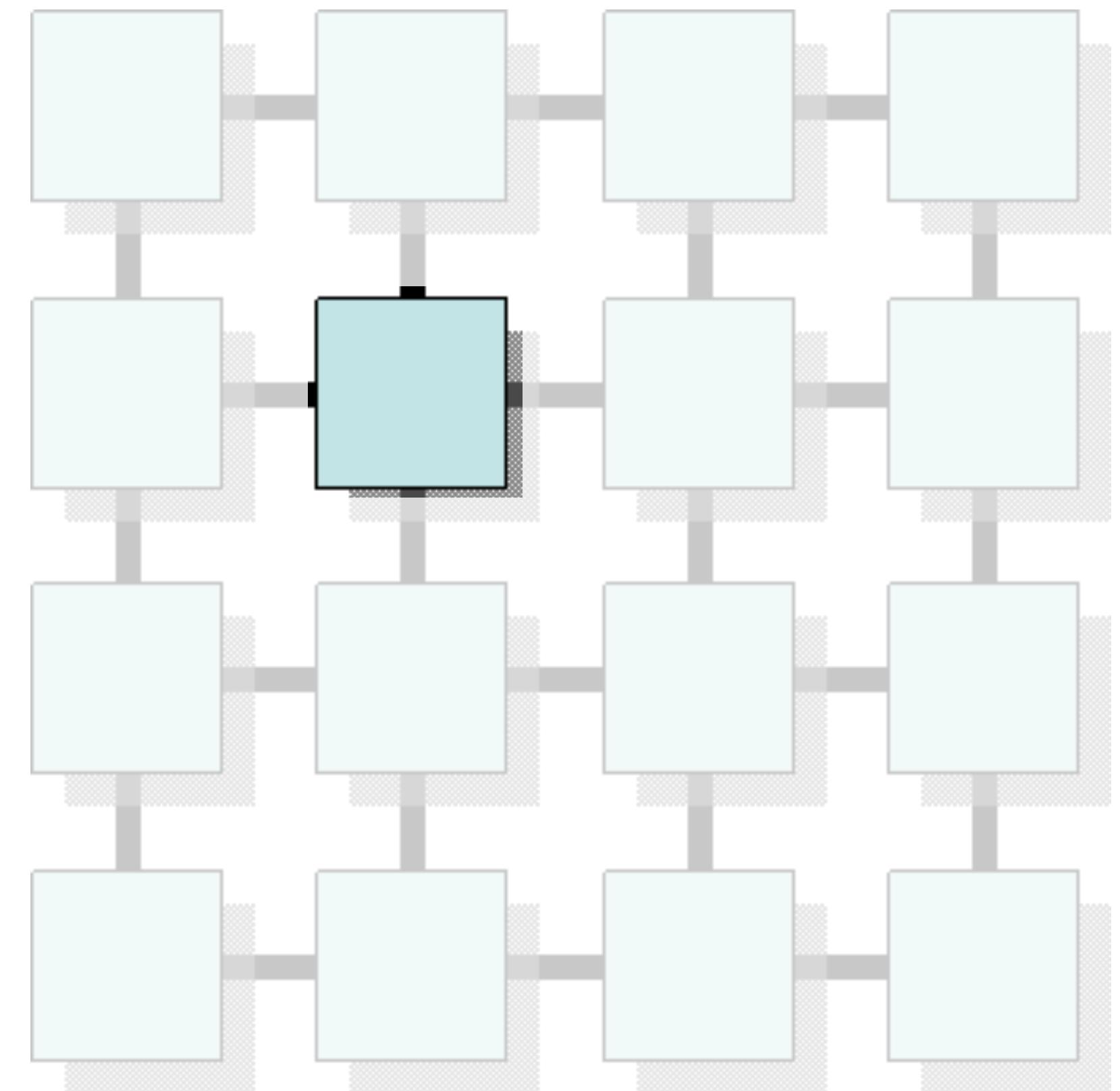
Distributed computing

Together, all the nodes form a cluster



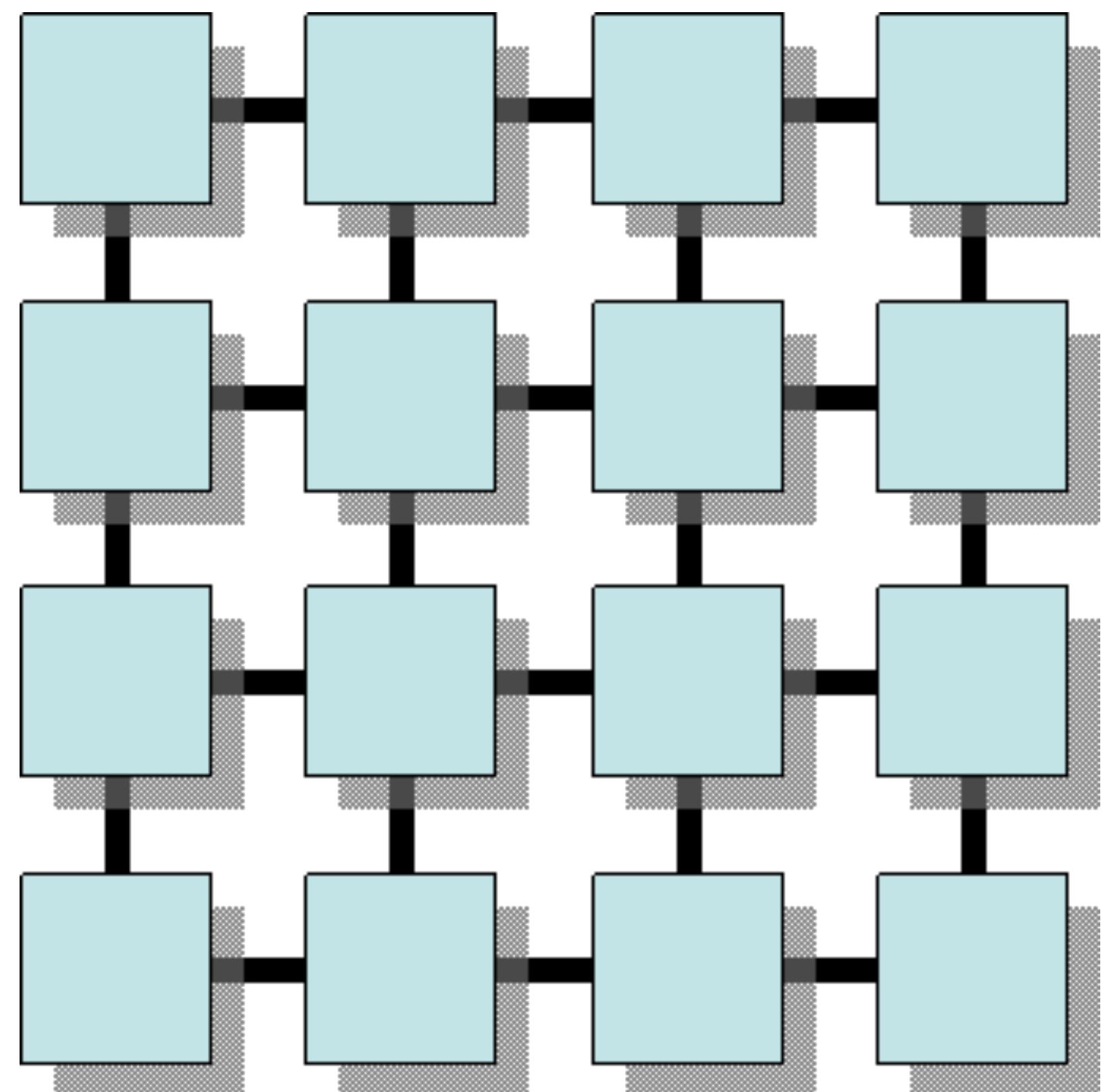
Distributed computing

Like ants,
Each individual
node is pretty
inconsequential



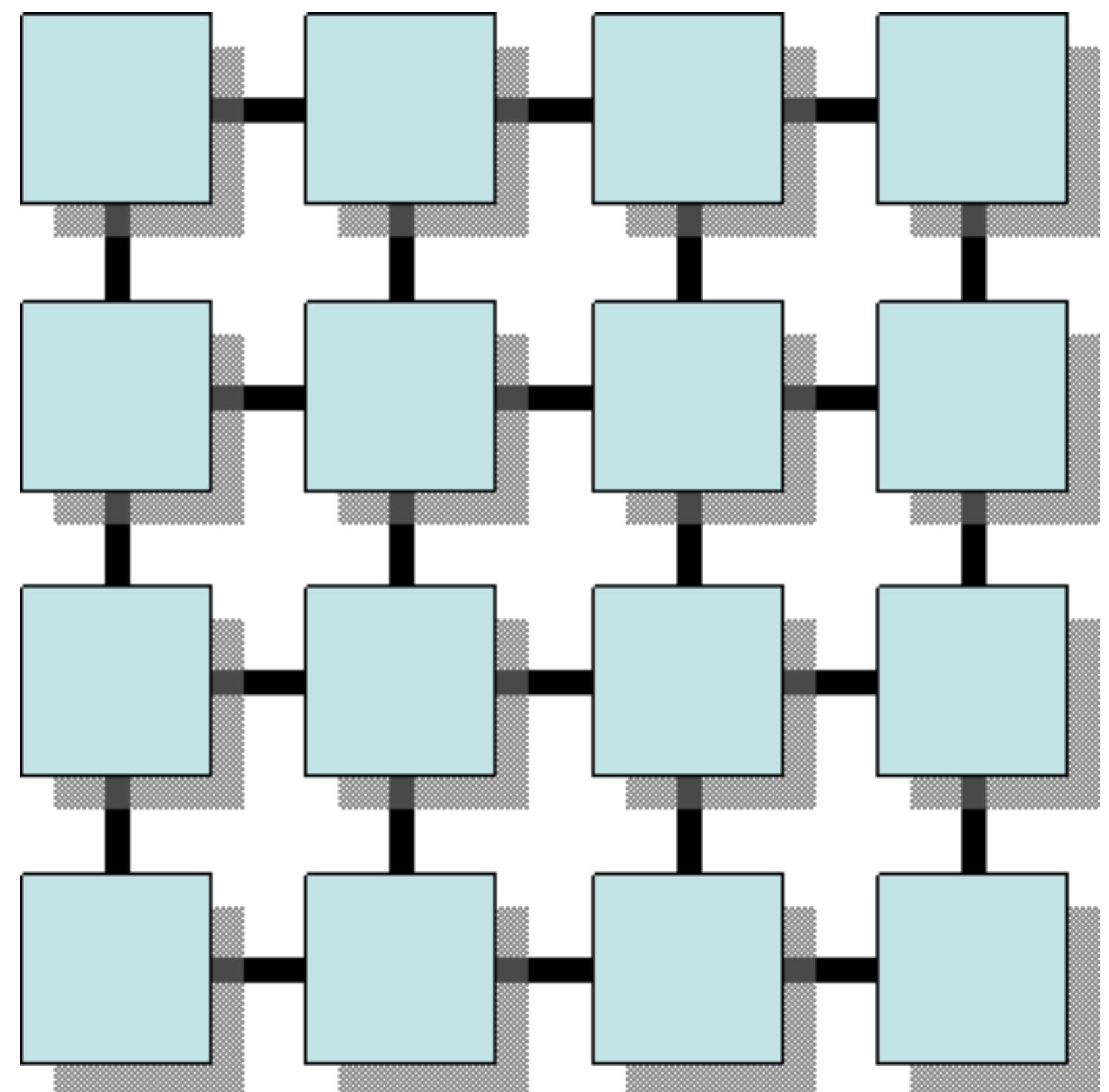
Distributed computing

Like ants,
Together these
nodes act like a
single entity with
a common goal



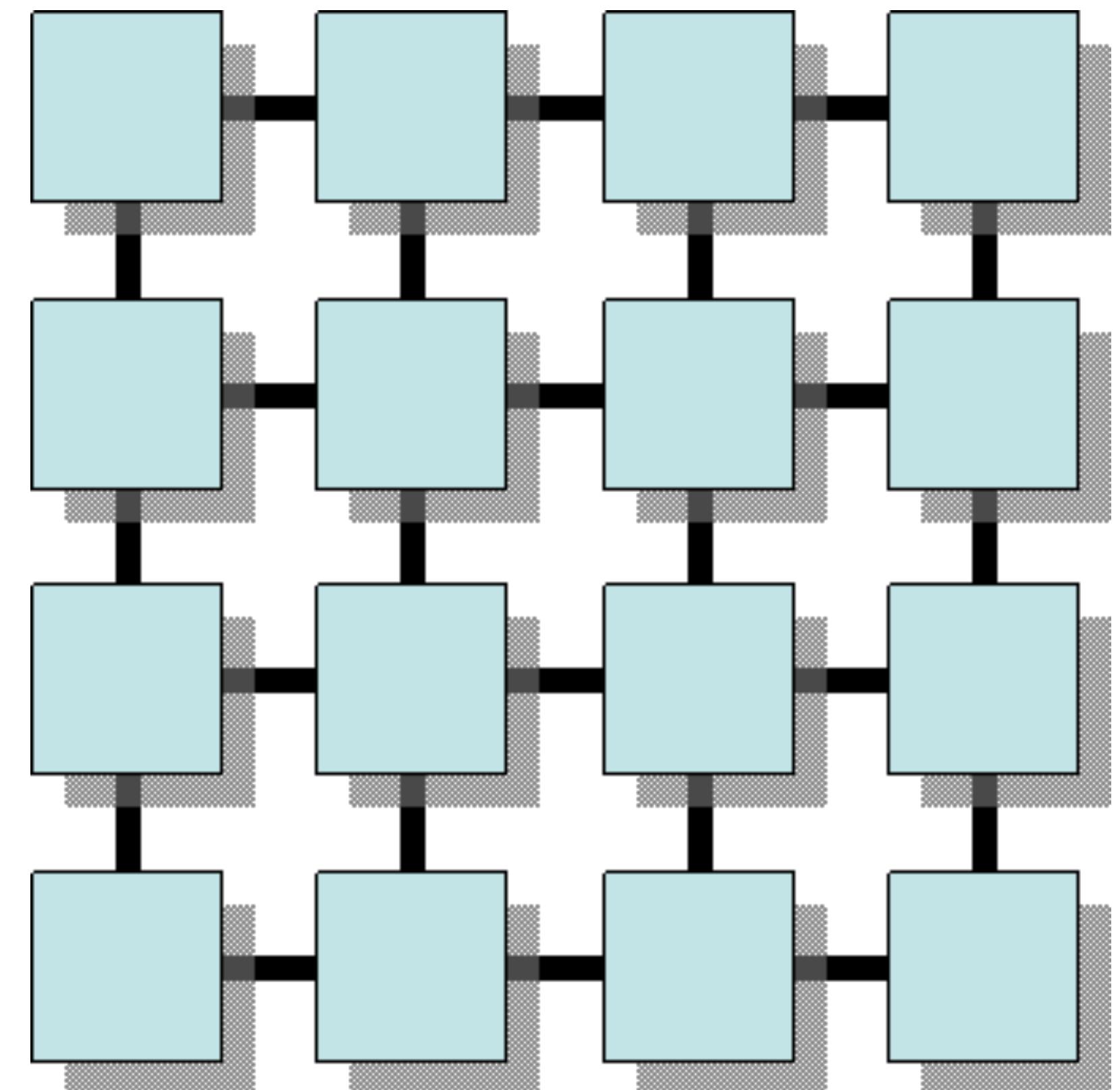
Distributed computing

Why is this so cool?



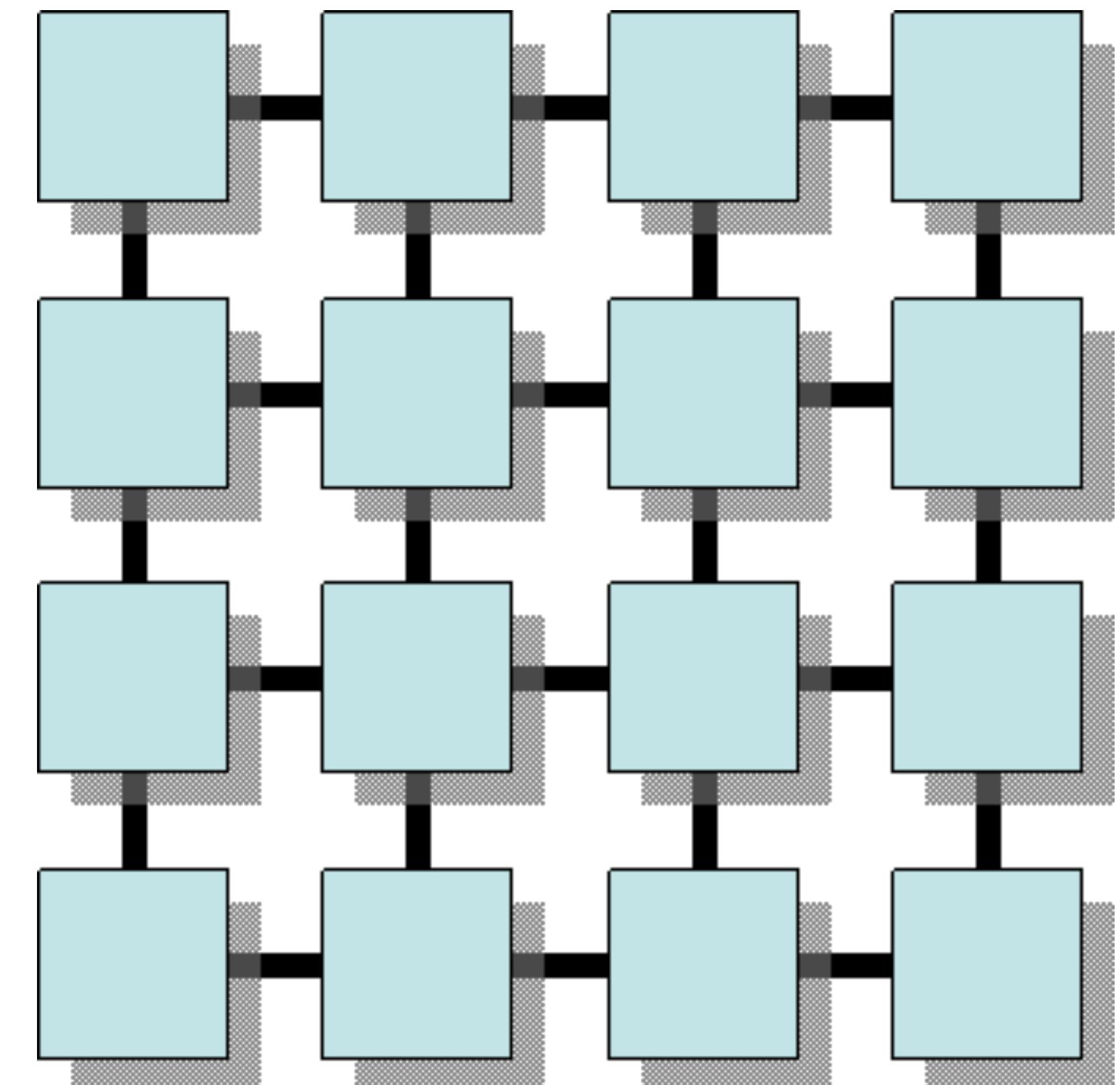
Distributed computing

The performance
of this system
scales linearly



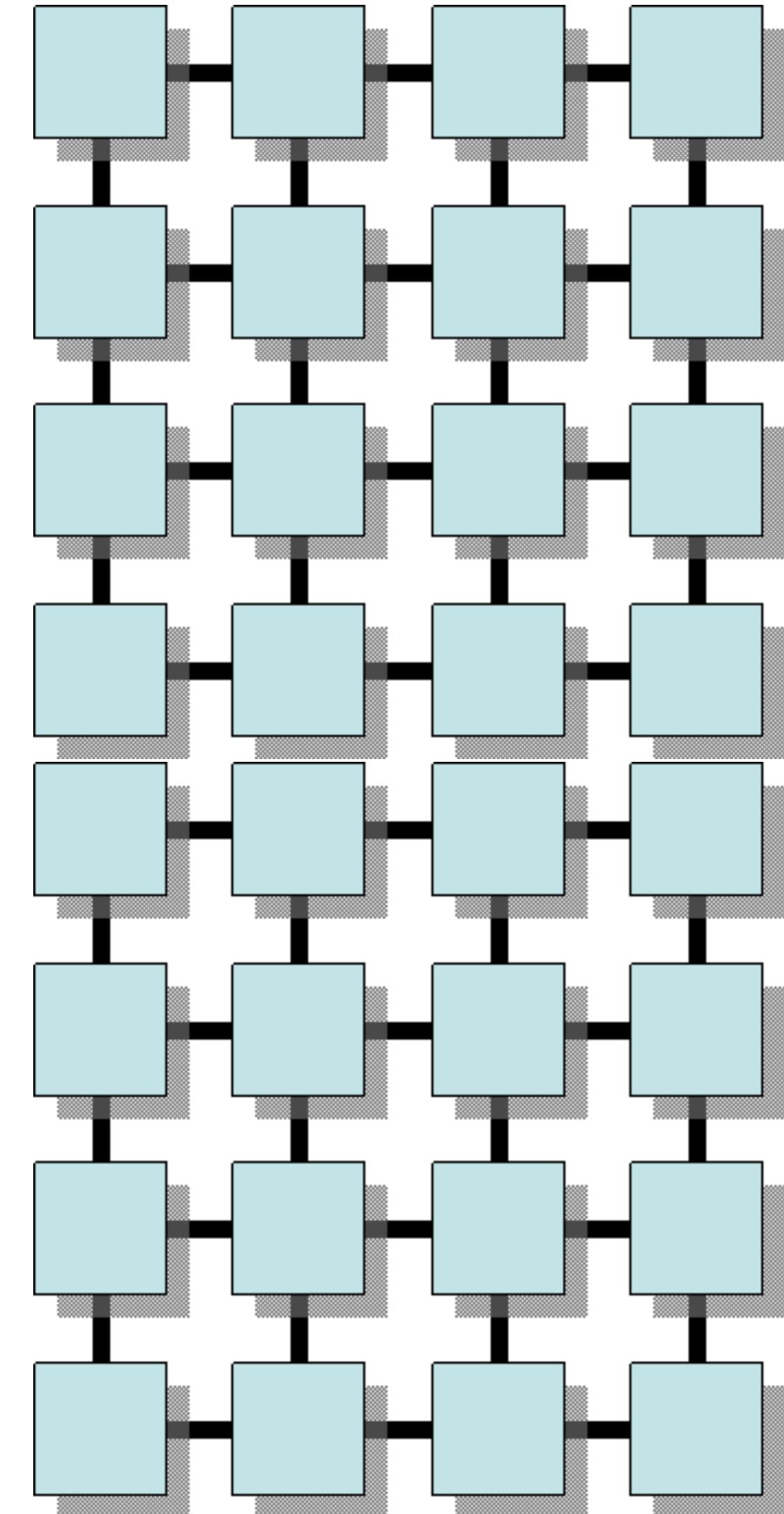
Distributed computing

To double the performance, just
double the number of nodes



Distributed computing

To double the performance, just
double the number of nodes



Distributed computing

This is not true
of individual
computers

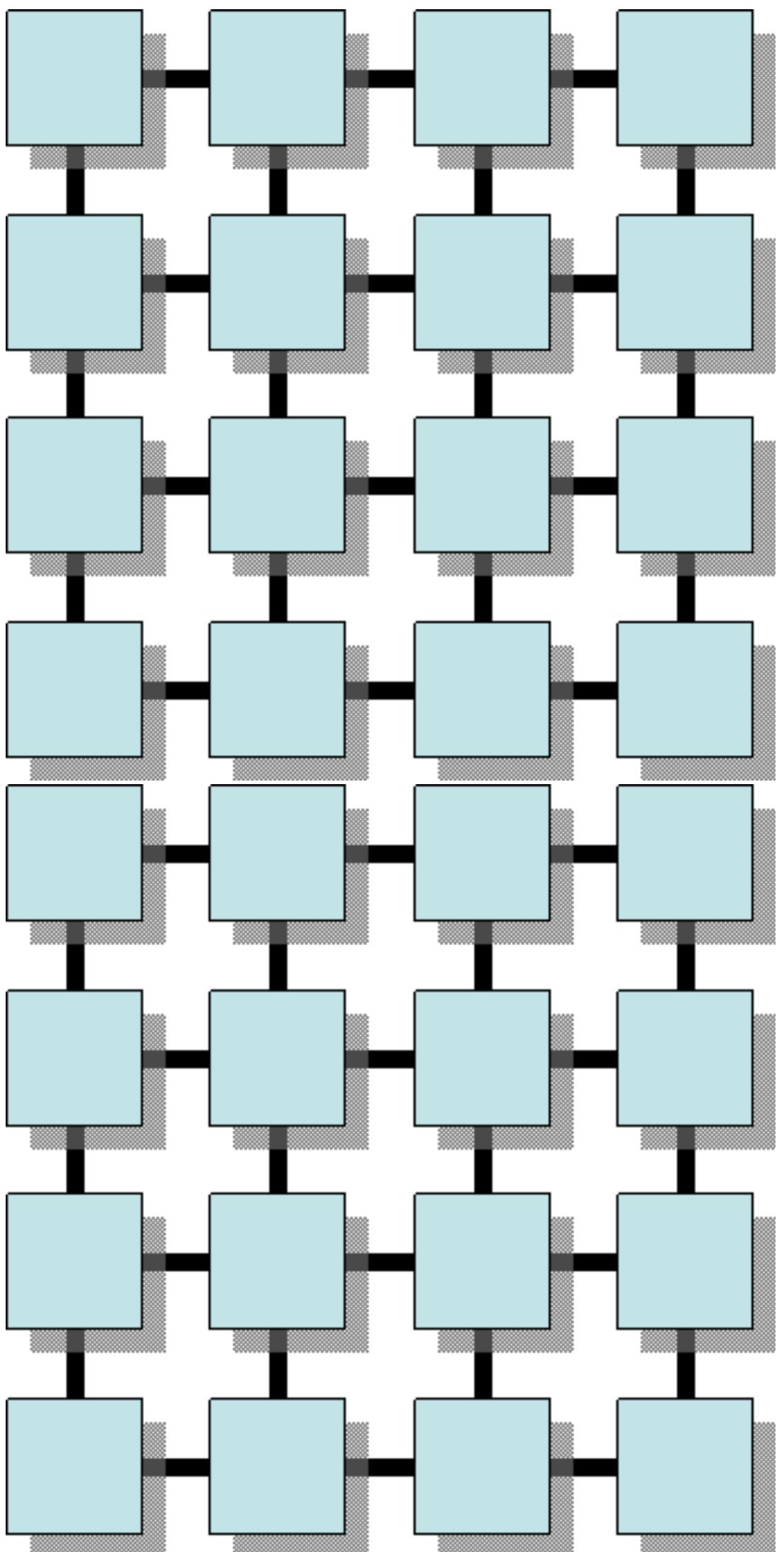


Distributed computing

A computer that's
twice as expensive,
will not necessarily
give you twice the
performance

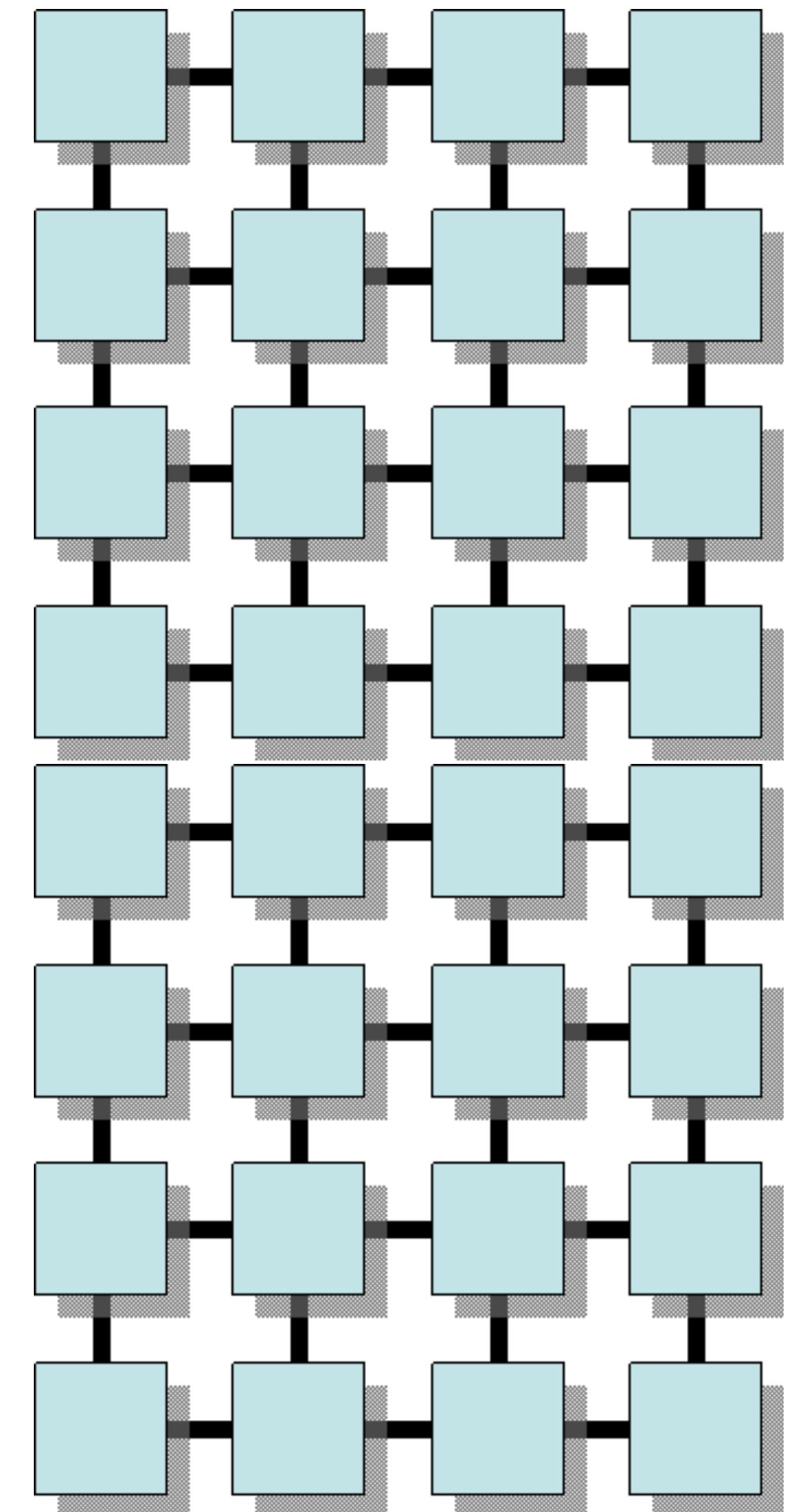


Distributed computing
can get **very complicated**



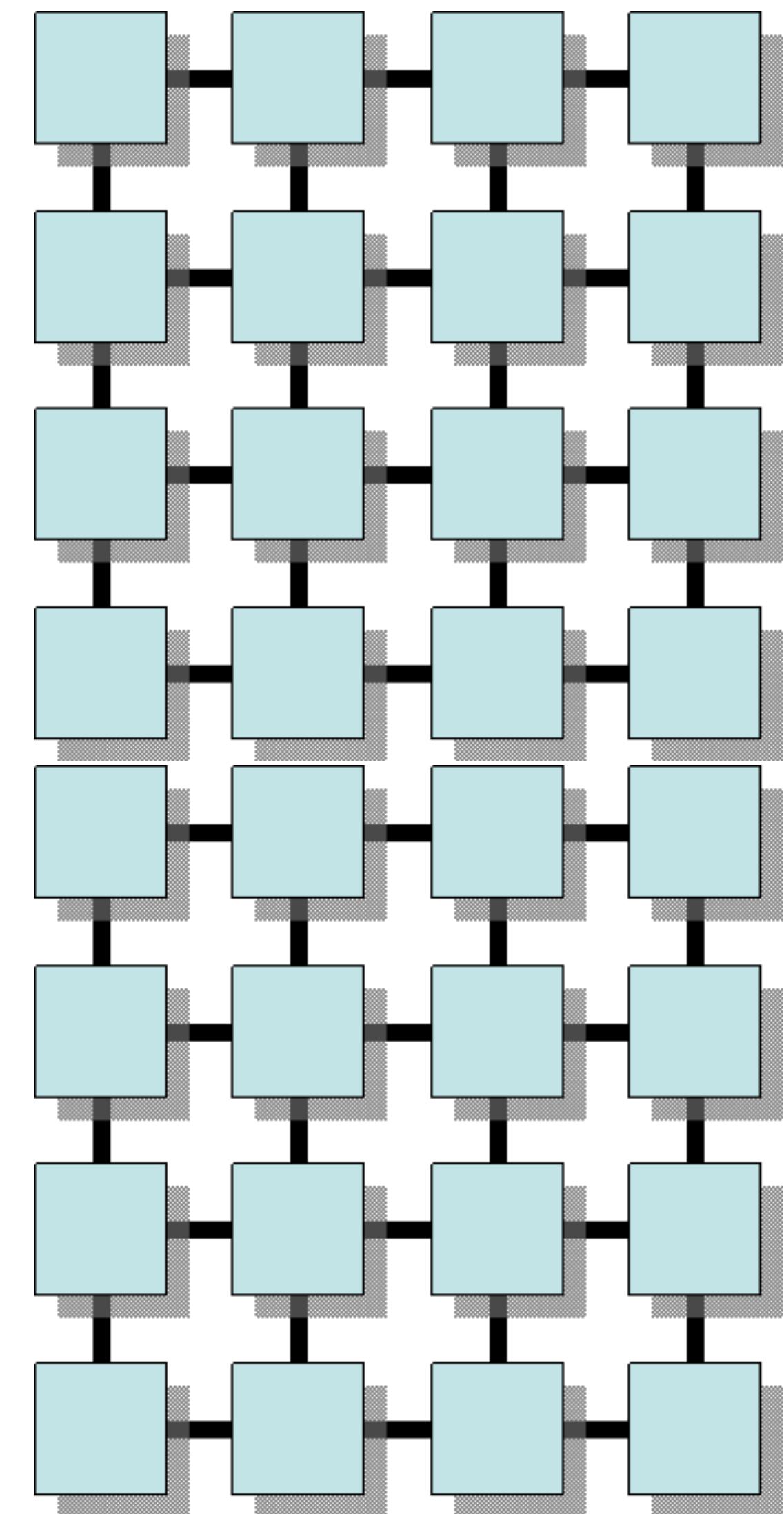
Distributed computing can get **very complicated**

1. You have to **manage resources and memory across multiple nodes**



Distributed computing can get **very complicated**
manage resources and memory

2. You have to **co-ordinate**
and schedule tasks



Distributed computing can get **very complicated**

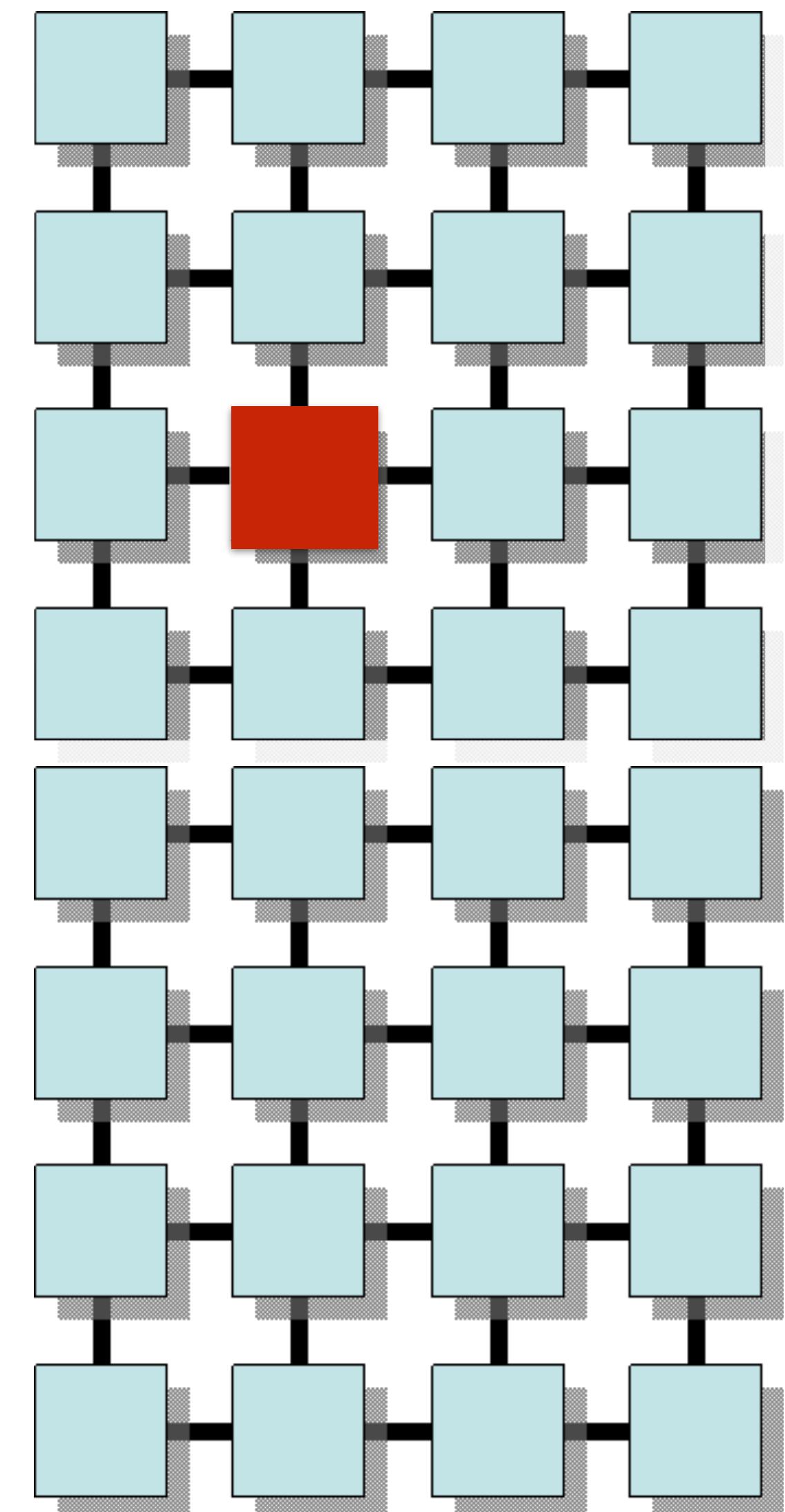
manage resources and memory

co-ordinate and schedule tasks

Fault Tolerance

3. If one node goes down, the system should not be affected

(Just like with ants)



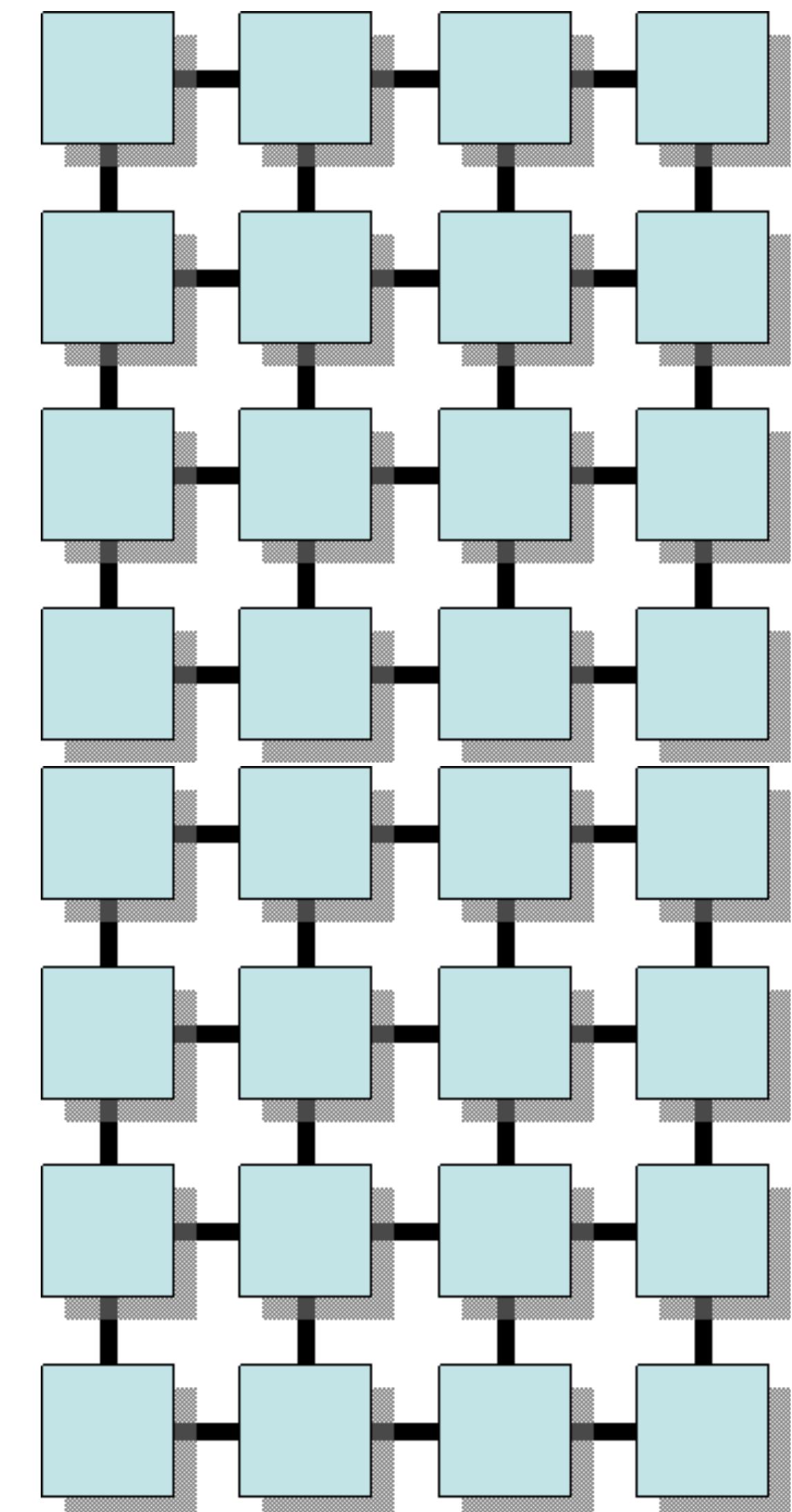
Distributed computing can get **very complicated**

manage resources and memory

co-ordinate and schedule tasks

Fault tolerance

Before the 2000s, all of these problems had to be taken care of by the programmer



Between 2003 and 2006

Google published 3 seminal papers

that completely changed the
world of distributed computing

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

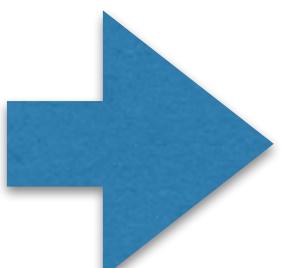
An ecosystem of Open source softwares
based on these architectures

3 seminal papers

Hadoop ecosystem

Google File System

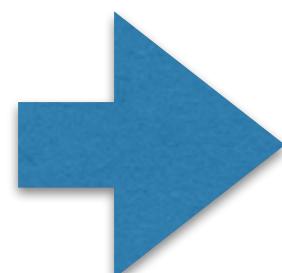
Storage



HDFS

MapReduce

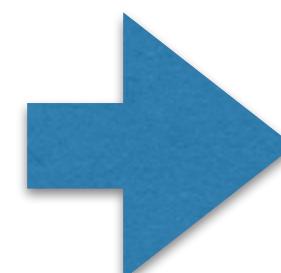
Processing data



Hadoop MapReduce

BigTable

Database management



HBase

Hadoop ecosystem

HDFS

Hadoop MapReduce

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

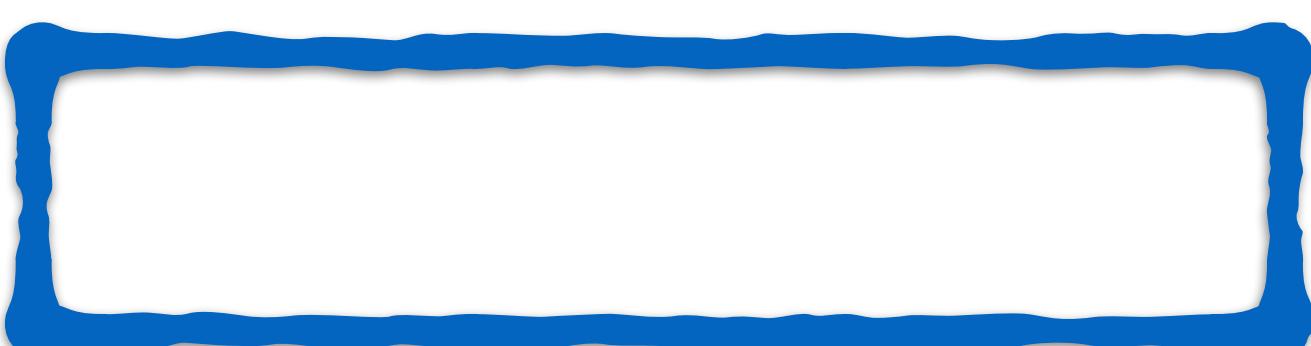
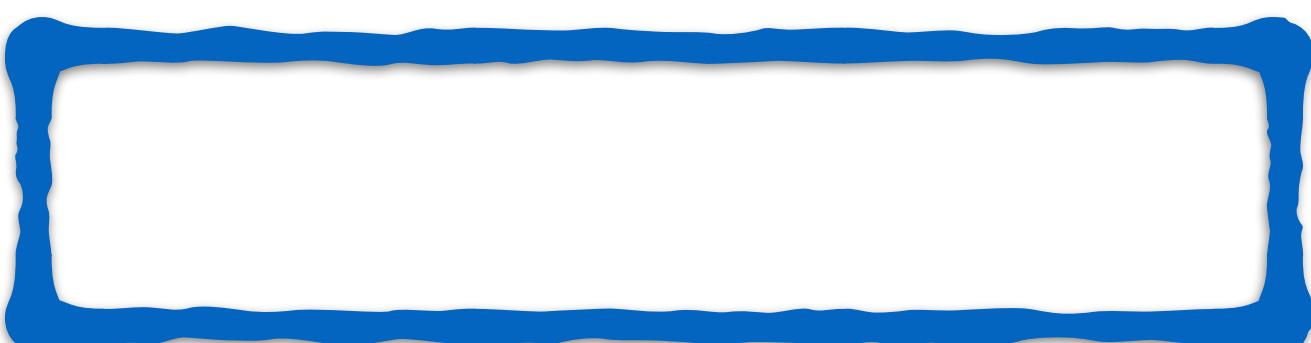
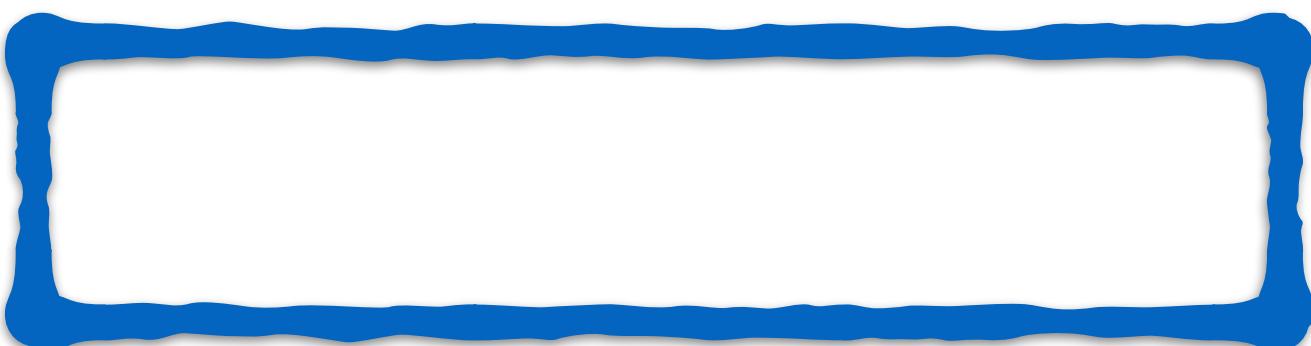
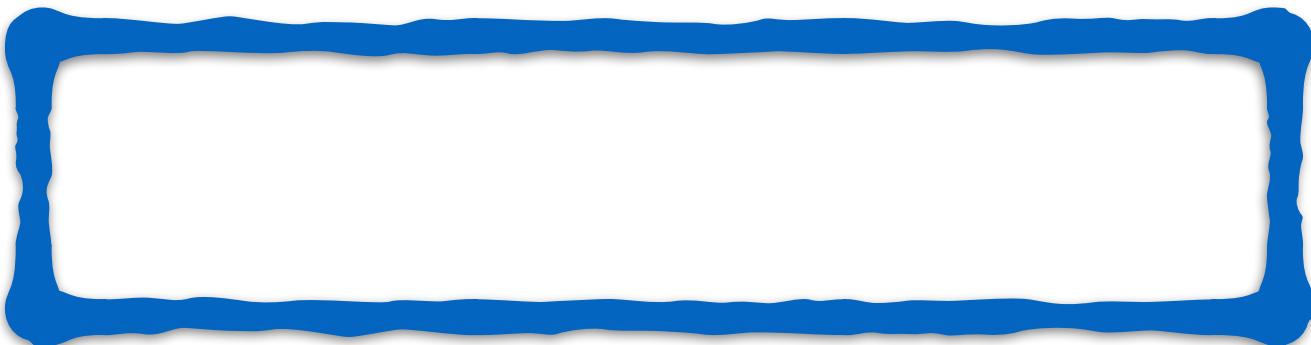
HDFS

The Hadoop Distributed File System

Hadoop uses this to store
data across multiple disks

HDFS

Name node



One of the nodes acts
as the master node

This node
manages the
overall file system

HDFS

Name node

The name node stores

1. The directory structure

2. Metadata for all the files

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

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

Ext 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 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 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 computing problems. 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 computation 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, is 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.

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

Block 5

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

Block 6

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

Block 7

local intermediate files, the segment files (shown as `\tbox{a-f}\medstrut` `\tbox{g-p}\medstrut` `\tbox{q-z}\medstrut` in Figure 4.5). For the reduce phase , we want all values for a given key to be grouped 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

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 parsers shown in the figure.

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

First the file is broken into blocks of size 128 MB

HDFS

Ext Up previous contents index
Next: 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 [1] 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 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 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 computing problems. 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 computation 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, is 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

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).)

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. This is 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 \$\\$rightarrow\$ mapping.

Block 7

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 local intermediate files, the segment files (shown as \$\backslash\$box{a-T\$\backslash\$medstrut} \$\backslash\$box{g-p\$\backslash\$medstrut} \$\backslash\$box{l-q-z\$\backslash\$medstrut}\$ in Figure 4.5).

Block 8

For the reduce phase, we want all values for a given key to be grouped 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 thus 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) and one list is the task of the inverters in the reduce phase. The

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

Next: 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. The result of the construction process is a distributed index that is partitioned across several machines - either according to term or 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 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 computing 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 computation 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, is 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).)

Block 6

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

Block 7

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 local intermediate files, the segment files (shown as \$\rightarrow\$ \tbox{ta-T\medstrut} \$\rightarrow\$ \tbox{ig-p\medstrut} \$\rightarrow\$ \tbox{lq-z\medstrut} in Figure 4.5).

Block 8

For the reduce phase, we want all values for a given key to be collected 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: term) in one list is the task of the inverters in the reduce phase. The

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

What if one of the
blocks gets corrupted?

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 1

Block 1

Block 2

Data node 3

Block 5

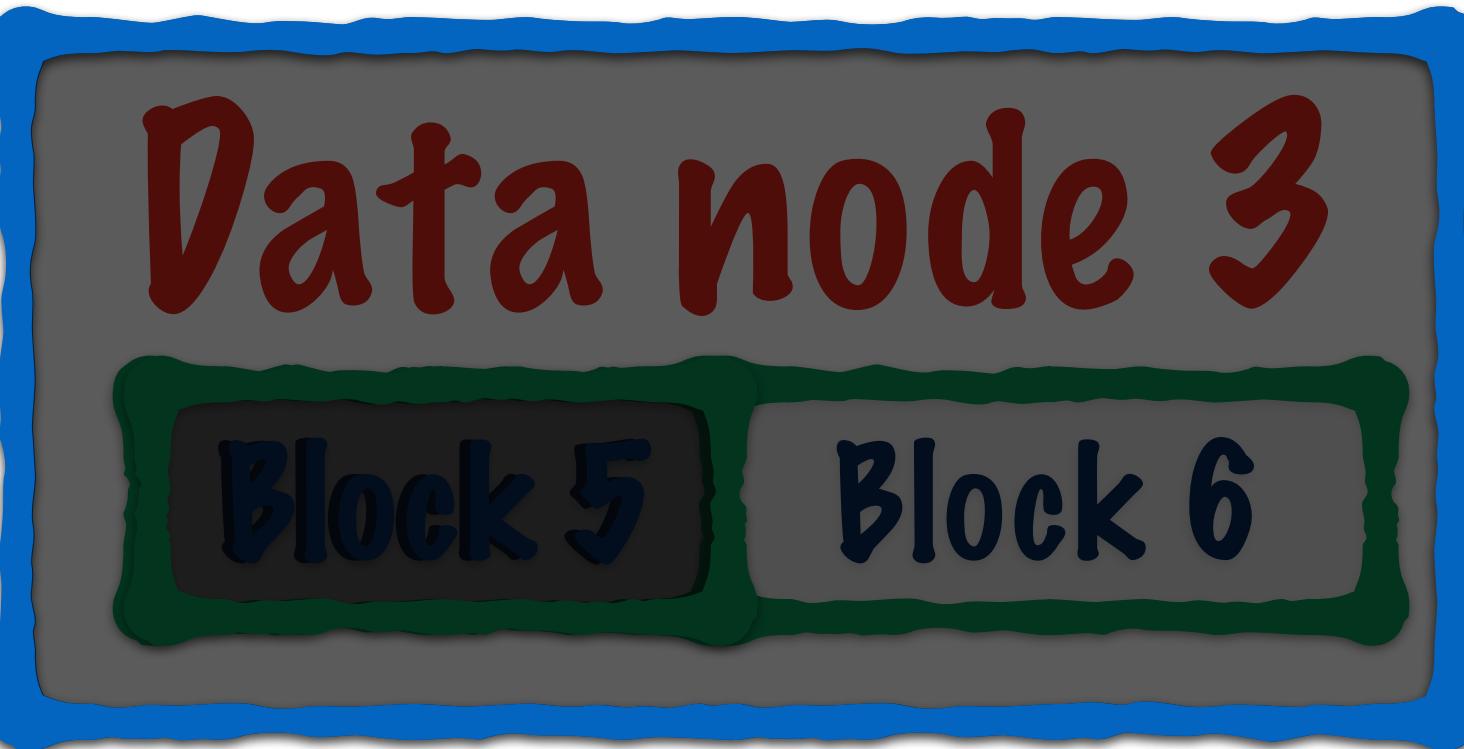
Block 6

Or one of the data
nodes crashes?

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



This is one of the key challenges in distributed storage

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

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

Data node 1

Block 1 Block 2

Data node 2

Block 3 Block 4

Block 1 Block 2

Data node 3

Block 5 Block 6

Name node

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

HDFS

Data node 1

Block 1

Block 2

Data node 3

Block 5

Block 6

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 file system to
manage the
storage of data

A framework to
process data across
multiple servers

MapReduce

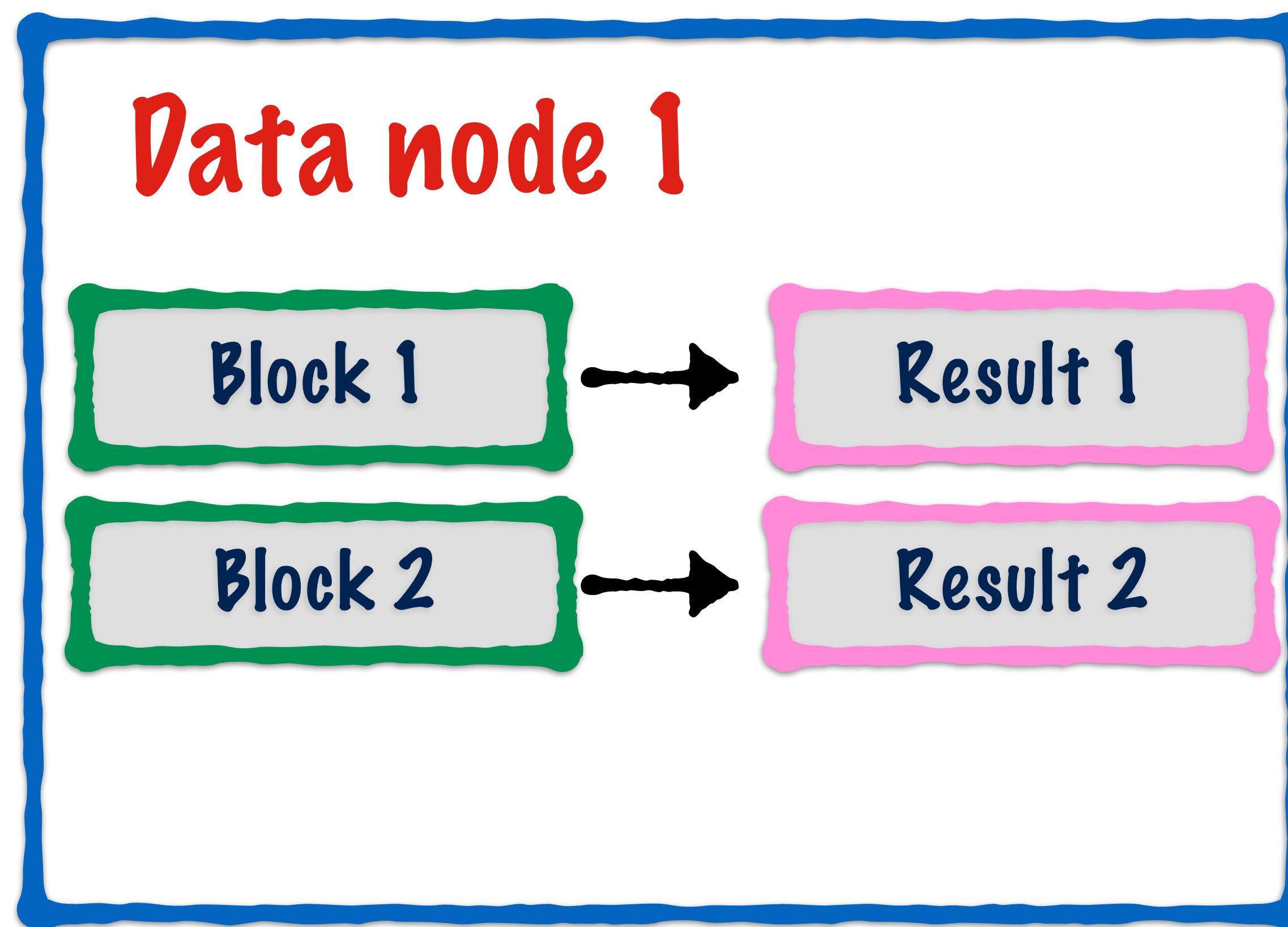
MapReduce is a way
to parallelize a data
processing task

MapReduce

MapReduce tasks
have 2 phases

MapReduce

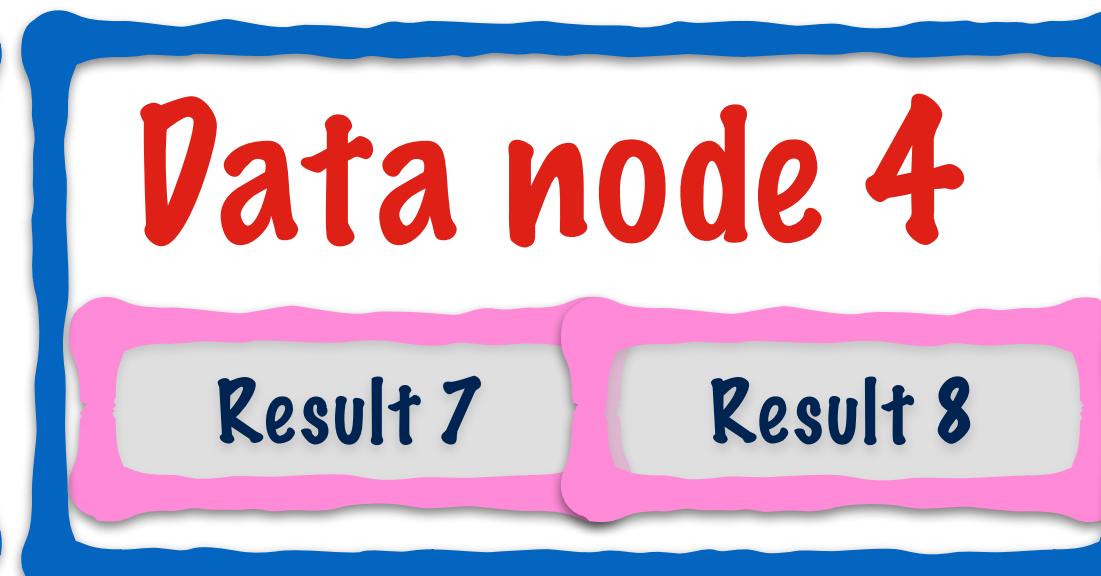
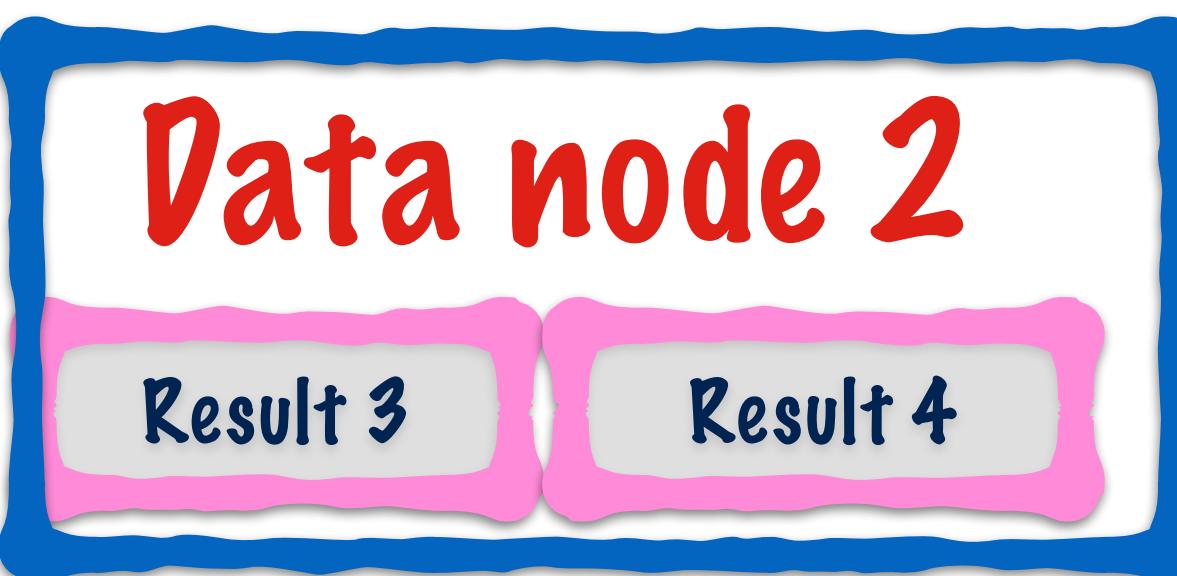
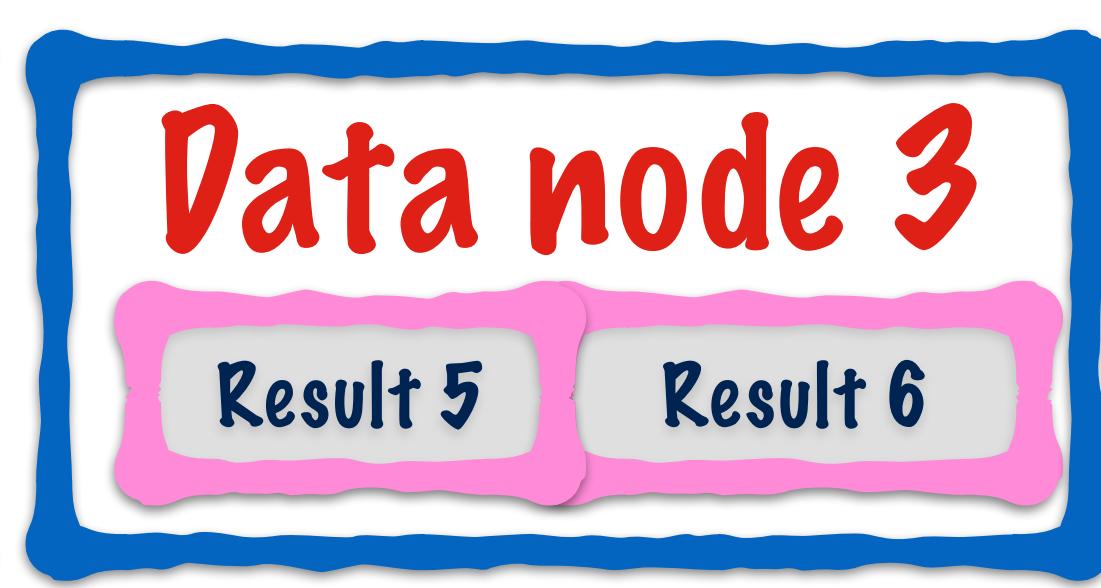
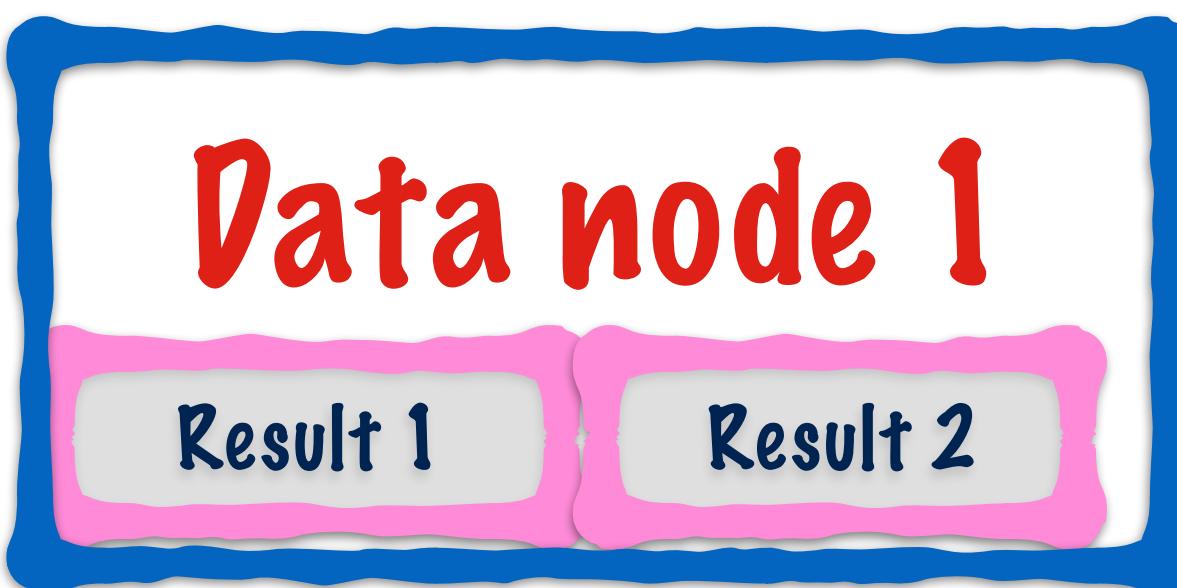
1. Process each block in the node it is stored in



Map phase

MapReduce

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



Name node
The name node stores metadata

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 file system to
manage the
storage of data

A framework to
process data across
multiple servers

Hadoop ecosystem

HDFS

Hadoop MapReduce

HBase

With Hadoop, you can

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

Hadoop ecosystem

HD**FS**

Hadoop MapReduce

H**Base**

Why then, do you need a separate architecture for database management?

Hadoop vs Databases

Databases are at **the heart of most** applications

e-mails

Sales

Bank accounts

Payroll

Hadoop vs Databases

e-mails

Sales

Bank accounts

Payroll

Databases that serve such applications do something called Transaction processing

Hadoop vs Databases

e-mails

Sales

Bank accounts

Payroll

They store data
in the form of
tables, rows,
columns

Hadoop vs Databases

e-mails
Sales
Bank accounts
Payroll

A transaction involves
Inserting, updating,
deleting data (or a
combination of these)

Hadoop vs Databases

e-mails
Sales
Bank accounts
Payroll

Transaction
processing has
certain requirements

Hadoop vs Databases

Hadoop has a few limitations
which make it unsuited for
transaction processing

Hadoop vs Databases

Hadoop limitations

1. Unstructured data
2. No random access
3. High latency
4. Not ACID compliant

Hadoop vs Databases

Hadoop limitations

1. Unstructured data
2. No random access
3. High latency
4. Not ACID compliant

Hadoop vs Databases

1. Unstructured data

Hadoop stores
data in HDFS

Hadoop vs Databases

1. Unstructured data

The data in HDFS is
Unstructured

Hadoop vs Databases

1. Unstructured data

Unlike databases, HDFS data doesn't have any schema

Hadoop vs Databases

1. Unstructured data

It's basically in the form of files

Text files

Log files

Video/Audio files

Hadoop vs Databases

1. Unstructured data

There's no concept of rows/columns

There are no tables

Hadoop vs Databases

1. Unstructured data

This is not to say that Hadoop can't be used to store structured data

Hadoop vs Databases

1. Unstructured data

You could store your data in a structured format
even in Hadoop

csv files

xml files

jsons

Hadoop vs Databases

1. Unstructured data

Each record
in these files
could be 1
row in a table

csv files
xml files
jsons

Hadoop vs Databases

1. Unstructured data

But unlike databases,
Hadoop will not
enforce the schema
or any constraints
on these rows/tables

csv files
xml files
jsons

Hadoop vs Databases

Hadoop limitations

1. Unstructured data
2. No random access
3. High latency
4. Not ACID compliant

Hadoop vs Databases

Hadoop limitations

1. Unstructured data
2. No random access
3. High latency
4. Not ACID compliant

Hadoop vs Databases

2. No random access

Applications that use databases
require random access

ie. the ability to create, access and
modify individual rows of a table

This is not possible with Hadoop

Hadoop vs Databases

2. No random access

HDFS is optimal for storing large files

MapReduce is optimal for processing these files as a **whole**

Hadoop vs Databases

2. No random access

If an HDFS file consists of many rows in a table

There is no provision to access or modify a specific row without processing the entire file

Hadoop vs Databases

Hadoop limitations

1. Unstructured data
2. No random access
3. High latency
4. Not ACID compliant

Hadoop vs Databases

Hadoop limitations

1. Unstructured data
2. No random access
3. High latency
4. Not ACID compliant

Hadoop vs Databases

3. High latency

Applications also require **low latency**

Any operations like inserting,
updating or deleting data should
occur as fast as possible

Hadoop vs Databases

3. High latency

All processing in Hadoop occurs via
MapReduce tasks on complete files

Even on large clusters, these tasks
might take **minutes or hours** at times

Hadoop vs Databases

Hadoop limitations

1. Unstructured data
2. No random access
3. High latency
4. Not ACID compliant

Hadoop vs Databases

Hadoop limitations

1. Unstructured data
2. No random access
3. High latency
4. Not ACID compliant

Hadoop vs Databases

4. Not ACID compliant

Databases are the
source of truth for the
data that they store

Hadoop vs Databases

4. Not ACID compliant

Databases guarantee ACID properties to maintain the integrity of their data

Hadoop vs Databases

4. Not ACID compliant

ACID
properties

Atomicity
Consistency
Isolation
Durability

Hadoop vs Databases

4. Not ACID compliant

Atomicity

Consistency

Isolation

Durability

Hadoop vs Databases

4. Not ACID compliant

Atomicity

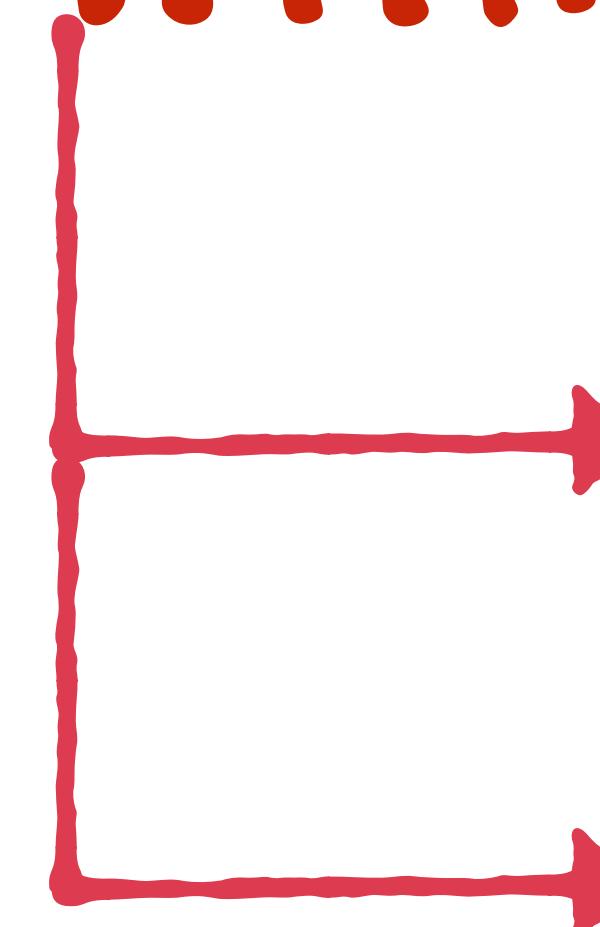
Operations (aka transactions) must be all-or-nothing

Hadoop vs Databases

4. Not ACID compliant

Atomicity

Example of a transaction :
Cash withdrawal from an ATM



Update cash balance

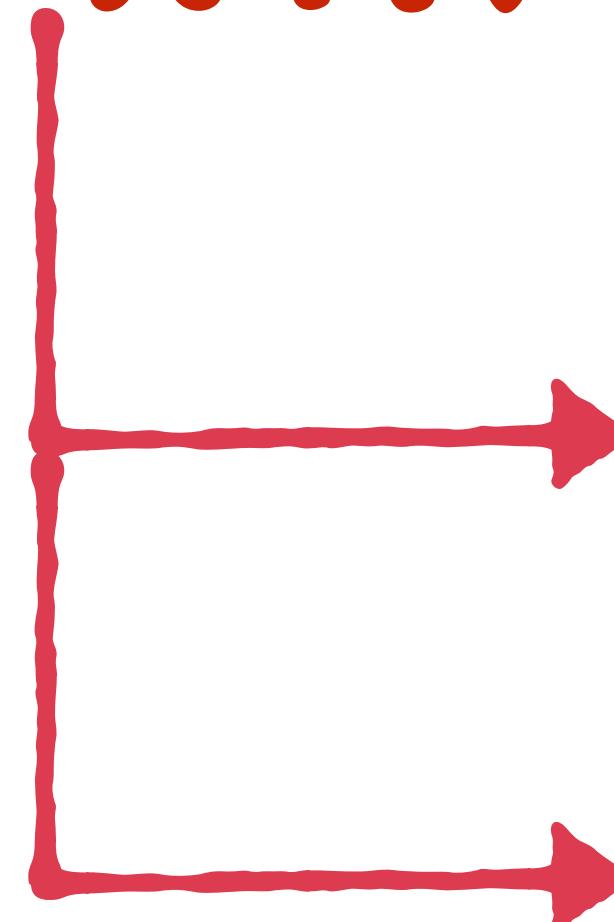
Update account balance

Hadoop vs Databases

Atomicity

4. Not ACID compliant

Cash withdrawal from an ATM



Update cash balance

Update account balance

If one of these fails, the whole transaction should fail

Hadoop vs Databases

4. Not ACID compliant

Atomicity
Consistency
Isolation
Durability

Any changes to the database **must not** violate any specified database constraints

Hadoop vs Databases

4. Not ACID compliant

Atomicity
Consistency
Isolation
Durability

If multiple/concurrent operations occur, the result is as if these operations are applied in sequence

Hadoop vs Databases

4. Not ACID compliant

Atomicity
Consistency
Isolation
Durability

Once a transaction
is executed, the
changes are
permanent

Hadoop vs Databases

4. Not ACID compliant

Atomicity
Consistency
Isolation
Durability

Traditional databases are designed to guarantee all of these properties

Hadoop vs Databases

4. Not ACID compliant

ACID guarantees require that
the database management
system is aware of the structure
and contents of the data

Hadoop vs Databases

4. Not ACID compliant

ACID guarantees require that the database management system **is aware** of the structure and contents of the data

HDFS being just a file storage system, has no such awareness

Hadoop vs Databases

Hadoop limitations

1. Unstructured data
2. No random access
3. High latency
4. Not ACID compliant

Hadoop vs Databases

Hadoop limitations

1. Unstructured data
2. No random access
3. High latency
4. Not ACID compliant

All these
limitations make
Hadoop unsuited
for transaction
processing

HBase

is a distributed database
management system that's part
of the Hadoop ecosystem

HBase

HBase uses HDFS to store
it's underlying data

HBase

HBase has the architecture
benefits of HDFS

1. Distributed storage
2. Fault tolerance

HBase

It also has many of the properties required for transaction processing

1. Awareness of the structure of data
2. Low latency
3. Random access
4. ACID compliant at some levels

HBase

To understand HBase

it's helpful understand how it's
different from a traditional
RDBMS

HBase vs RDBMS

In a traditional RDBMS, all operations like creating, inserting, updating rows are done using SQL

HBase does not support SQL

HBase vs RDBMS

Only CRUD operations

HBase only supports a basic set of operations (**Create-Read-Update-Delete**)

HBase vs RDBMS

(Create-Read-Update-Delete)

Only CRUD operations

All these operations have to
be applied at a row level

HBase vs RDBMS

(Create-Read-Update-Delete)

Only **CRUD** operations

HBase does not support any
operations across rows (or)
across tables

HBase vs RDBMS

(Create-Read-Update-Delete)

Only CRUD operations

This means that you cannot
perform operations like
Joins, Group by etc

HBase vs RDBMS

Only CRUD operations

Denormalized

HBase tables are **not designed**
using a relational data model

HBase vs RDBMS

Only CRUD operations

Denormalized

All the data pertaining to
an entity is stored in 1 row
(ie tables are denormalized)

HBase vs RDBMS

Only CRUD operations

Denormalized

Column oriented storage

HBase has a special kind of
data model

HBase vs RDBMS

Only **CRUD** operations

Denormalized

Column oriented storage

ACID at a row level

HBase is **ACID** compliant for limited kinds of transactions

HBase vs RDBMS

Column oriented storage

Denormalized

Only CRUD operations

ACID at a row level

Let's understand the implications of each of these

HBase vs RDBMS

Column oriented storage

Say we have an application that manages notifications to the users of a social network



A screenshot of a Facebook notifications feed. At the top, there's a search bar and a notifications icon. Below it, the word "Notifications" is displayed. The feed lists several notifications:

- Jessica [REDACTED] accepted your friend request.** Since she's new to Facebook, you should suggest people she knows.
" [REDACTED]
Jessica is new on Facebook. To welcome her:
[Suggest people that she knows](#)
[Write on her wall](#)
34 minutes ago
- Jon Dugan commented on your photo.** about an hour ago
- Daniel J. Adams commented on your link.** 2 hours ago
- Chaz Yoon commented on [REDACTED]'s photo.** 2 hours ago
- Brendan McLaughlin commented on your link.** 3 hours ago
- Brendan McLaughlin likes your link.** 3 hours ago
- Bruce Robert Abbott commented on your photo.** 3 hours ago
- Chaz Yoon commented on your link.** 3 hours ago

HBase vs RDBMS



The screenshot shows a notifications section from a social media platform. At the top, there's a search bar and a notifications icon. Below it, a heading says "Notifications". A message from "Jessica" is displayed: "Jessica [REDACTED] accepted your friend request. Since she's new to Facebook, you should suggest people she knows." It includes a small profile picture of Jessica and a link to her wall. Below this, there are several other notification items:

- Jon Dugan commented on your photo. about an hour ago
- Daniel J. Adams commented on your link. 2 hours ago
- [REDACTED] commented on [REDACTED]'s photo. 2 hours ago
- Brendan McLaughlin commented on your link. 3 hours ago
- Brendan McLaughlin likes your link. 3 hours ago
- Bruce Robert Abbott commented on your photo. 3 hours ago
- Chaz Yoon commented on your link. 3 hours ago

Column oriented storage

Here is a table that stores some notification related data

id	type	for user	from user	timestamp
1	Friend request status	Ryan	Jessica	146710201
2	Comment	Chaz	Daniel	146711200
3	Comment	Rick	Brendan	1467112205
4	Like	Rick	Brendan	1467112213

HBase vs RDBMS

Column oriented storage

This is how data is stored
in traditional databases

id	type	for user	from user	timestamp
1	Friend request status	Ryan	Jessica	146710201
2	Comment	Chaz	Daniel	146711200
3	Comment	Rick	Brendan	1467112205
4	Like	Rick	Brendan	1467112213

HBase vs RDBMS

Column oriented storage

A table with a fixed schema is defined

id	type	for user	from user	timestamp
1	Friend request status	Ryan	Jessica	146710201
2	Comment	Chaz	Daniel	146711200
3	Comment	Rick	Brendan	1467112205
4	Like	Rick	Brendan	1467112213

HBase vs RDBMS

Column oriented storage

Each row represents a data point

id	type	for user	from user	timestamp
1	Friend request status	Ryan	Jessica	146710201
2	Comment	Chaz	Daniel	146711200
3	Comment	Rick	Brendan	1467112205
4	Like	Rick	Brendan	1467112213

HBase vs RDBMS

Column oriented storage

In a column oriented store, each cell represents a datapoint

id	type	for user	from user	timestamp
1	Friend request status	Ryan	Jessica	146710201
2	Comment	Chaz	Daniel	146711200
3	Comment	Rick	Brendan	1467112205
4	Like	Rick	Brendan	1467112213

HBase vs RDBMS

Column oriented storage

id	type	for user	from user	timestamp
1	Friend request status	Ryan	Jessica	146710201
2	Comment	Chaz	Daniel	146711200
3	Comment	Rick	Brendan	1467112205
4	Like	Rick	Brendan	1467112213

Data is stored
in a map

Key = <Row id, Col id>

Value = <data>

HBase vs RDBMS

Column oriented storage

id	type	for user	from user	timestamp
1	Friend request status	Ryan	Jessica	146710201
2	Comment	Chaz	Daniel	146711200
3	Comment	Rick	Brendan	1467112205
4	Like	Rick	Brendan	1467112213

Data is stored
in a map

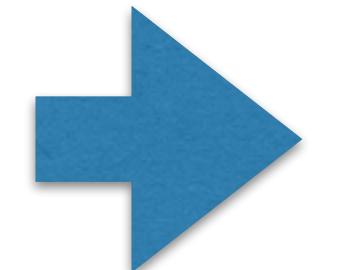
Key = 2, for_user

Value = Chaz

HBase vs RDBMS

Column oriented storage

id	type	for user	from user	timestamp
1	Friend request status	Ryan	Jessica	146710201
2	Comment	Chaz	Daniel	146711200
3	Comment	Rick	Brendan	1467112205
4	Like	Rick	Brendan	1467112213



row	column	value
1	type	Friend request status
1	for user	Ryan
1	from user	Jessica
1	timestamp	146710201
2	type	Comment
2	for user	Chaz
2	from user	Daniel
2	timestamp	146711200
3	type	Comment
3	for user	Rick
3	from user	Brendan
3	timestamp	1467112205

HBase vs RDBMS

Column oriented storage

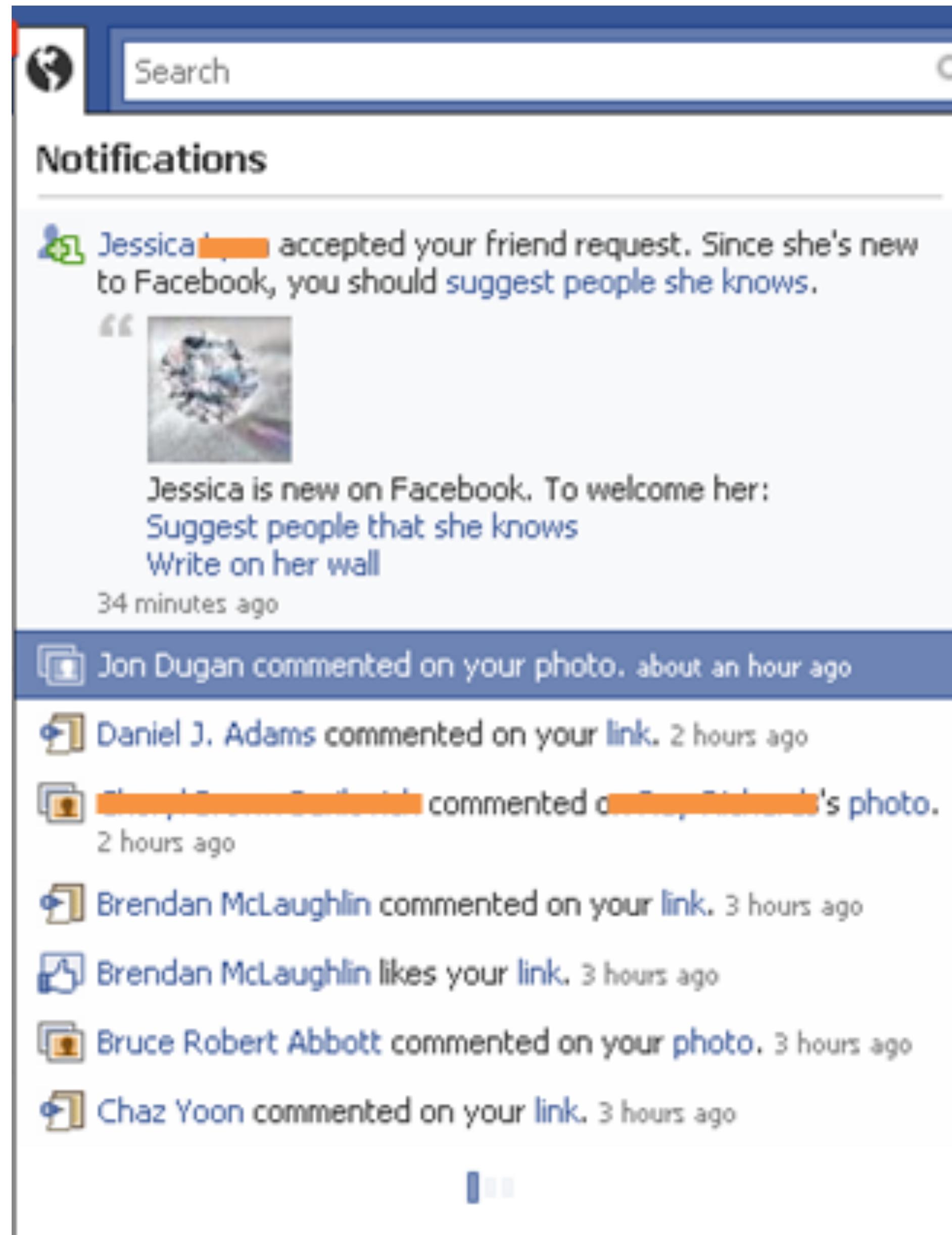
An HBase table
is in fact a
sorted map

Keys Values

row	column	value
1	type	Friend request status
	for user	Ryan
	from user	Jessica
	timestamp	146710201
2	type	Comment
	for user	Chaz
	from user	Daniel
	timestamp	146711200
3	type	Comment
	for user	Rick
	from user	Brendan

HBase vs RDBMS

Column oriented storage



A screenshot of a Facebook notifications page. At the top, there's a search bar and a globe icon. Below it, a section titled "Notifications" shows a friend request from Jessica. It includes a small profile picture of a diamond ring, a welcome message, and three interaction options: "Suggest people she knows", "Write on her wall", and a timestamp "34 minutes ago". Below this, a blue header bar indicates a comment from "Jon Dugan". The main list continues with comments from "Daniel J. Adams", "Brendan McLaughlin", and others, each with a timestamp like "2 hours ago" or "3 hours ago".

Let's say some notifications have special attributes depending on their type

Column oriented storage

Friend request notifications might have information about the friend

Jessica is new on Facebook. To welcome her:

Suggest people that she knows

Write on her wall

34 minutes ago



Jon Dugan commented on your photo. about an hour ago



Daniel J. Adams commented on your link. 2 hours ago



[REDACTED] commented on [REDACTED]'s photo.

2 hours ago



Brendan McLaughlin commented on your link. 3 hours ago



Brendan McLaughlin likes your link. 3 hours ago



Bruce Robert Abbott commented on your photo. 3 hours ago



Chaz Yoon commented on your link. 3 hours ago



Jessica is new on Facebook. To welcome her:
Suggest people that she knows
Write on her wall

34 minutes ago

-  Jon Dugan commented on your photo. about an hour ago
-  Daniel J. Adams commented on your link. 2 hours ago
-  [REDACTED] commented on [REDACTED]'s photo.
2 hours ago
-  Brendan McLaughlin commented on your link. 3 hours ago
-  Brendan McLaughlin likes your link. 3 hours ago
-  Bruce Robert Abbott commented on your photo. 3 hours ago
-  Chaz Yoon commented on your link. 3 hours ago

Column oriented storage

Comments and likes have information about a link or photo that prompted them

HBase vs RDBMS

Column oriented storage

id	type	for user	from user	timestamp	friend type	commented on
1	Friend request	Ryan	Jessica	146710201	new	-
2	Comment	Chaz	Daniel	146711200	-	link
3	Comment	Rick	Brendan	1467112205	-	photo
4	Like	Rick	Brendan	1467112213	-	-

In the **RDBMS** table, each of these attributes becomes a new column

HBase vs RDBMS

Column oriented storage

id	type	for user	from user	timestamp	friend type	commented on
1	Friend request	Ryan	Jessica	146710201	new	-
2	Comment	Chaz	Daniel	146711200	-	link
3	Comment	Rick	Brendan	1467112205	-	photo
4	Like	Rick	Brendan	1467112213	-	-

This results in tables that
are **very sparse**

HBase vs RDBMS

Column oriented storage

id	type	for user	from user	timestamp	friend type	commented on
1	Friend request	Ryan	Jessica	146710201	new	-
2	Comment	Chaz	Daniel	146711200	-	link
3	Comment	Rick	Brendan	1467112205	-	photo
4	Like	Rick	Brendan	1467112213	-	-

In an RDBMS, Sparse tables utilize disk space even for these empty cells

HBase vs RDBMS

Column oriented storage

id	type	for user	from user	timestamp	friend type	commented on
1	Friend request	Ryan	Jessica	146710201	new	-
2	Comment	Chaz	Daniel	146711200	-	link
3	Comment	Rick	Brendan	1467112205	-	photo
4	Like	Rick	Brendan	1467112213	-	-

In a column-oriented store, these cells can be skipped completely

HBase vs RDBMS

Column oriented storage

id	type	for user	from user	timestamp	friend type	commented on
1	Friend request status	Ryan	Jessica	146710201	new	link
2	Comment	Chaz	Daniel	146711200	-	link
3	Comment	Rick	Brendan	1467112205	-	photo
4	Like	Rick	Brendan	1467112213	-	-

row id	column	value
1	type	Friend request status
1	for user	Ryan
1	from user	Jessica
1	timestamp	146710201
1	friend type	new

HBase vs RDBMS

Column oriented storage

id	type	for user	from user	timestamp	friend type	commented on
1	Friend request status	Ryan	Jessica	146710201	new	link
2	Comment	Chaz	Daniel	146711200	new	link
3	Comment	Rick	Brendan	1467112205	-	photo
4	Like	Rick	Brendan	1467112213	-	-

row id	column	value
1	type	Friend request status
1	for user	Ryan
1	from user	Jessica
1	timestamp	146710201
1	friend type	new
2	type	Friend request status
2	for user	Ryan
2	from user	Jessica
2	timestamp	146710201
2	commented on	link

HBase vs RDBMS

Column oriented storage

Column oriented storage has some powerful advantages

1. You can store **really sparse** tables very efficiently
2. You can accommodate dynamically changing attributes

HBase vs RDBMS

Column oriented storage

Each row id can have a different set of col ids

1. You can store really sparse tables very efficiently

2. You can accommodate dynamically changing attributes

HBase vs RDBMS

Column oriented storage

The schema for a row id is not fixed, you can keep changing it

ie, Add or remove new col ids

2. You can accommodate dynamically changing attributes

HBase vs RDBMS

Column oriented storage ✓

Denormalized

Only CRUD operations

ACID at a row level

HBase vs RDBMS

Denormalized

LET'S SAY WE HAVE AN EMPLOYEES DATABASE

WE WANT TO CAPTURE EMPLOYEE NAME,
ADDRESS, SUBORDINATES

HBase vs RDBMS

Denormalized

A TRADITIONAL RDBMS WOULD MODEL IT AS 3 TABLES

EmplID	EmpName	AddressId
1	Vitthal	1

AddressId	Street	City
1	Bellandur	Bangalore

EmplID	SubordinateEmplID
1	3
1	4
1	8

HBase vs RDBMS

Denormalized

A TRADITIONAL RDBMS WOULD MODEL IT AS 3 TABLES

EmplID	EmpName	AddressId
1	Vitthal	1

AddressId	Street	City
1	Bellandur	Bangalore

EmplID	SubordinateEmplID
1	3
1	4
1	8

THIS KIND OF DESIGN
MINIMIZES REDUNDANT
STORAGE OF DATA

HBase vs RDBMS

Denormalized

EmplID	EmpName	AddressId
1	Vitthal	1

AddressId	Street	City
1	Bellandur	Bangalore

EmplID	SubordinateEmplID
1	3
1	4
1	8

THIS KIND OF DESIGN
MINIMIZES REDUNDANT
STORAGE OF DATA

HBase vs RDBMS

Denormalized

EmplID	EmpName	AddressId
1	Vitthal	1

AddressId	Street	City
1	Bellandur	Bangalore

EmplID	SubordinateEmplID
1	3
1	4
1	8

THESE STREET AND CITY NAMES
ARE ONLY STORED ONCE

AND REFERRED TO BY AN
INTEGER ID THEREAFTER

HBase vs RDBMS

Denormalized

EmplID	EmpName	AddressId
1	Vitthal	1

AddressId	Street	City
1	Bellandur	Bangalore

EmplID	SubordinateEmplID
1	3
1	4
1	8

**NORMALIZATION
OPTIMIZES FOR
STORAGE**

HBase vs RDBMS

Denormalized

EmplID	EmpName	AddressId
1	Vitthal	1

AddressId	Street	City
1	Bellandur	Bangalore

EmplID	SubordinateEmplID
1	3
1	4
1	8

**IN A DISTRIBUTED SYSTEM,
STORAGE IS CHEAP**

**INSTEAD YOU NEED TO
OPTIMIZE DISK SEEKS**

HBase vs RDBMS

Denormalized

EmplID	EmpName	AddressId
1	Vitthal	1

AddressId	Street	City
1	Bellandur	Bangalore

EmplID	SubordinateEmplID
1	3
1	4
1	8

IF YOU STORE DATA
ACROSS DIFFERENT TABLES

YOU HAVE TO PERFORM
DISK SEEKS FOR EACH TABLE

HBase vs RDBMS

Denormalized

INSTEAD WE CAN EMBED ALL 3 TABLES INTO A SINGLE TABLE

EmplID	EmpName	Address	Subordinates
1	Vitthal	<STRUCT>	<ARRAY>

“Street”: “Bellandur”,
“City”: “Bangalore”

“Anuradha”,
“Arun”,
“Swetha”)

HBase vs RDBMS

Denormalized

EmplID	EmpName	Address	Subordinates
1	Vitthal	<STRUCT>	<ARRAY>

THIS IS A
DENORMALIZED
DESIGN

HBase vs RDBMS

Denormalized

EmplID	EmpName	Address	Subordinates
1	Vitthal	<STRUCT>	<ARRAY>

ALL THE DATA
CORRESPONDING TO AN
EMPLOYEE IS STORED
IN A SINGLE TABLE

HBase vs RDBMS

Denormalized

IN HBASE DATA IS STORED IN A
DENORMALIZED MANNER

HBase vs RDBMS

Column oriented storage ✓

Denormalized ✓

Only CRUD operations

ACID at a row level

HBase vs RDBMS Only CRUD operations

HBase architecture is designed such that you can get random read-write access to a specific row

HBase vs RDBMS Only CRUD operations

Unlike, traditional
RDBMS, HBase does
not support SQL

NoSQL

HBase vs RDBMS Only CRUD operations

HBase only supports a
limited set of
operations

HBase vs RDBMS Only CRUD operations

HBase only supports a limited set of operations

Create

Add a new value to the table

Read

Read the value for a specific row id, col id

Update

Update the value for a specific row id, col id

Delete

Delete the value for a specific row id, col id

HBase vs RDBMS Only CRUD operations

Create

Read

Update

Delete

All HBase operations
deal with a specific
row

HBase vs RDBMS Only CRUD operations

Create

Read

Update

Delete

HBase does not support
any operations across
tables

No Joins

No Foreign key
constraints

HBase vs RDBMS Only CRUD operations

Create

Read

Update

Delete

HBase does not support
any operations across
row ids

No Grouping/Aggregation

HBase vs RDBMS Only CRUD operations

Create

Read

Update

Delete

This is another reason
why denormalization
is important in HBase

HBase vs RDBMS Only CRUD operations

Create

Read

Update

Delete

All the data needed to
describe an entity
should be self-contained
within its row id

HBase vs RDBMS Only CRUD operations

LET'S GO BACK TO THE EMPLOYEE EXAMPLE

A TRADITIONAL RDBMS WOULD MODEL IT AS 3 TABLES

EmplID	EmpName	AddressId
1	Vitthal	1

AddressId	Street	City
1	Bellandur	Bangalore

EmplID	SubordinateEmplID
1	3
1	4
1	8

HBase vs RDBMS Only CRUD operations

WHEN AN APPLICATION ASKS FOR AN EMPLOYEE'S DETAILS

EmplID	EmpName	AddressId
1	Vitthal	1

AddressId	Street	City
1	Bellandur	Bangalore

**YOU WOULD NEED TO JOIN 2 TABLES
TO FETCH THE ADDRESS**

HBase vs RDBMS Only CRUD operations

WHEN AN APPLICATION ASKS FOR AN EMPLOYEE'S DETAILS

EmplID	EmpName	AddressId
1	Vitthal	1

EmplID	SubordinateEmplID
1	3
1	4
1	8

**YOU WOULD NEED TO
JOIN THESE 2 TABLES
TWICE TO GET THE
LIST OF SUBORDINATES
FOR AN EMPLOYEE**

HBase vs RDBMS Only CRUD operations

IN AN RDBMS THESE JOINS CAN
BE MADE EFFICIENT WITH THE
ADDITION OF INDICES

HBase vs RDBMS Only CRUD operations

IN HBASE, THERE IS NO SUPPORT FOR
JOINING TABLES ON THE FLY WHILE
FETCHING THE DETAILS FOR 1 ROW

HBase vs RDBMS Only CRUD operations

YOU COULD USE AN EXTERNAL
APPLICATION LIKE MAPREDUCE
TO PERFORM JOINS

WHILE THIS IS FINE FOR ANALYTICAL
QUERIES, IT WOULD NOT BE SUITABLE
FOR TRANSACTION PROCESSING

HBase vs RDBMS Only CRUD operations IN A DENORMALIZED DESIGN

EmplID	EmpName	Address	Subordinates
1	Vitthal	<STRUCT>	<ARRAY>

YOU CAN USE THE HBASE SUPPORTED
READ OPERATION TO READ THE
ROW AND FETCH ALL THE DATA.

HBase vs RDBMS

Column oriented storage ✓

Denormalized ✓

Only CRUD operations ✓

ACID at a row level

HBase vs RDBMS ACID at a row level

HBase is ACID compliant, but
only at a row id level

For example, let's look at
Atomicity

HBase vs RDBMS

ACID at a row level

Atomicity

Transaction 1:
Update values
for 2 col ids
within 1 row

Transaction 2:
Update values
for 2 col ids
for 10 row ids

HBase vs RDBMS

ACID at a row level

Atomic

Transaction 1:

Update values
for 2 col ids
within 1 row

Atomicity

Transaction 2:

If one col id update
fails, the entire
transaction fails

HBase vs RDBMS

ACID at a row level

Atomicity

If the operation fails after 5 row ids are updated, the row ids which are updated remain updated

Not Atomic

Transaction 2:

Update values
for 2 col ids
for 10 row ids

HBase vs RDBMS

Column oriented storage ✓

Denormalized ✓

Only CRUD operations ✓

ACID at a row level. ✓

If you are familiar with the Hadoop ecosystem, you might know of other technologies which seem similar to HBase

HIVE FOR INSTANCE

HBase is a database management system

Used for both transaction processing and analytical processing

HIVE IS A DATA WAREHOUSE

Used only for analytical processing

HBase is a database management system

Provides low latency and random access for some supported operations

HIVE IS A DATA WAREHOUSE

Only suitable for batch processing jobs that can tolerate high latency

HBase does not provide
any SQL interface

Hive does!

HIVE

HADOOP

HDFS

MapReduce

YARN

HIVE IS A DATAWAREHOUSE
BUILT ON TOP OF HADOOP

HIVE

HADOOP

HDFS

MapReduce

YARN

HIVE STORES IT'S DATA
AS FILES IN HDFS

HIVE

HADOOP

HDFS

MapReduce

YARN

All processing tasks in Hadoop
are run using MapReduce tasks

HIVE

HADOOP

HDFS

MapReduce

YARN

MapReduce tasks are usually
written using a Java Framework

HIVE

HADOOP

HDFS

MapReduce

YARN

Writing these MapReduce
tasks can be pretty daunting

HIVE

HADOOP

HDFS

MapReduce

YARN

Traditional databases/closed-source
data warehouses normally use **SQL**

HIVE

HADOOP

HDFS

MapReduce

YARN

SQL = Structured Query
Language

SQL = Structured Query Language

**SQL is really much easier to use
and understand :)**

SQL = Structured Query Language

It's widely used by analysts and
programmers to work with
databases/data warehouses

SQL = Structured Query Language

**SQL has a few easy to
understand constructs**

Select, group by, join etc

SQL = Structured Query Language

Most data processing tasks are defined
using a combination of these constructs

Select, group by, join etc

HIVE

HADOOP

HDFS

MapReduce

YARN

HIVE PROVIDES AN SQL LIKE
INTERFACE TO DATA IN HDFS

HIVE

HADOOP

HDFS

MapReduce

YARN

THE FILES IN HDFS ARE EXPOSED TO
THE USER IN THE FORM OF TABLES

HIVE

HADOOP

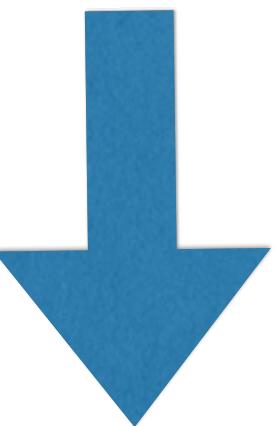
HDFS

MapReduce

YARN

THE USER CAN WRITE SQL-LIKE
QUERIES TO WORK WITH THESE TABLES

SQL-LIKE QUERY



HIVE

HADOOP

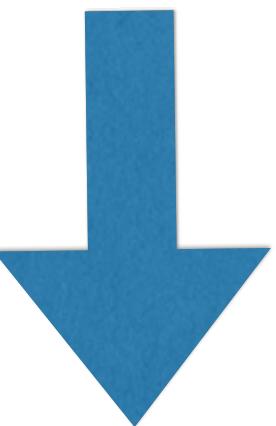
HDFS

MapReduce

YARN

HIVE WILL
TRANSLATE THE
QUERY INTO 1/MORE
MAPREDUCE TASKS

SQL-LIKE QUERY



HIVE



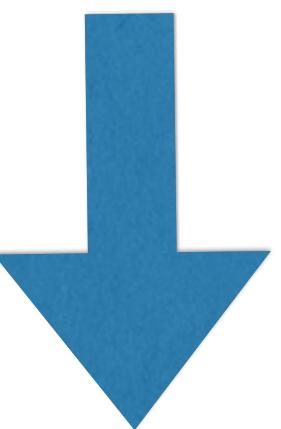
HDFS

MapReduce

YARN

THE MAPREDUCE
TASKS WILL PROCESS
THE DATA IN HDFS
AND RETURN ANY
RESULTS TO HIVE

SQL-LIKE QUERY



HIVE

HADOOP

HDFS

MapReduce

YARN

THE QUERIES ARE
WRITTEN IN A
SQL LIKE
LANGUAGE
CALLED **HIVEQL**

DIFFERENCES BETWEEN HIVE AND HBASE

HIVE

USED FOR BATCH
PROCESSING

HBASE

USED FOR BOTH
BATCH AND
TRANSACTION
PROCESSING

HIVE

USED FOR BATCH PROCESSING

PROVIDES AN SQL
SKIN FOR HADOOP

HBASE

USED FOR BOTH BATCH AND
TRANSACTION PROCESSING

NO SQL
INTERFACE

HIVE

USED FOR BATCH PROCESSING
PROVIDES AN **SQL SKIN FOR**
HADOOP

**USES BOTH HDFS
AND THE
MAPREDUCE ENGINE**

HBASE

USED FOR BOTH BATCH AND
TRANSACTION PROCESSING
NO SQL INTERFACE

**USES HDFS BUT
HAS IT'S OWN
ARCHITECTURE**

HIVE

USED FOR BATCH PROCESSING
PROVIDES AN SQL SKIN FOR
HADOOP

USES BOTH HDFS AND THE
MAPREDUCE ENGINE

DATA MODEL IS
SIMILAR TO
DATABASES (TABLES
WITH FIXED SCHEMA)

HBASE

USED FOR BOTH BATCH AND
TRANSACTION PROCESSING
NO SQL INTERFACE

USES HDFS BUT HAS IT'S OWN
ARCHITECTURE

DATA MODEL IS
COLUMN ORIENTED
STORAGE