

Module - Big Data

Overview

Understanding Hadoop

Why Cloud Computing

Data is getting bigger

World is getting smaller

Nothing fits in-memory on a single machine

Big Data, Small World



Super-computer

Cluster of generic
computers

Big Data, Small World



Monolithic

Distributed



Distributed

Lots of cheap hardware
Replication & fault tolerance
Distributed computing



Distributed

Lots of cheap hardware

HDFS

Replication & fault tolerance

YARN

Distributed computing

MapReduce



Distributed

“Clusters”

“Nodes”

“Server Farms”

“Server Farms”



Many Faces of Tim Duncan

[r/nba-memes](#) | [tumblr](#)

All of these servers need to be
co-ordinated by a single piece
of software

Single Co-ordinating Software



Many Faces of Tim Duncan

nba-memes | tumblr

Partition data

Co-ordinate computing tasks

Handle fault tolerance and recovery

Allocate capacity to processes

Single Co-ordinating Software

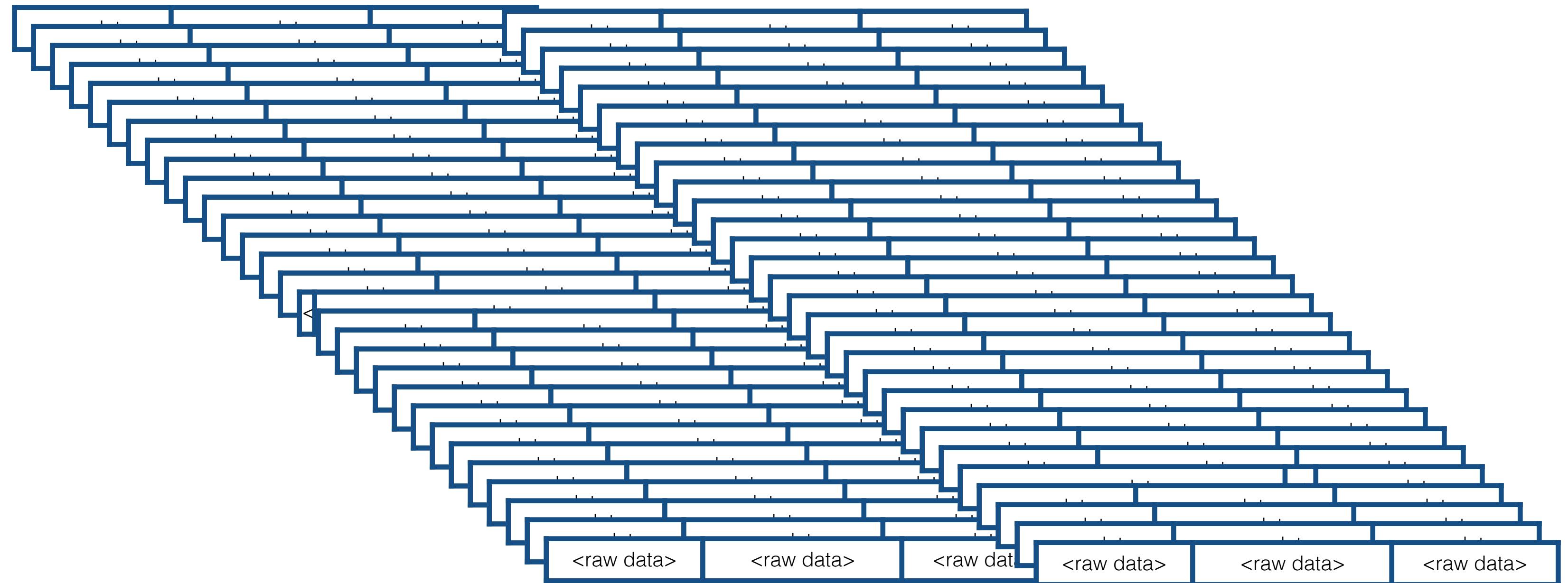


Many Faces of Tim Duncan

nba-memes | tumblr

Google developed proprietary
software to run on these
distributed systems

Single Co-ordinating Software



First: store millions of records on multiple machines

Single Co-ordinating Software



Many Faces of Tim Duncan

nba-memes | tumblr



Second: run processes on all these machines to crunch data

Single Co-ordinating Software

Google File System

To solve distributed
storage

MapReduce

To solve distributed
computing

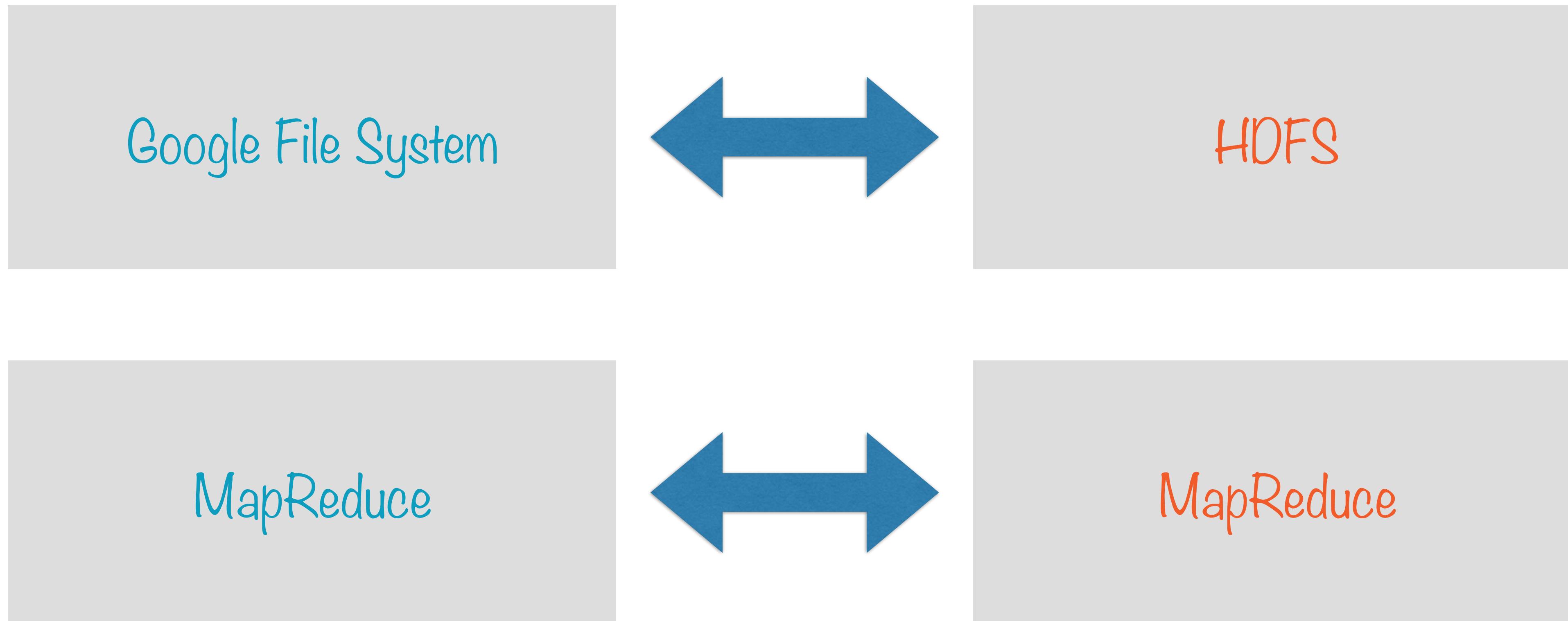
Single Co-ordinating Software

Google File System

MapReduce

Apache developed open source
versions of these technologies

Single Co-ordinating Software



Hadoop

The diagram consists of a large blue-bordered box containing two smaller gray boxes. The left gray box contains the text "HDFS" in orange. The right gray box contains the text "MapReduce" in orange.

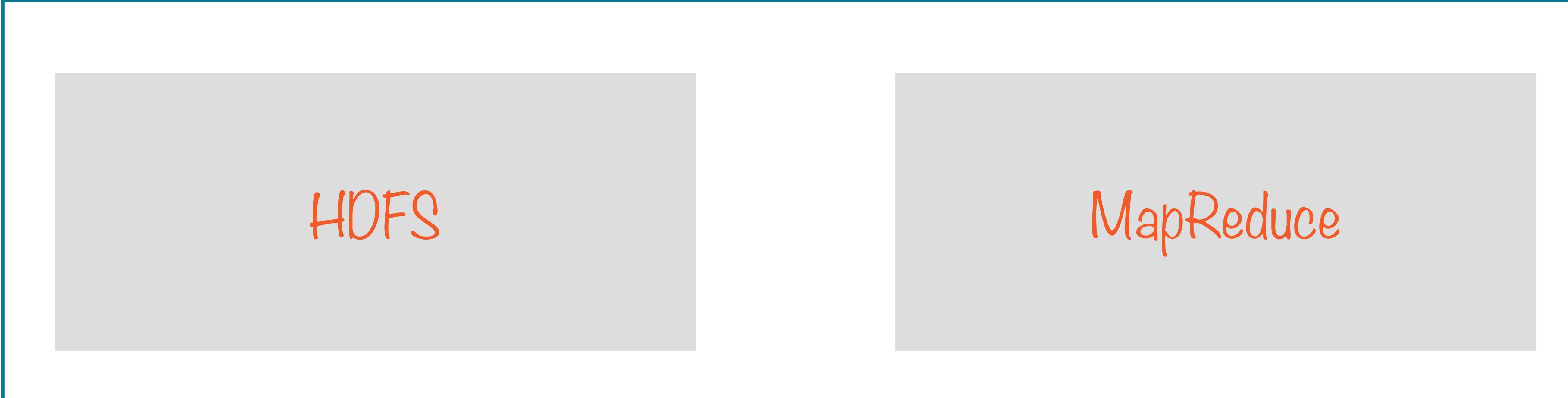
HDFS

MapReduce

A file system to manage
the storage of data

A framework to process
data across multiple servers

Hadoop



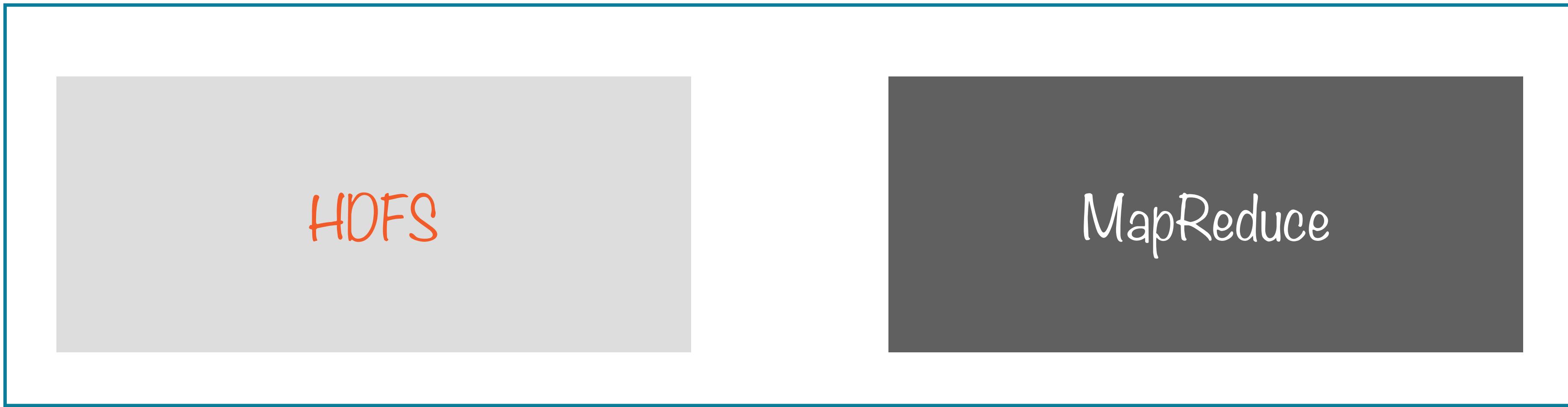
A diagram illustrating the Hadoop framework. It features a large blue-bordered rectangle representing the overall Hadoop system. Inside this rectangle, there are two gray rectangular boxes. The left box contains the text "HDFS" in orange. The right box contains the text "MapReduce" in orange.

HDFS

MapReduce

In 2013, Apache released Hadoop 2.0

Hadoop



MapReduce was broken into two separate
parts

Hadoop

HDFS

MapReduce

YARN

A framework to define
a data processing task

A framework to run
the data processing task

Hadoop

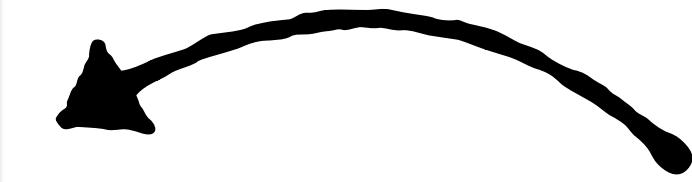
HDFS

MapReduce

YARN

Each of these components have corresponding configuration files

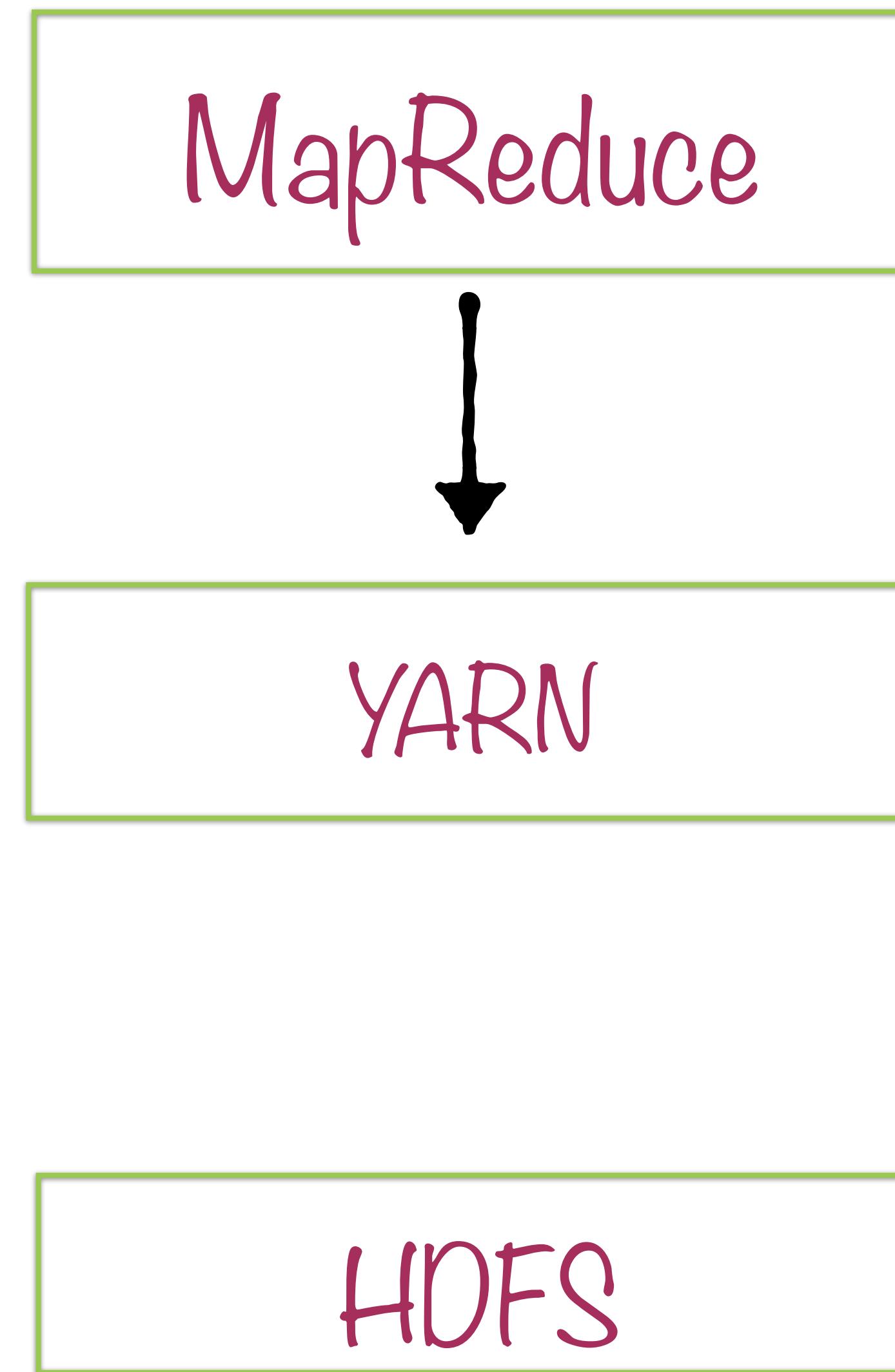
Co-ordination Between Hadoop Blocks



User defines map and
reduce tasks using the

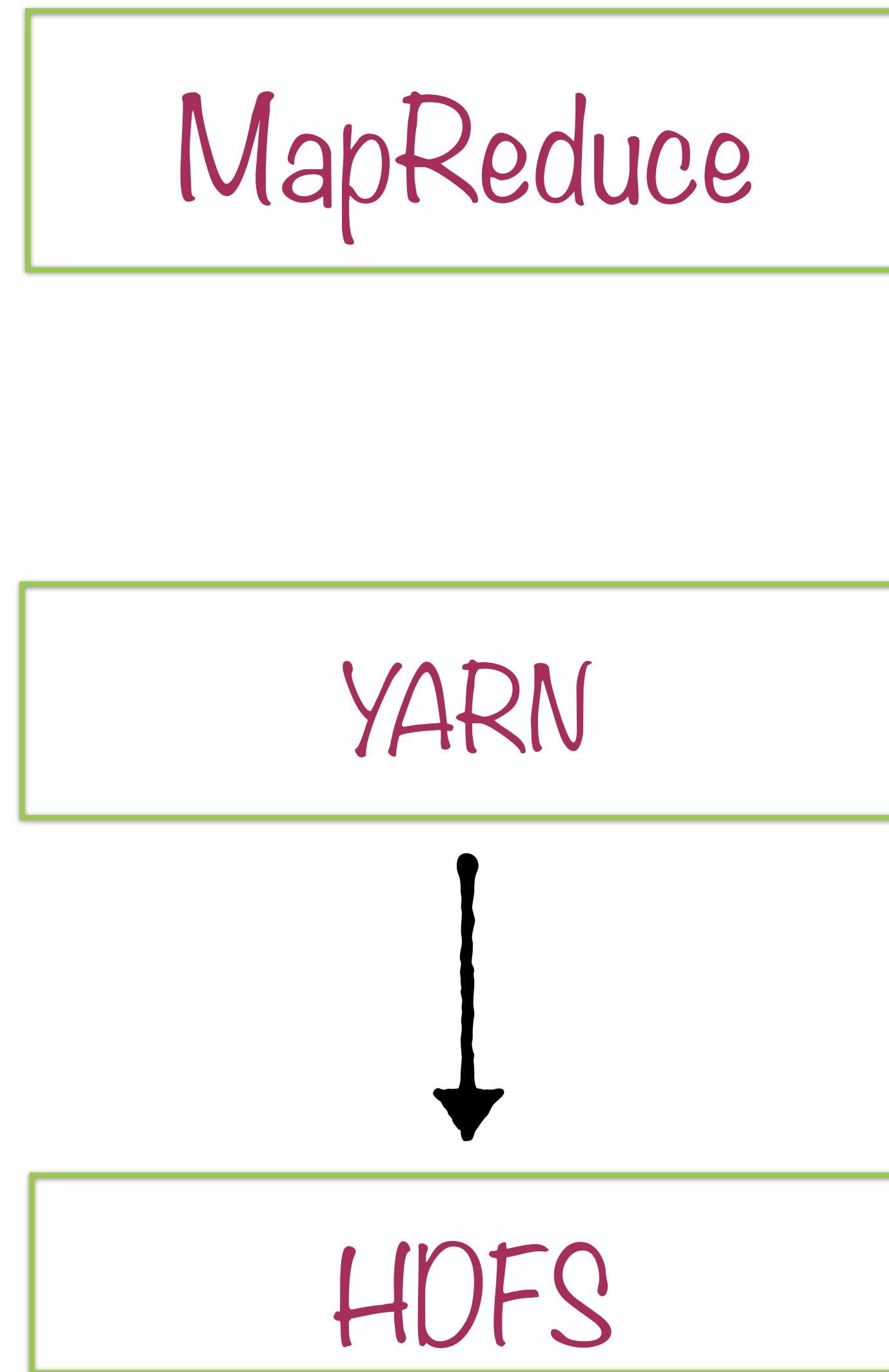
MapReduce API

Co-ordination Between Hadoop Blocks



A job is triggered on
the cluster

Co-ordination Between Hadoop Blocks



YARN figures out where and how to run the job, and stores the result in HDFS

Hadoop Ecosystem



Hadoop

An ecosystem of tools have sprung up around this
core piece of software

Hadoop Ecosystem

Hive

HBase

pig

Hadoop

Kafka

Spark

Oozie

Hadoop Ecosystem

Hive

HBase

pig

Kafka

Spark

Oozie

Hive

Provides an SQL interface to Hadoop

The bridge to Hadoop for folks who don't have
exposure to OOP in Java

HBase

A database management system on top of
Hadoop

Integrates with your application just like a
traditional database

Pig

A data manipulation language

Transforms unstructured data into a
structured format

Query this structured data using interfaces
like Hive

Spark

A distributed computing engine used along with Hadoop

Interactive shell to quickly process datasets

Has a bunch of built in libraries for machine learning, stream processing, graph processing etc.

Oozie

A tool to schedule workflows on all the
Hadoop ecosystem technologies

Kafka

Stream processing for unbounded datasets

Hadoop Ecosystem

Hive

HBase

pig

Hadoop

Kafka

Spark

Oozie

Understanding HDFS

HDFS

Hadoop Distributed File System



HDFS



Built on commodity hardware

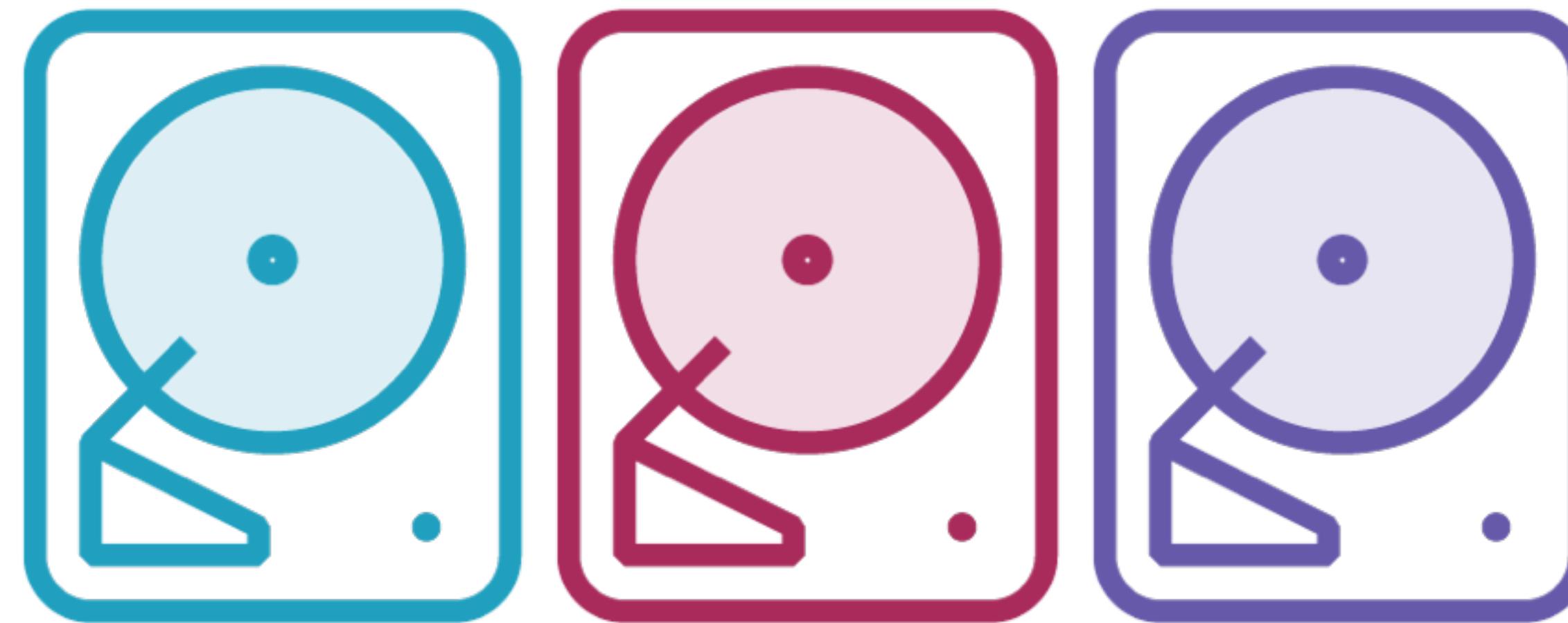
Highly fault tolerant, hardware failure is the norm

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

Supports very large data sets

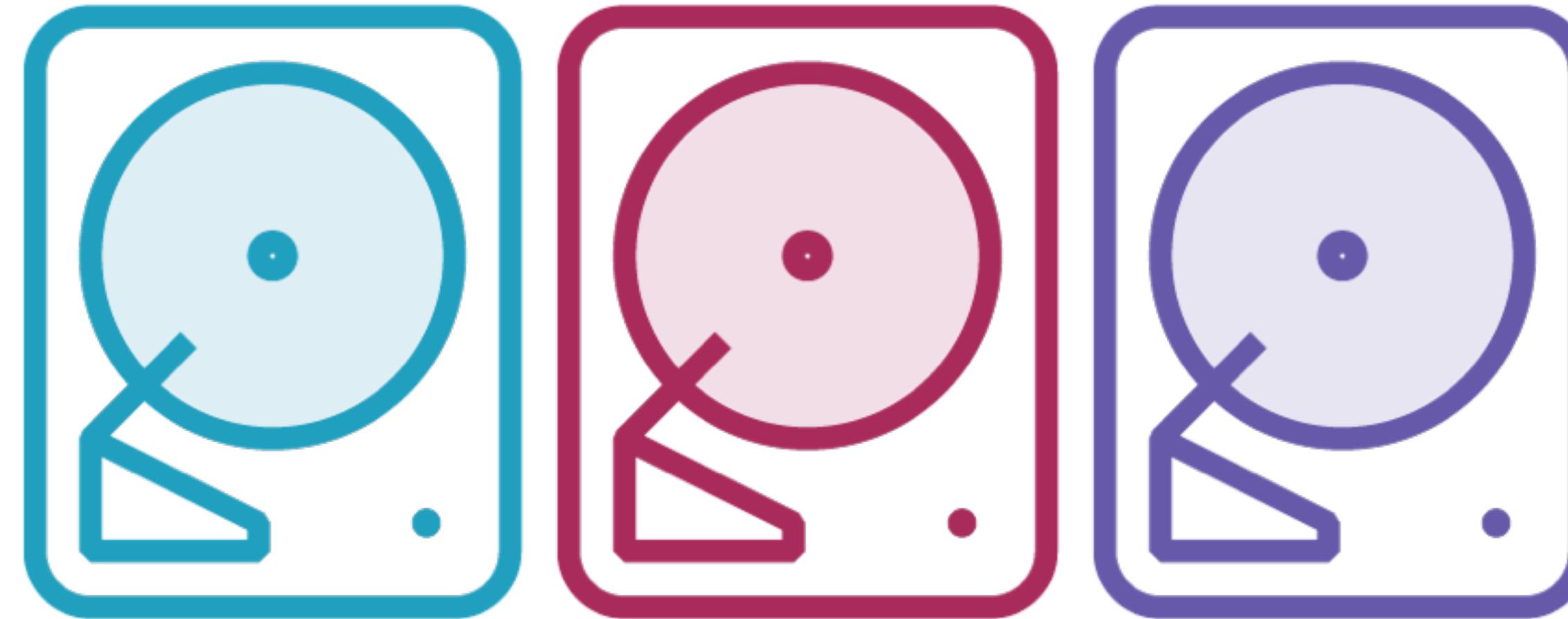
HDFS

Manage file storage across
multiple disks



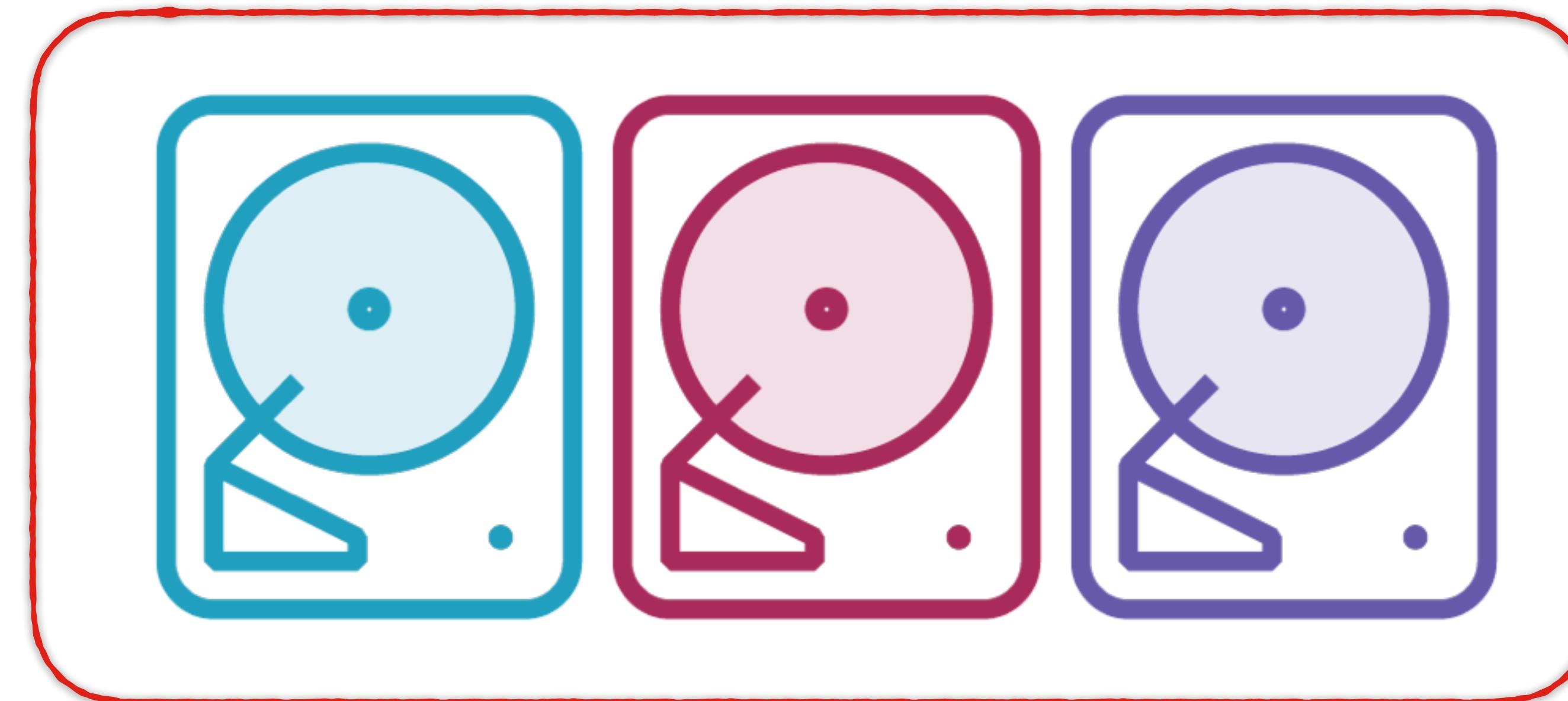
HDFS

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



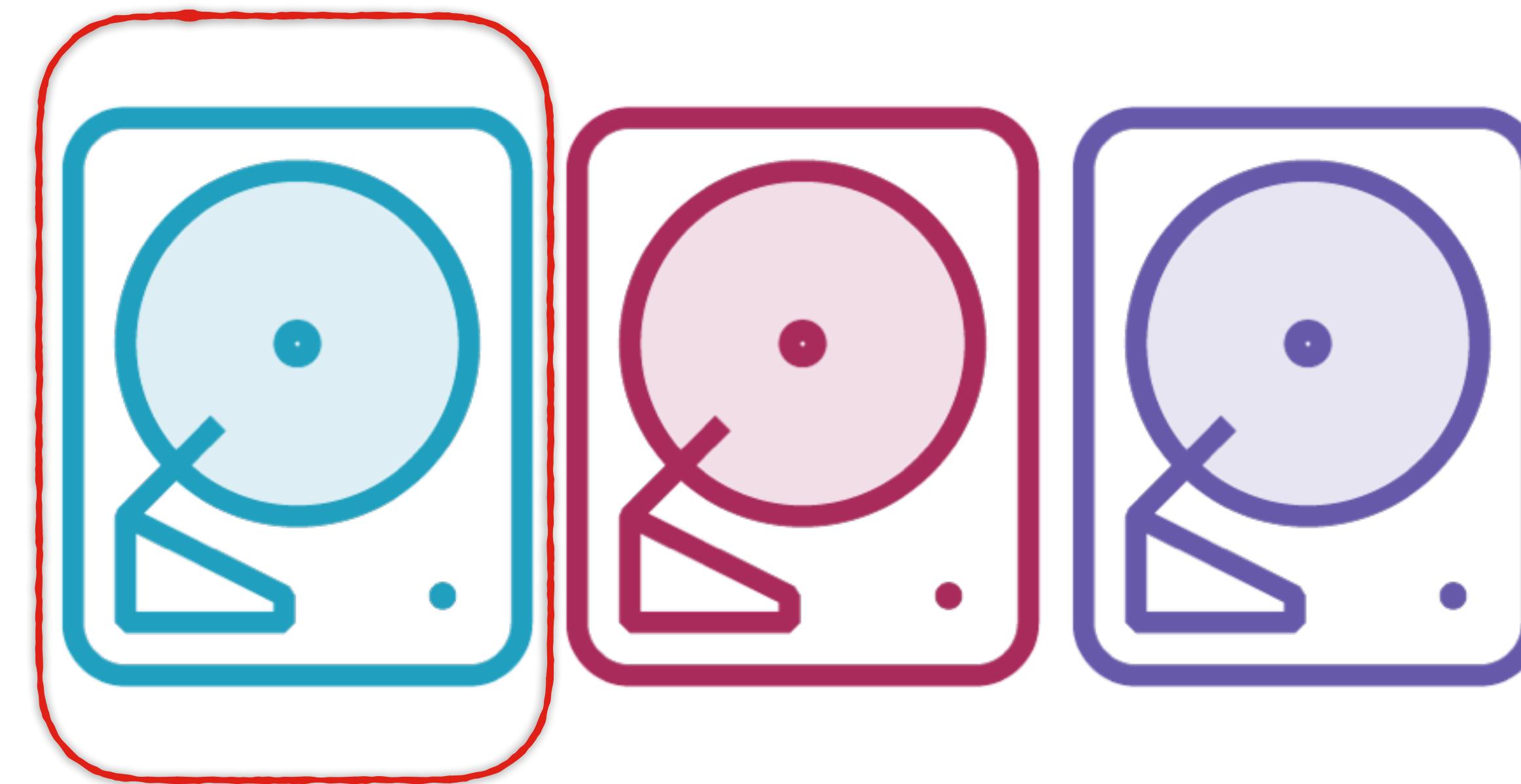
HDFS

A cluster of machines



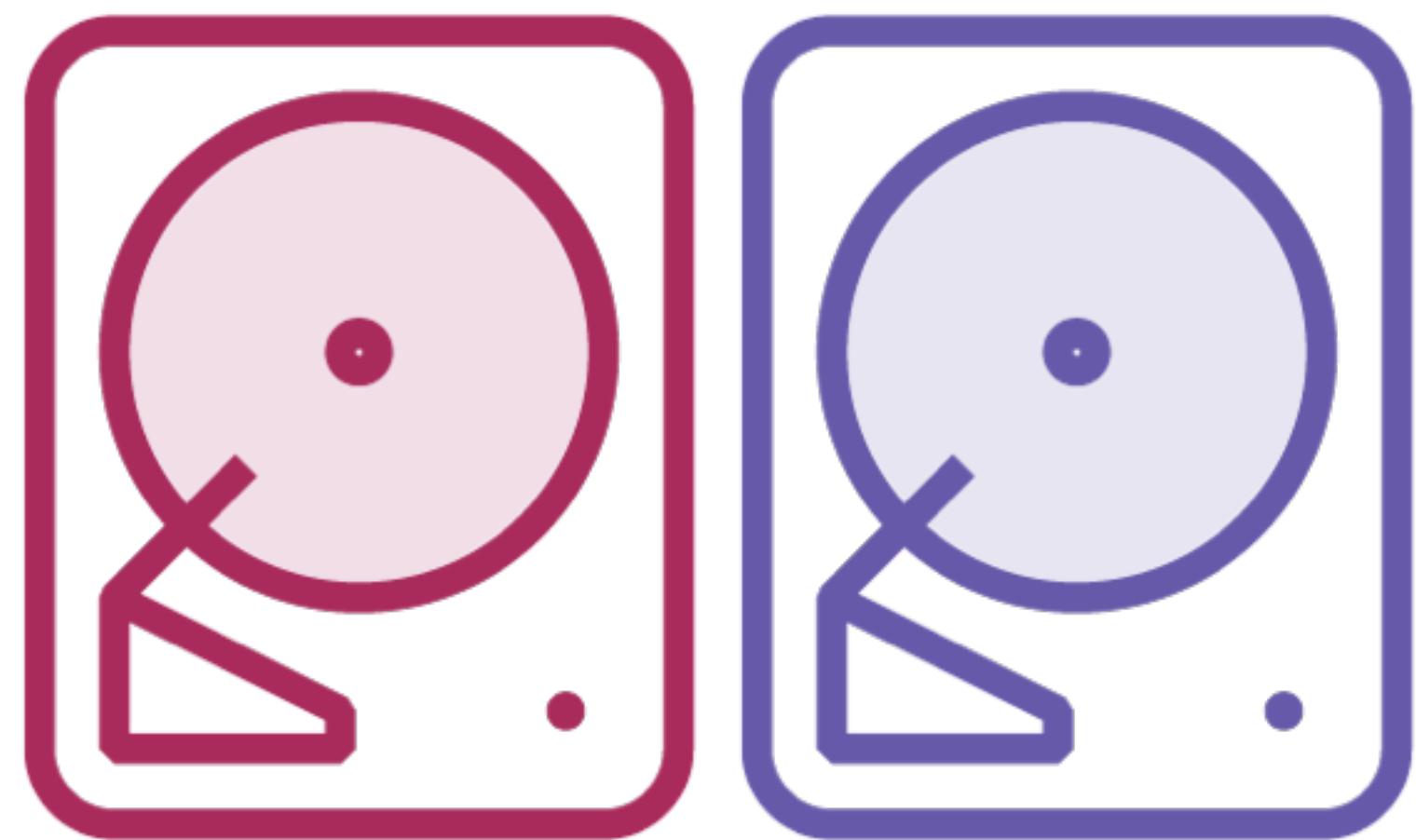
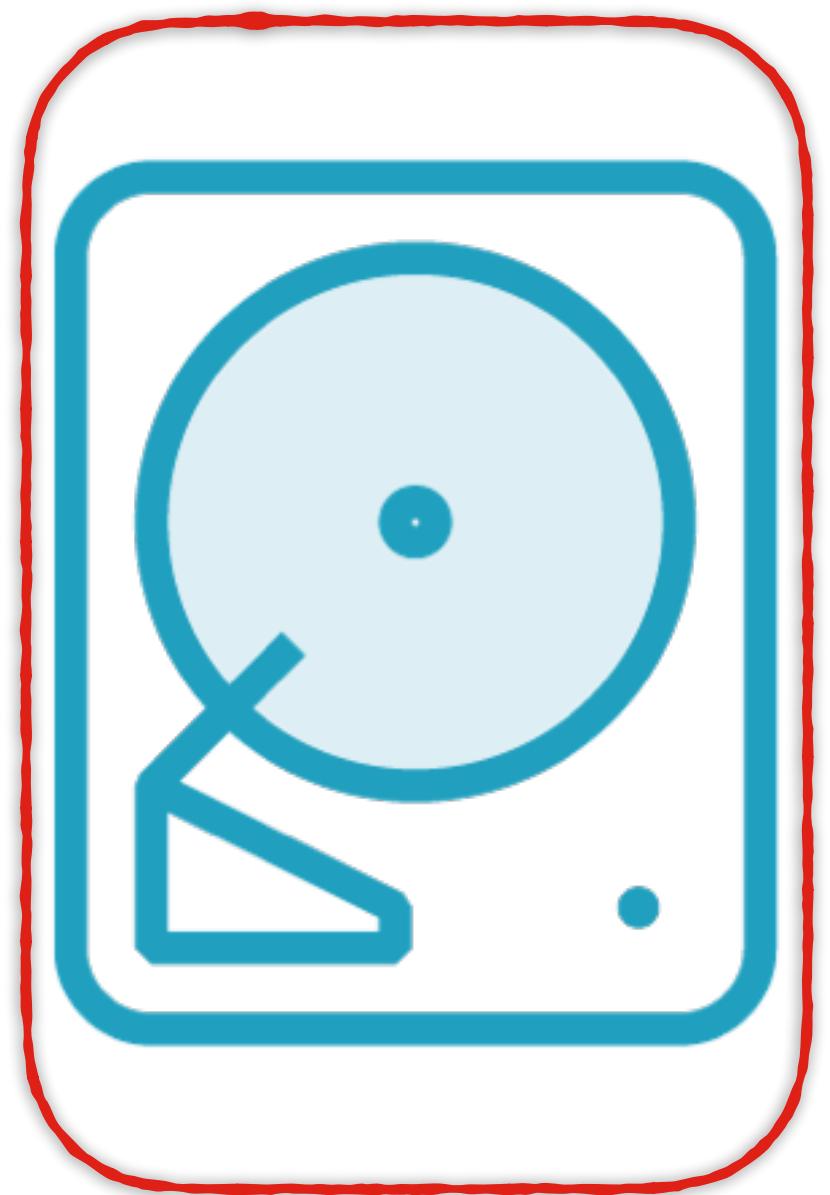
HDFS

A node in the cluster



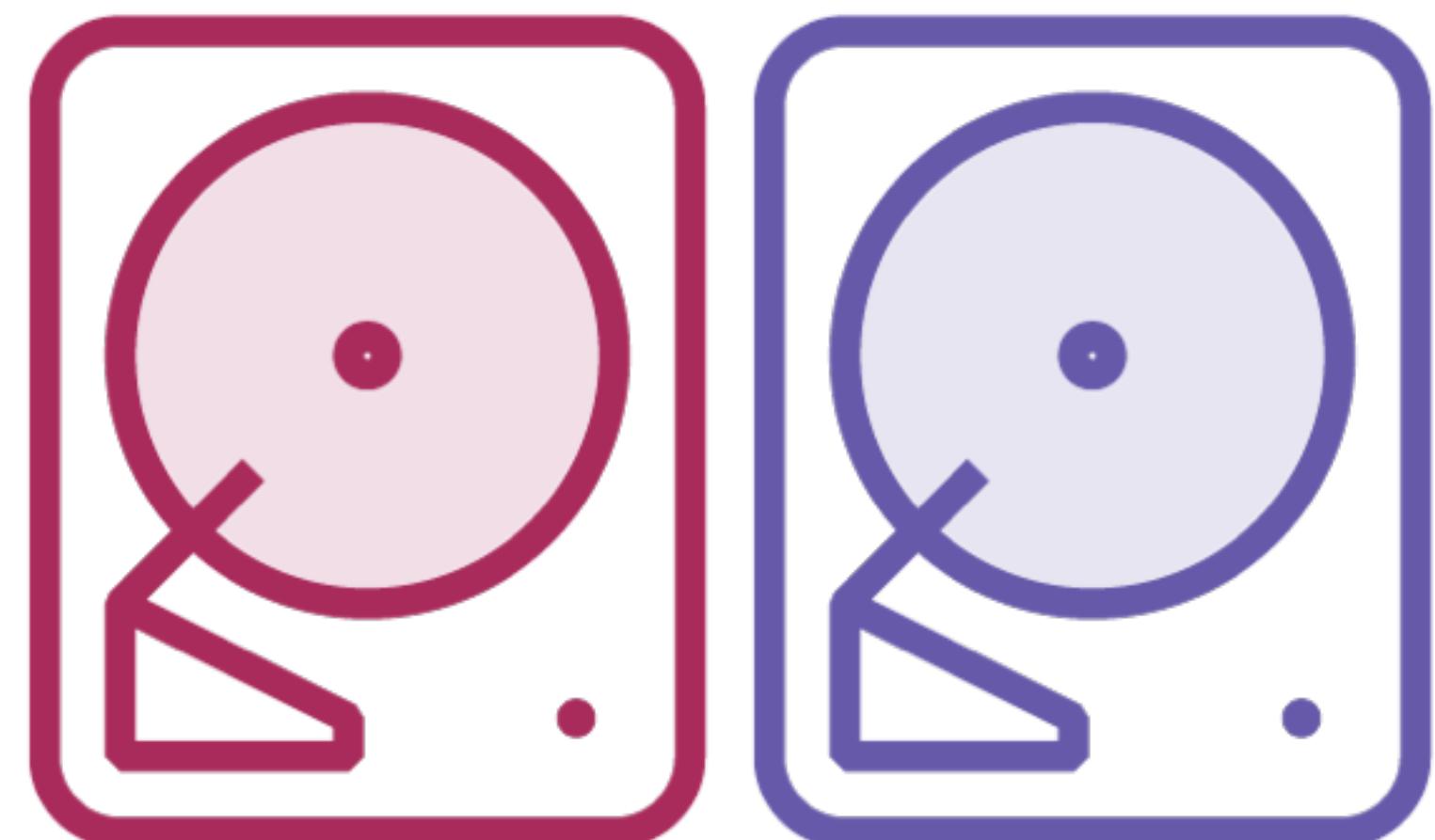
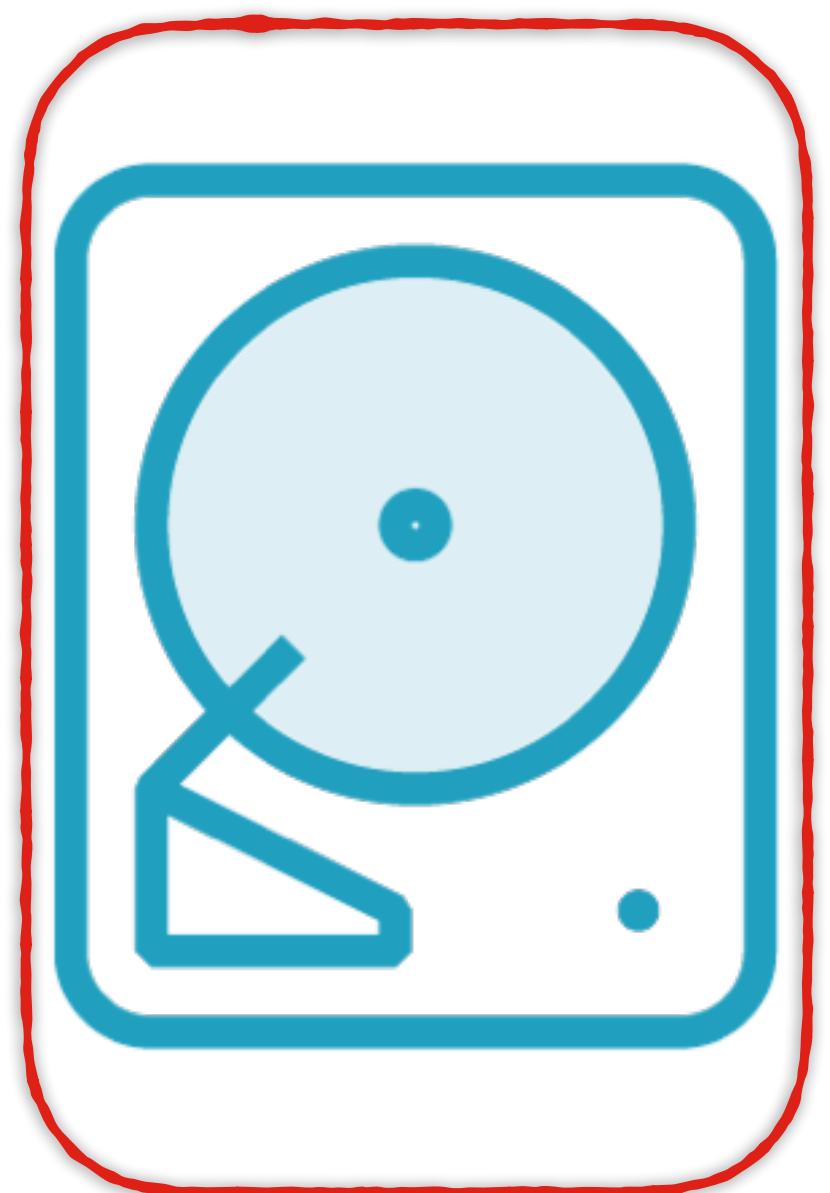
HDFS

1 node is the
master node



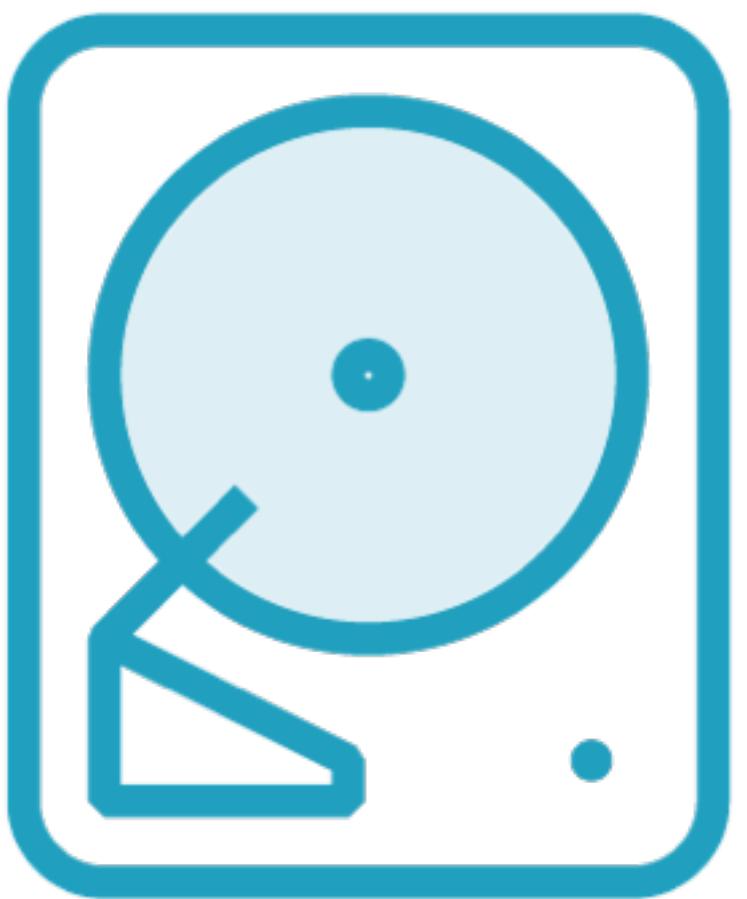
HDFS

Name node

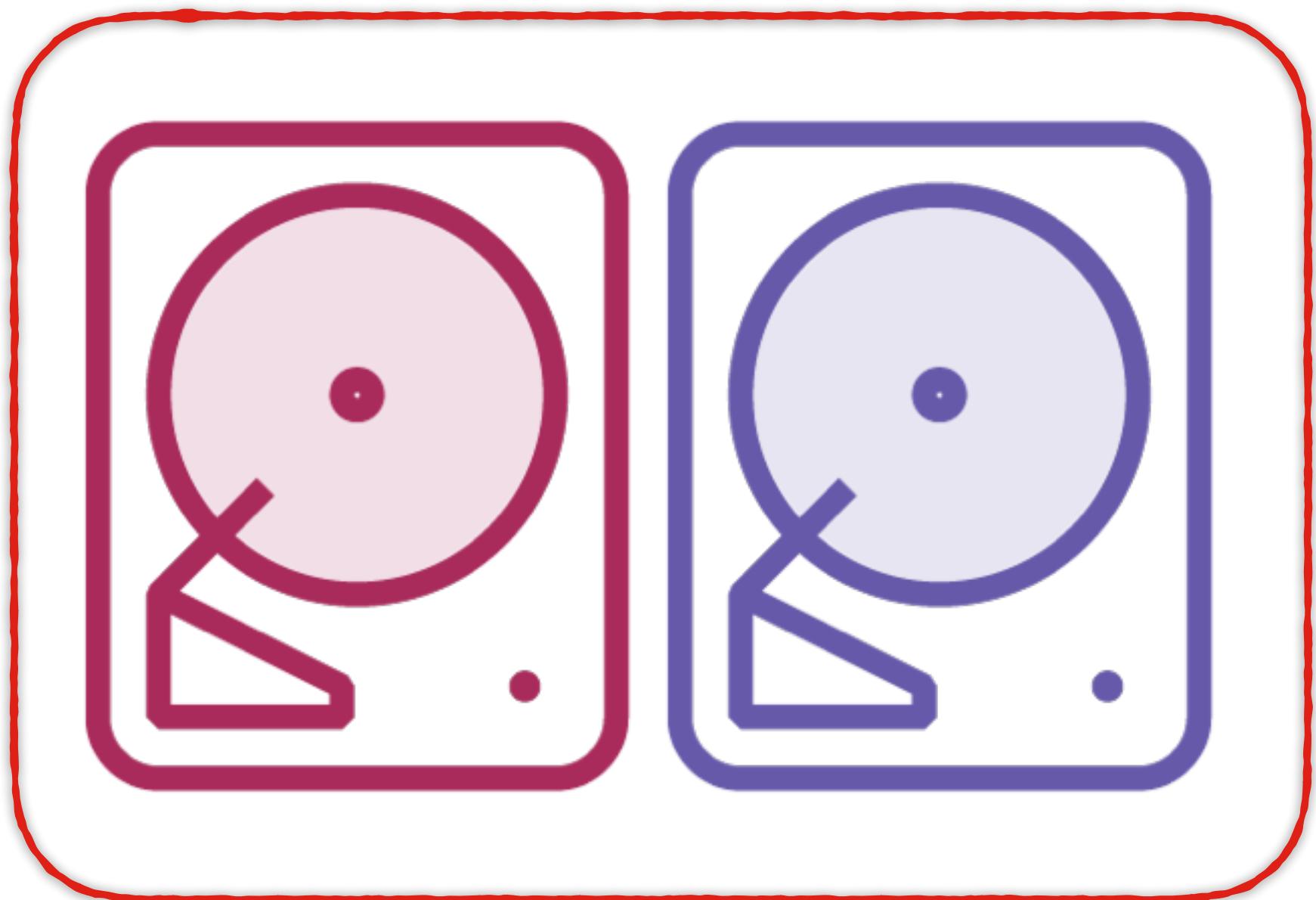


HDFS

Name node

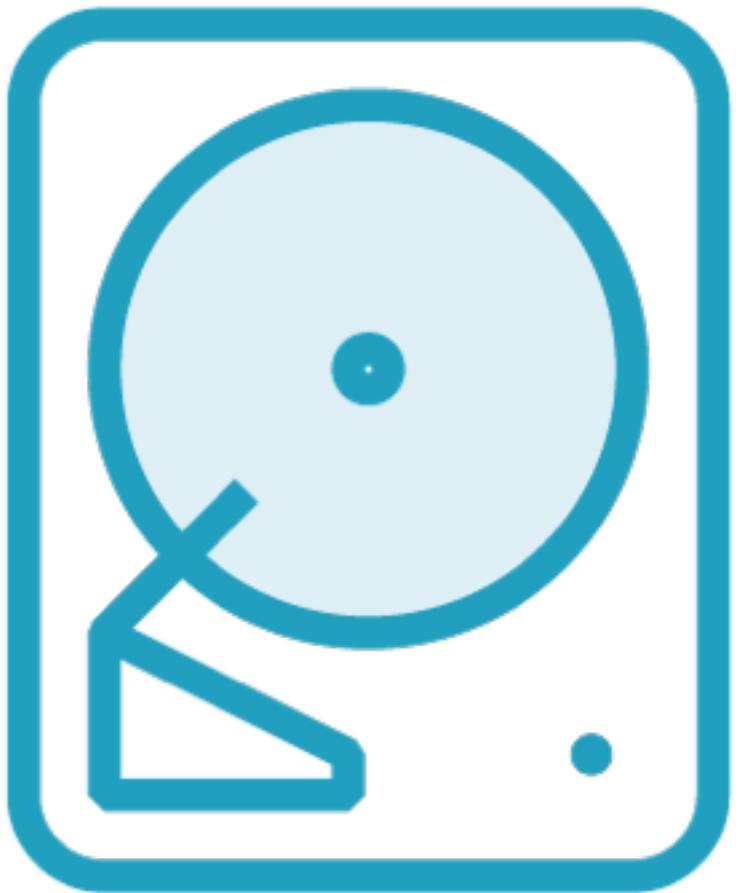


Data nodes

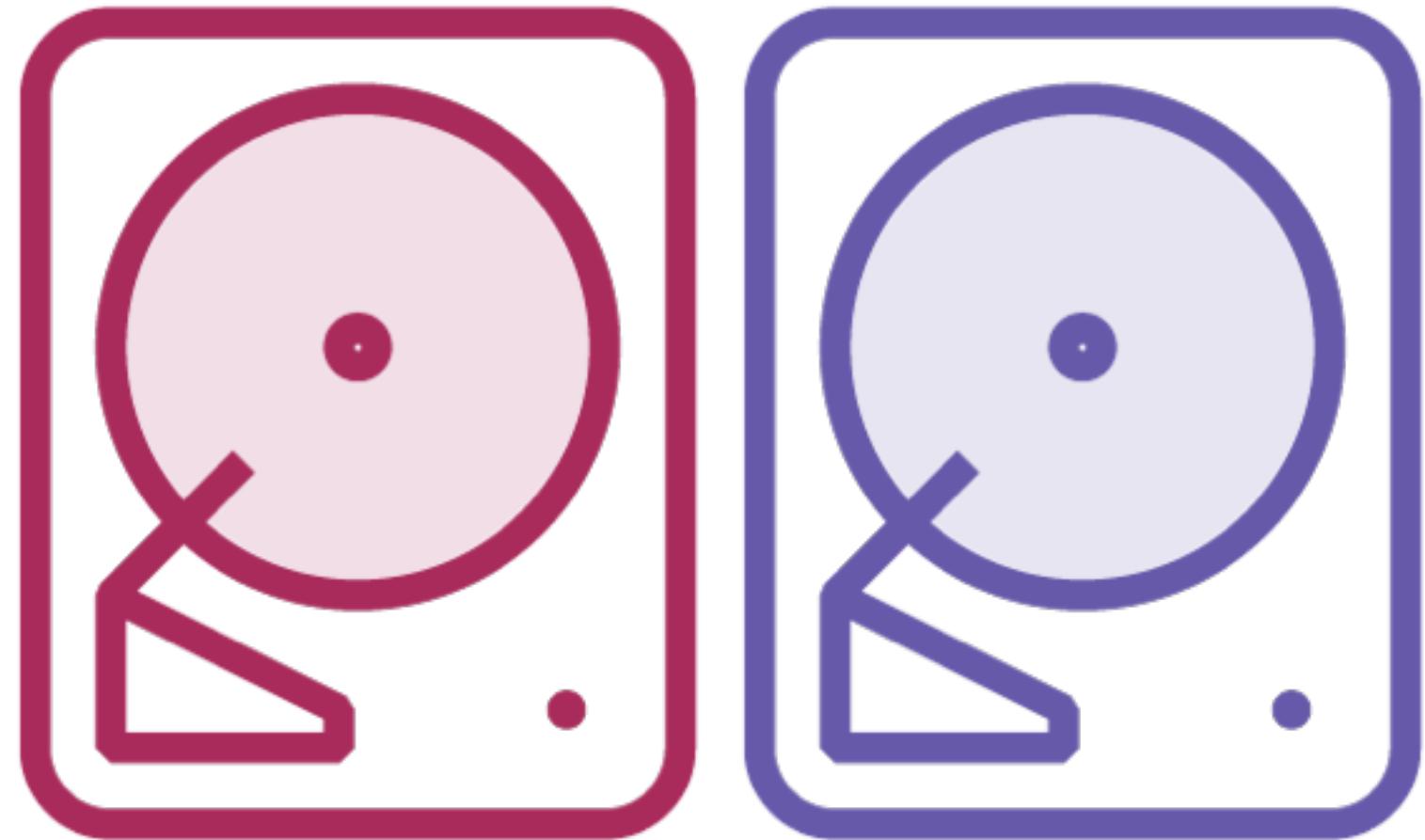


HDFS

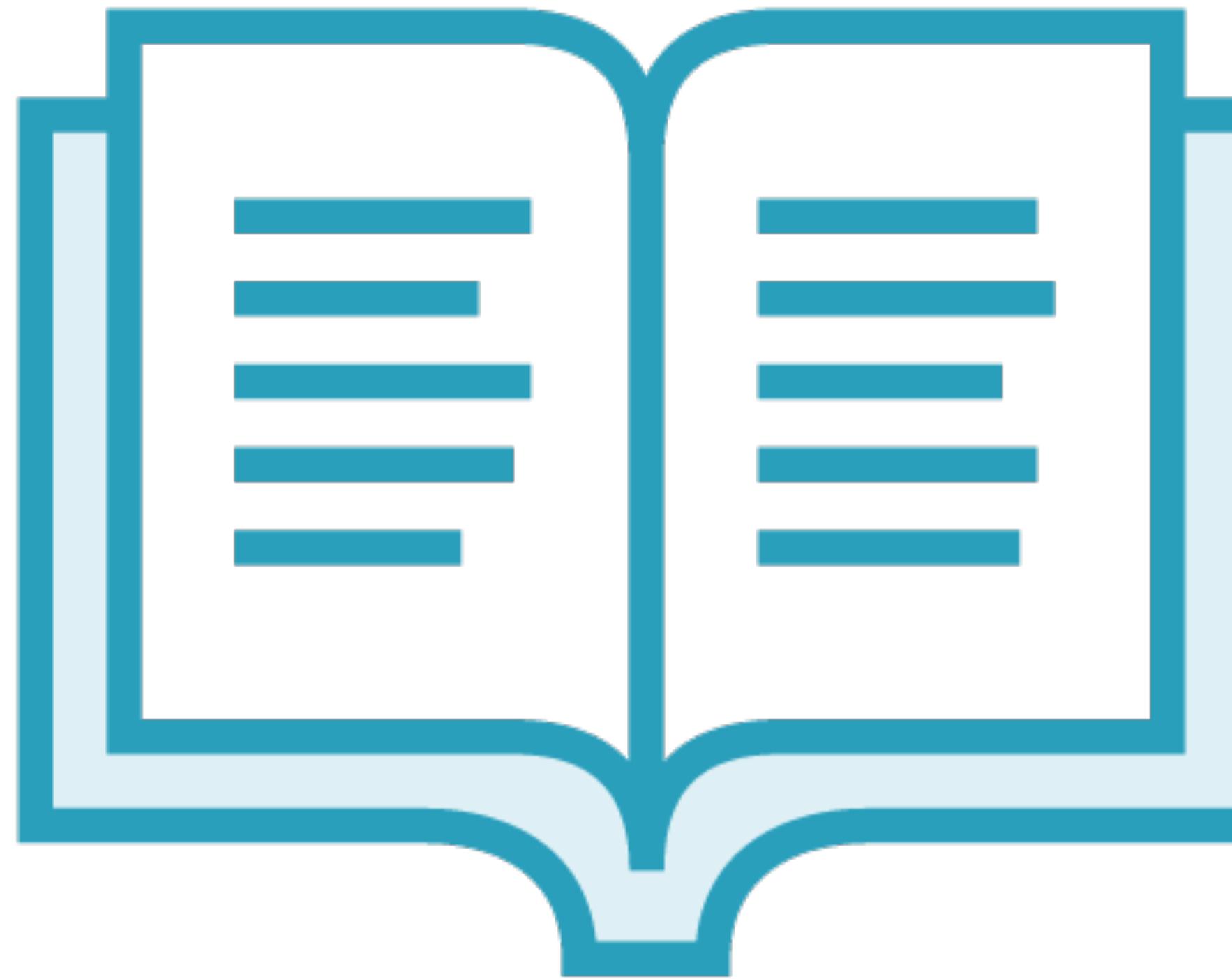
Name node



Data nodes

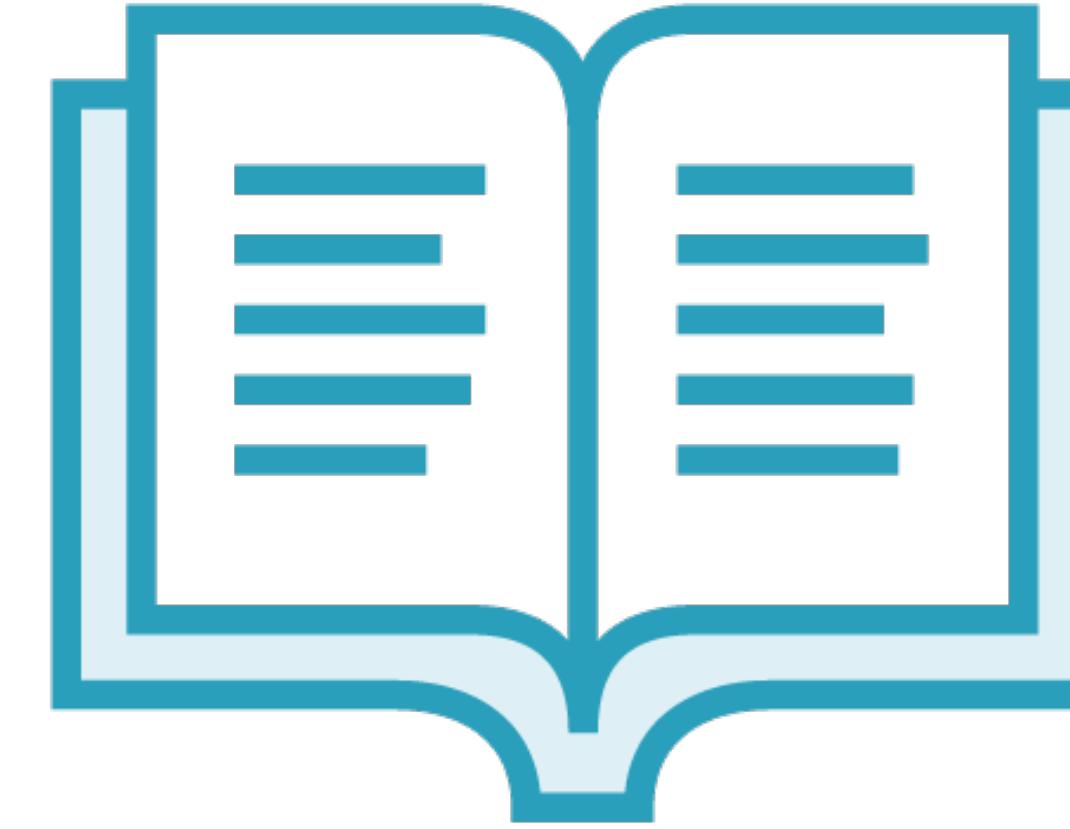
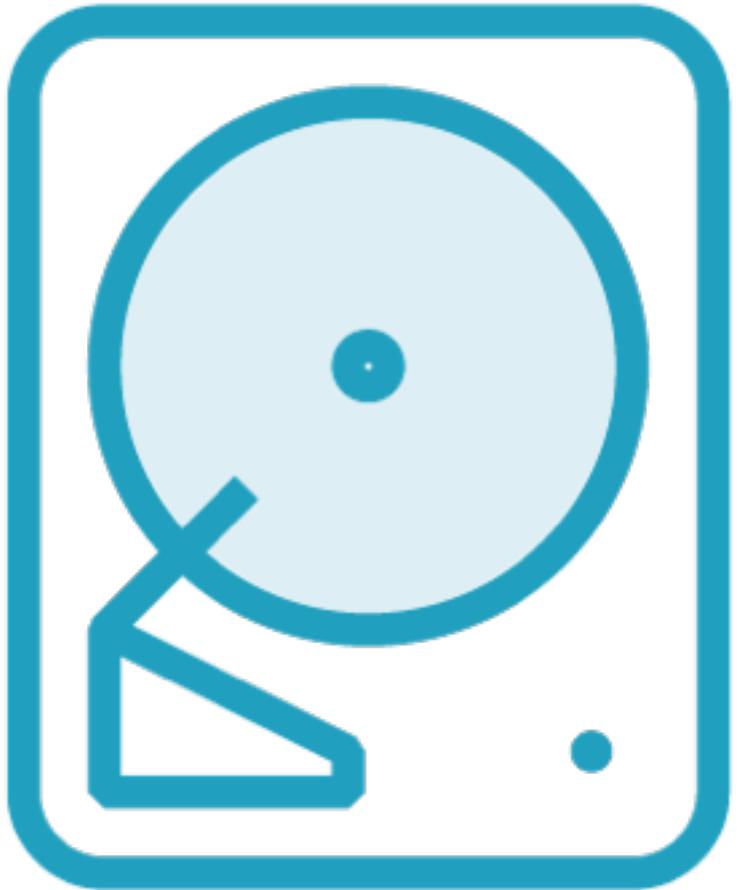


HDFS



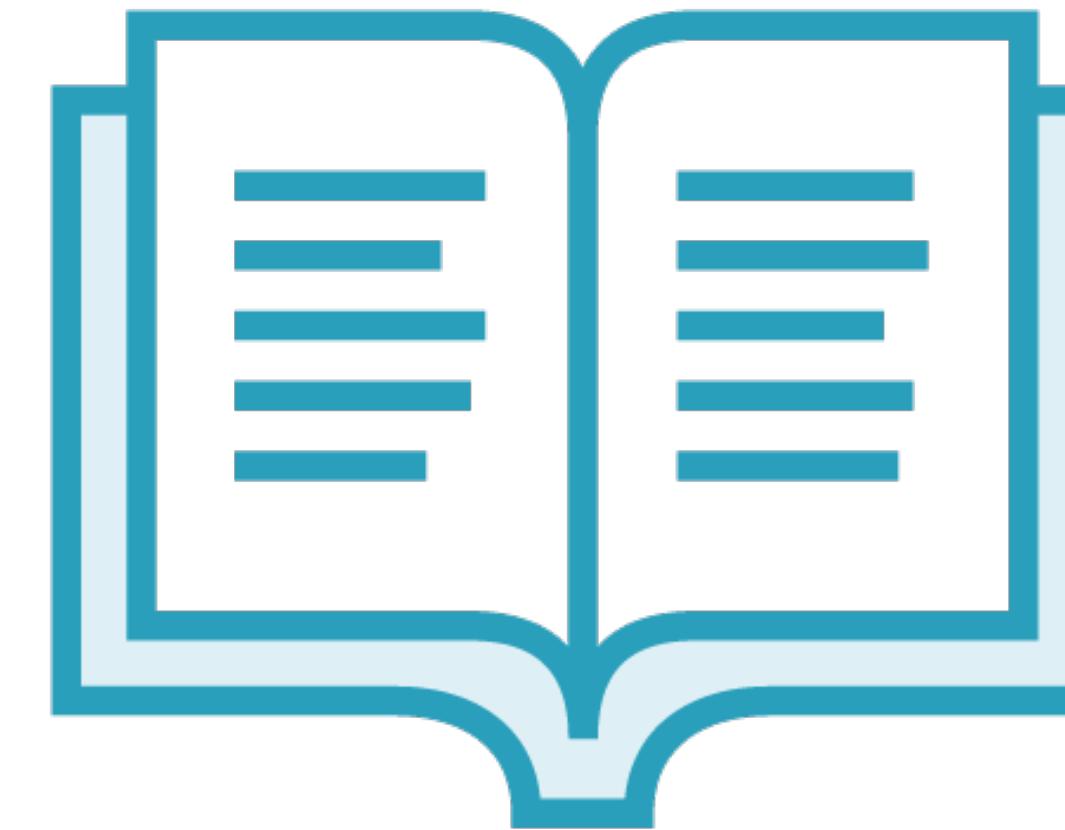
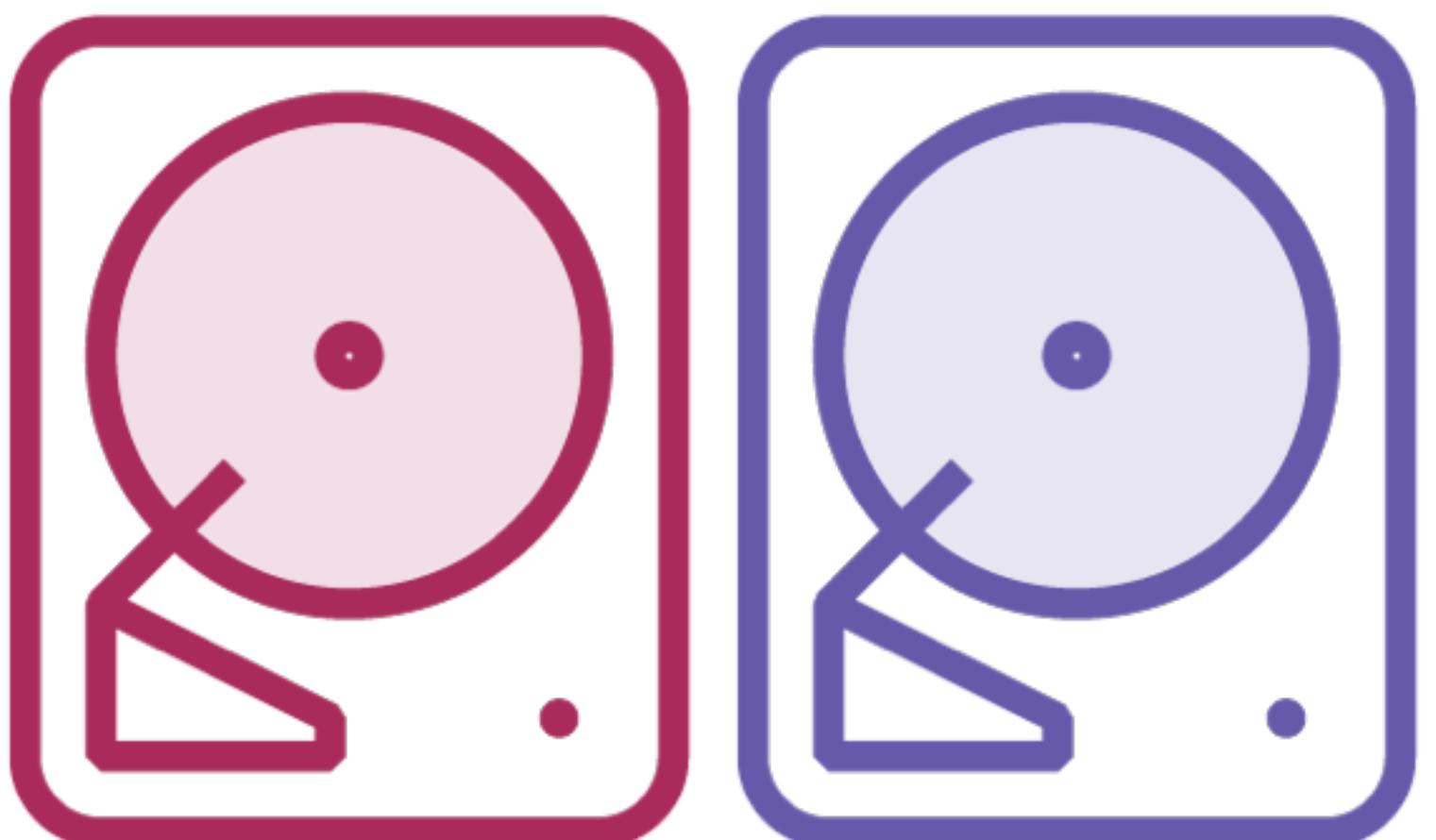
If the data in the distributed
file system is a book

Name node



The name node
is the table of
contents

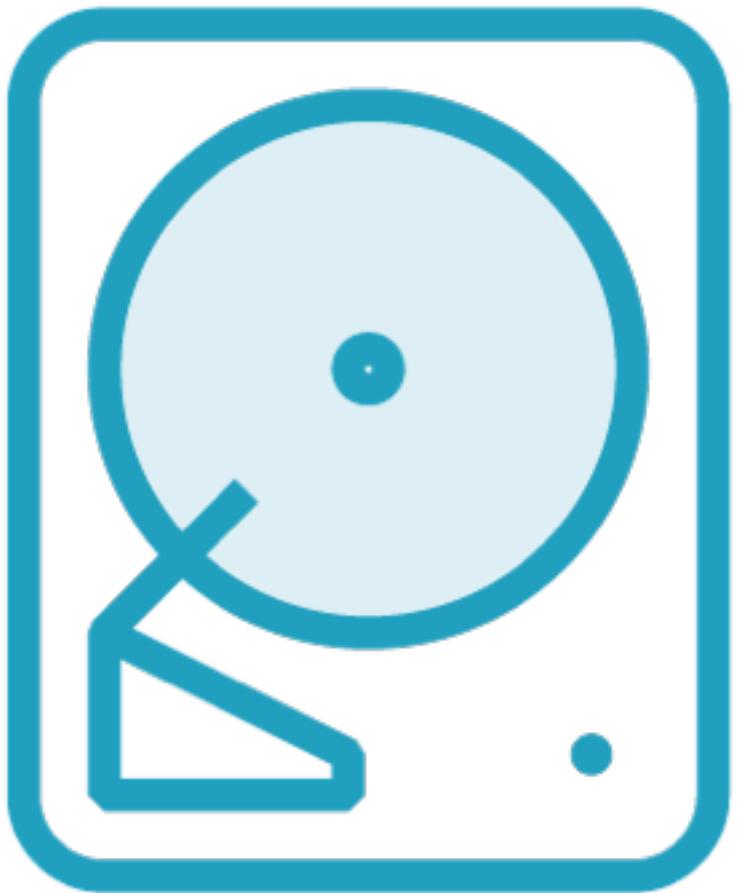
Data nodes



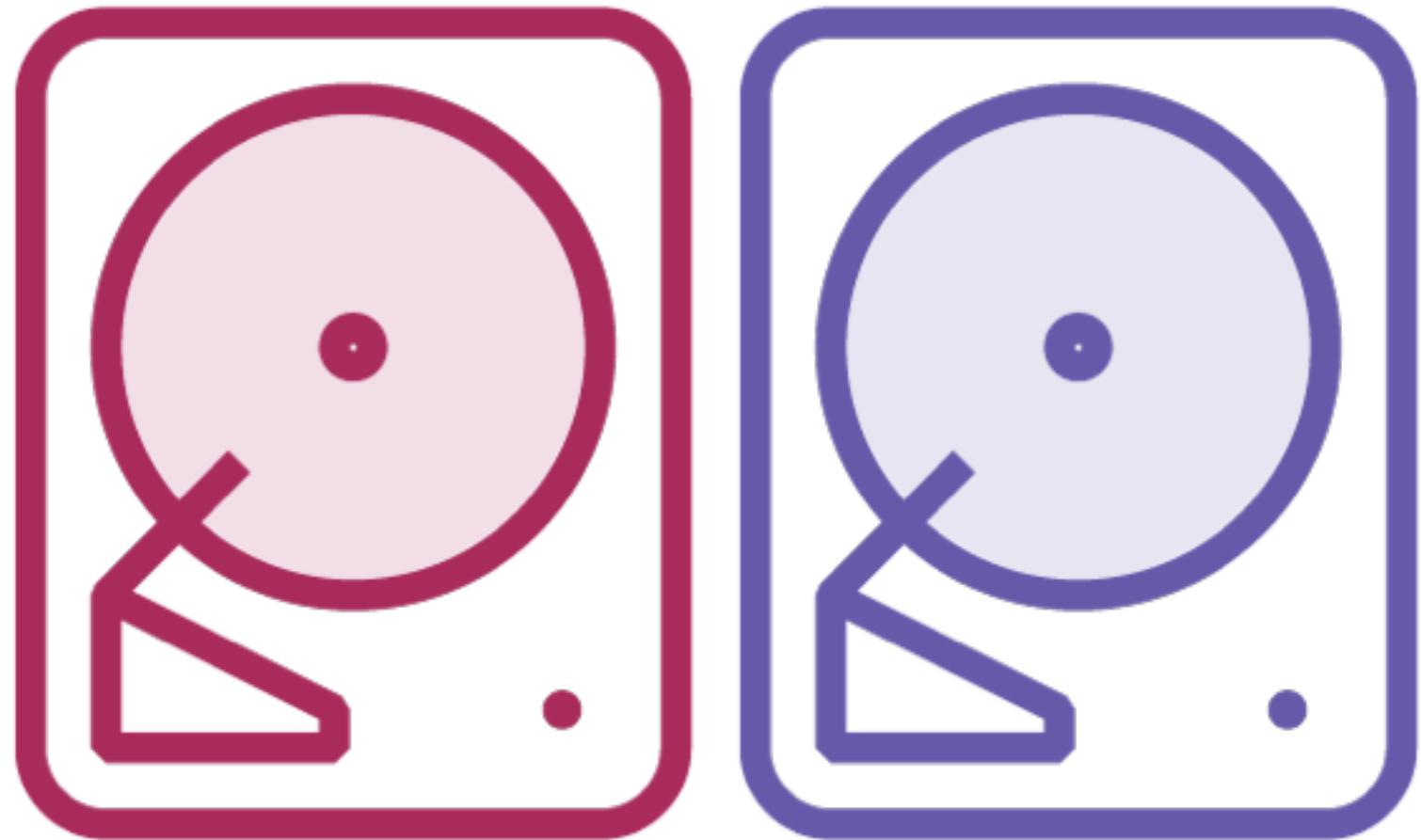
The data nodes hold the actual text in each page

HDFS

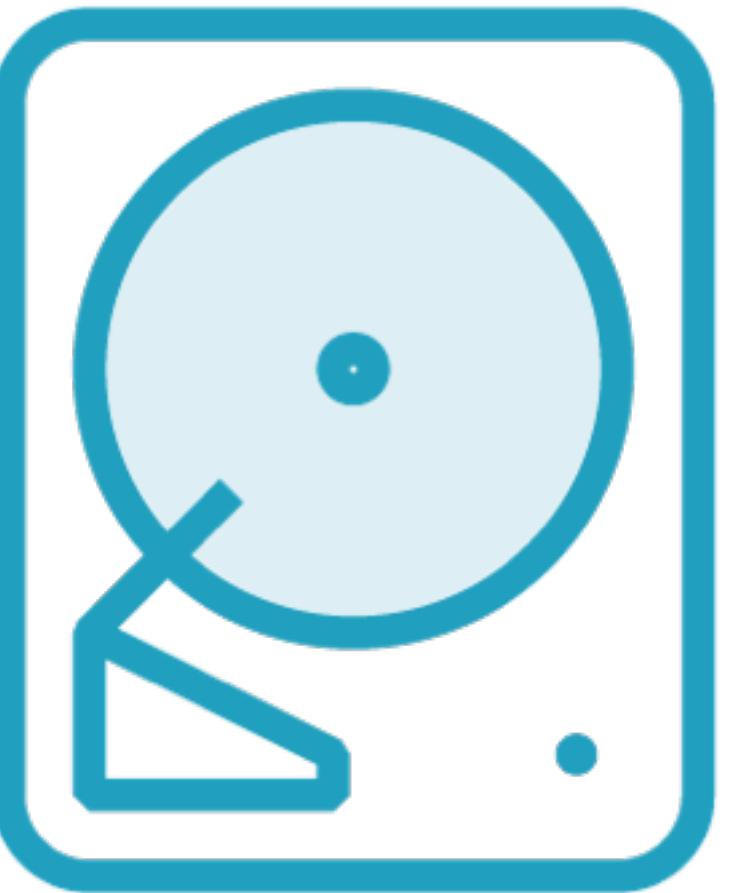
Name node



Data nodes



Name node

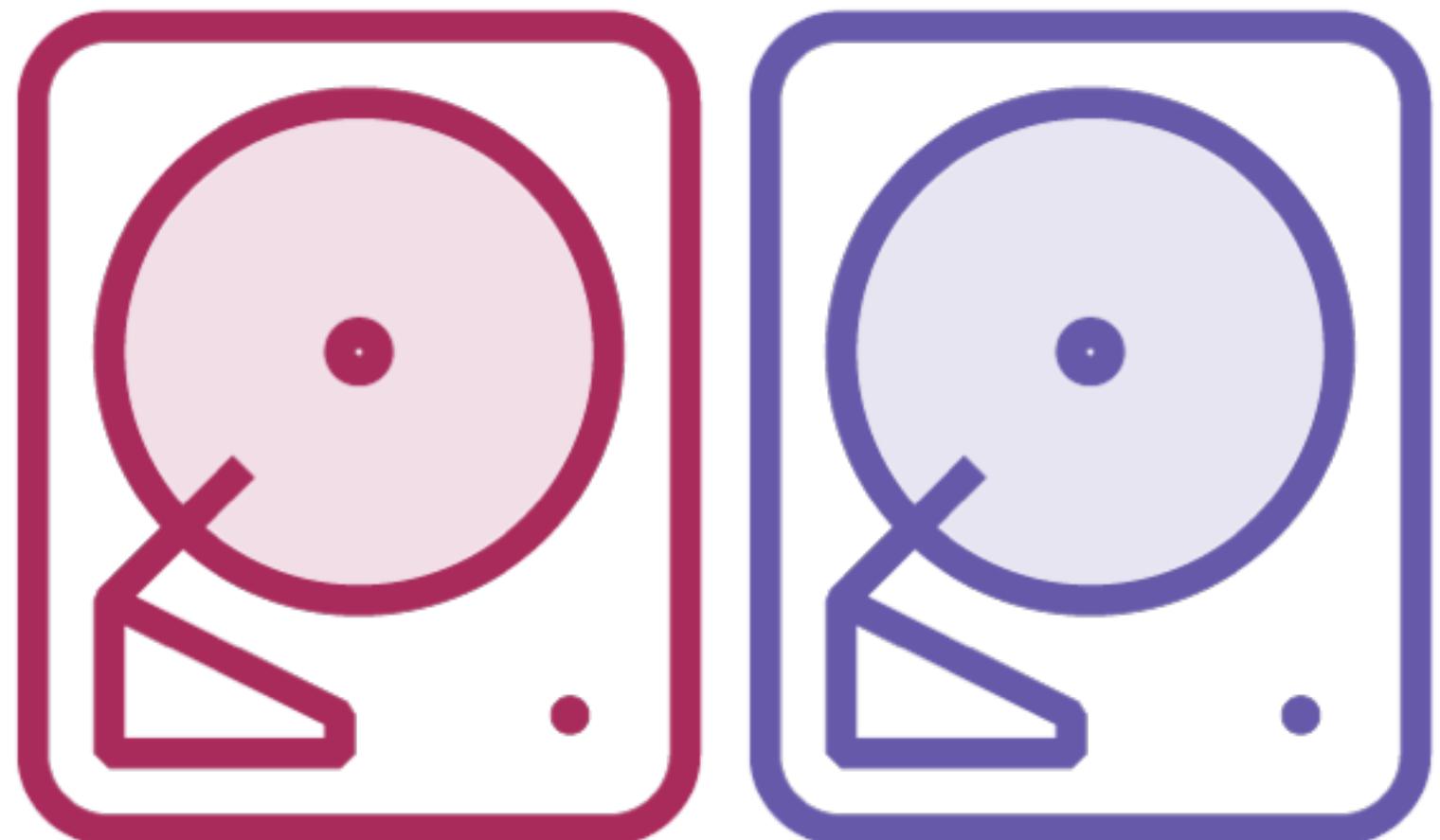


Manages the overall file system

Stores

- The directory structure
- Metadata of the files

Data nodes



**Physically stores the data
in the files**

Storing a File in HDFS

[next](#) [up](#) [previous](#) [contents](#) [index](#)
Next: Dynamic indexing Up: Index construction Previous: Single-pass in-memory indexing [Contents](#) [Index](#)

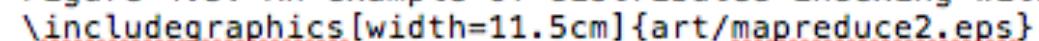
Distributed indexing

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

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

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

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

 \includegraphics[width=11.5cm]{art/mapreduce2.eps}

In general, MapReduce breaks a large computing problem into smaller parts by recasting it in terms of manipulation of key-value pairs. For indexing, a key-value pair has the form $(termID, docID)$. In distributed indexing, the mapping from terms to termIDs is also distributed and therefore more complex than in single-machine indexing. A simple solution is to maintain a (perhaps precomputed) mapping for frequent terms that is copied to all nodes and to use terms directly (instead of termIDs) for infrequent terms. We do not address this problem here and assume that all nodes share a consistent term $\$\\rightarrow\$$ termID mapping.

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

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

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

A large
text file

Storing a File in HDFS

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

Block 1

Distributed indexing

Block 2

World Wide Web for which we need large computer clusters [*]to construct any reasonably sized web index. Web search engines, therefore, use distributed indexing algorithms for index construction. The result of the construction process is a distributed index that is partitioned across several machines – either according to 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-

Block 3

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

Block 4

The map and reduce phases of MapReduce split up the computing job into chunks that standard machines can process in a short time. The various steps of MapReduce are shown in Figure 4.5 and an example of a collection consisting of two documents is shown in Figure 4.6 . First, the input data, in our case a collection of web pages, are split into \$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

Block 5

assigned by the master node on an ongoing basis: As a machine finishes processing one split, it is assigned the next one. If a machine dies or becomes a laggard due to hardware problems, the split it is working on is simply reassigned to another machine.

Figure 4.5: An example of distributed indexing with MapReduce. Adapted from Dean and Ghemawat (2004).
\includegraphics[width=11.5cm]{art/mapreduce2.eps}

Block 6

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

Block 7

and therefore more complex than in single-machine indexing. A simple solution is to maintain a (perhaps precomputed) mapping for frequent terms that is copied to all nodes and to use terms directly (instead of termID) for infrequent terms. We do not address this problem here and assume that all nodes share a consistent term \$\rightarrow\$ term ID mapping.

Block 8

The map phase of MapReduce consists of mapping splits of the input data to key-value pairs. This is the same parsing task we also

local intermediate files, the segment files (shown as \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 continuous terms or termIDs.) The term partitions are

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

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

Break the data into blocks

Storing a File in HDFS

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

Block 1

Distributed indexing

Block 2

World Wide Web for which we need large computer clusters [*]to construct any reasonably sized web index. Web search engines, therefore, use distributed indexing algorithms for index construction. The result of the construction process is a distributed index that is partitioned across several machines – either according to 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-

Block 3

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

Block 4

The map and reduce phases of MapReduce split up the computing job into chunks that standard machines can process in a short time. The various steps of MapReduce are shown in Figure 4.5 and an example of a collection consisting of two documents is shown in Figure 4.6 . First, the input data, in our case a collection of web pages, are split into \$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

Block 5

assigned by the master node on an ongoing basis: As a machine finishes processing one split, it is assigned the next one. If a machine dies or becomes a laggard due to hardware problems, the split it is working on is simply reassigned to another machine.

Figure 4.5: An example of distributed indexing with MapReduce. Adapted from Dean and Ghemawat (2004).
\includegraphics[width=11.5cm]{art/mapreduce2.eps}

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

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

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 continuous terms or termIDs.) The term partitions are

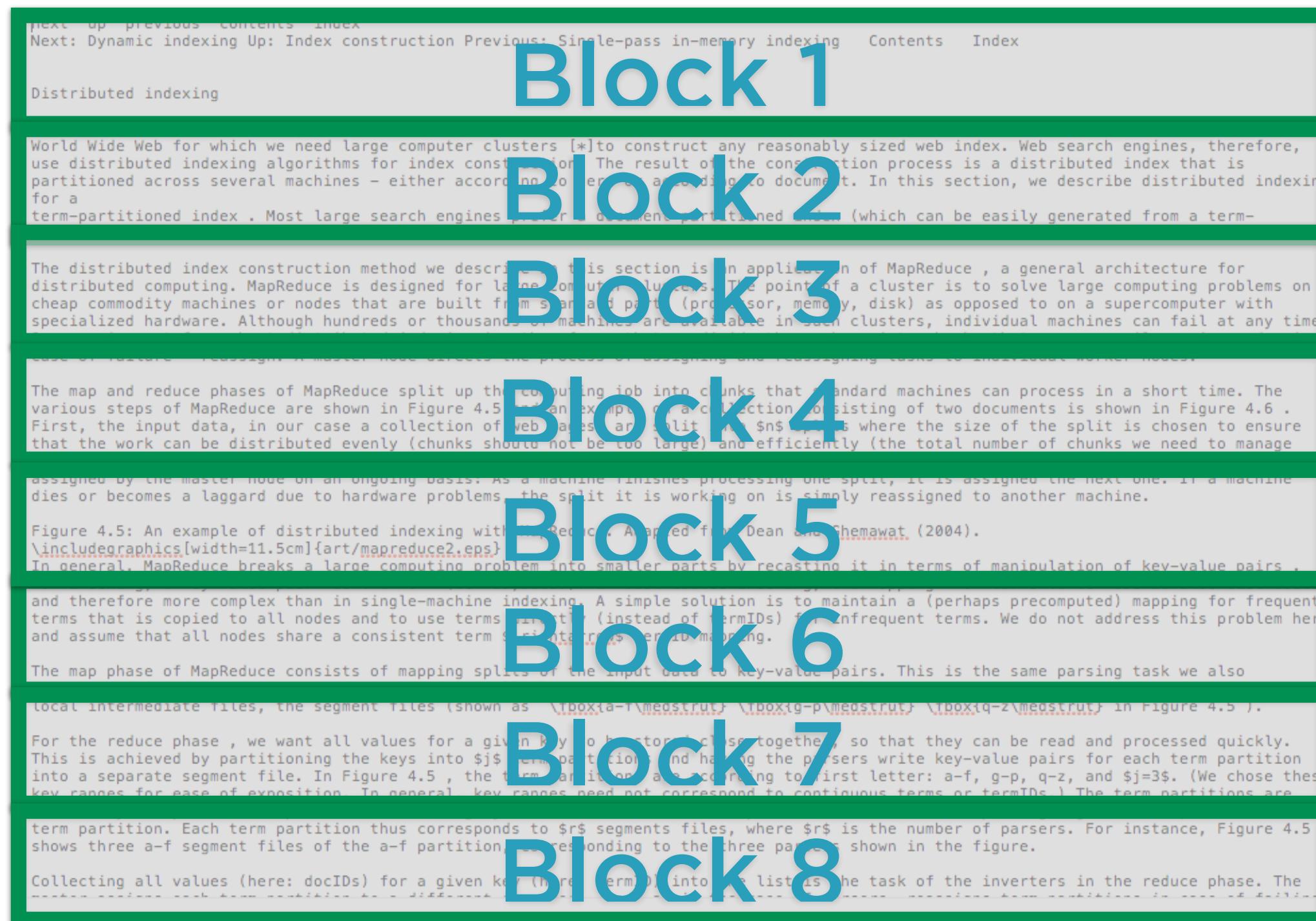
Block 8

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) is the task of the inverters in the reduce phase. The

Break the data into blocks

Storing a File in HDFS



Break the data into blocks

Different length files are treated the same way

Storage is simplified

Unit for replication and fault tolerance

Storing a File in HDFS

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

Distributed indexing

Block 1

World Wide Web for which we need large computer clusters [*]to construct any reasonably sized web index. Web search engines, therefore, use distributed indexing algorithms for index construction. The result of the construction process is a distributed index that is partitioned across several machines – either according to terms or according to document. In this section, we describe distributed indexing for a term-partitioned index. Most large search engines, for example, are partitioned (which can be easily generated from a term-partitioned index).

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 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.

Block 3

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

Figure 4.5: An example of distributed indexing with MapReduce. Adapted from Dean and Ghemawat (2004).
\includegraphics[width=11.5cm]{art/mapreduce2.eps}

In general, MapReduce breaks a large computing problem into smaller parts by recasting it in terms of manipulation of key-value pairs, and therefore more complex than in single-machine indexing. A simple solution is to maintain a (perhaps precomputed) mapping for frequent terms that is copied to all nodes and to use terms $\$t$ as keys (instead of termIDs) for infrequent terms. We do not address this problem here and assume that all nodes share a consistent term $\$t$ to $\$r$ key-value 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

local intermediate files, the segment files (shown as \100X(a-\\$m)eos100) \100X(g-\\$m)eos100 \100X(q-\\$m)eos100 in Figure 4.5).

For the reduce phase, we want all values for a given key to be stored close together so that they can be read and processed quickly. This is achieved by partitioning the keys into $\$j$ partitions and having the parsers write key-value pairs for each term partition into a separate segment file. In Figure 4.5, the term partitions are corresponding 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

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 a list is the task of the inverters in the reduce phase. The

Block 4

Block 5

Block 6

Block 7

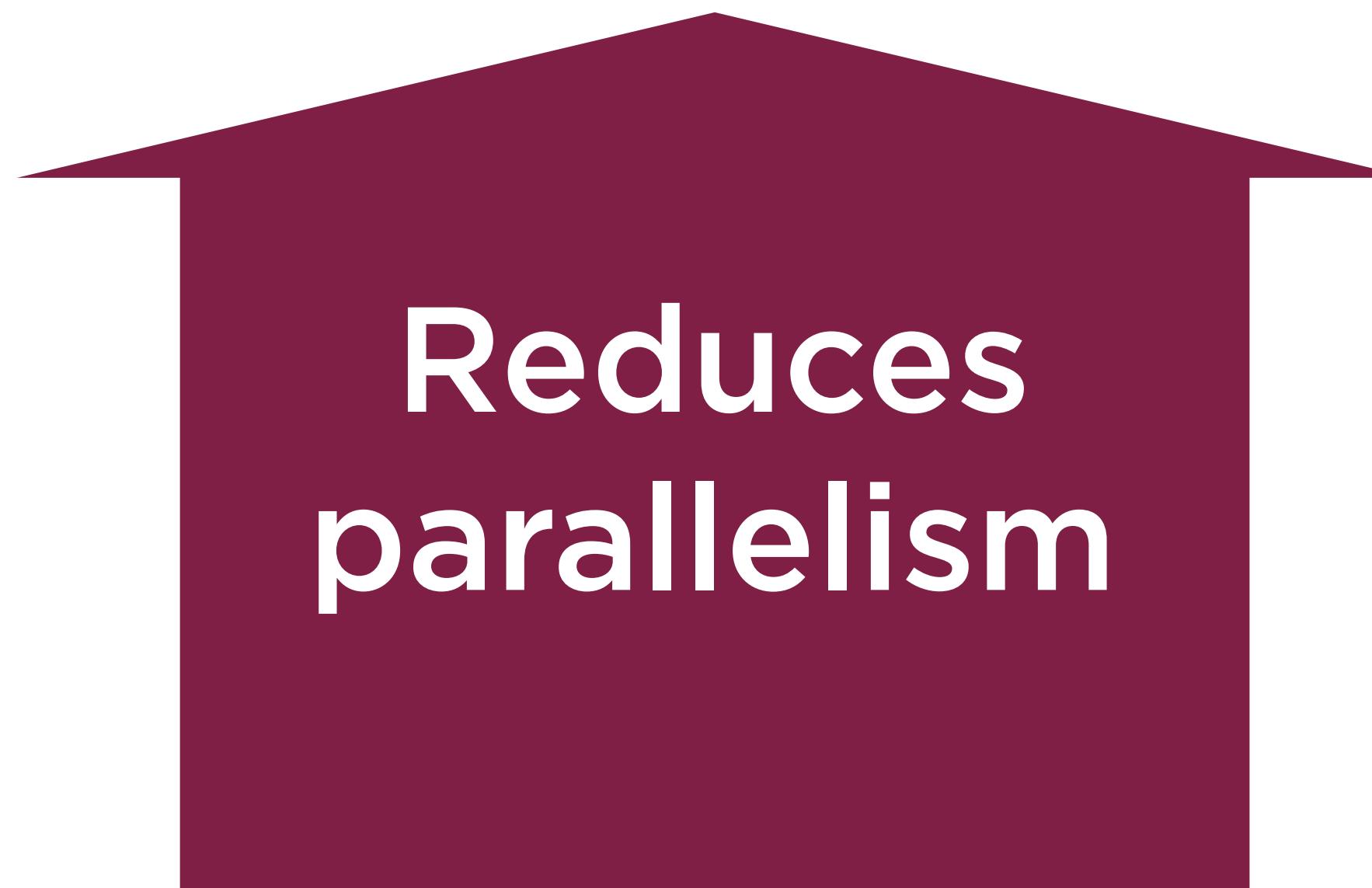
Block 8

The blocks are
of size 128 MB

Storing a File in HDFS

size 128 MB

Block size is a trade off



Storing a File in HDFS

size 128 MB

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

Storing a File in HDFS

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

Distributed indexing

Block 1

World Wide Web for which we need large computer clusters [*]to construct any reasonably sized web index. Web search engines, therefore, use distributed indexing algorithms for index construction. The result of the construction process is a distributed index that is partitioned across several machines – either according to terms or according to document. In this section, we describe distributed indexing for a term-partitioned index. Most large search engines, for example, use term-partitioned indexes (which can be easily generated from a term-partitioned index).

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 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.

Block 3

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

Figure 4.5: An example of distributed indexing with MapReduce. Adapted from Dean and Ghemawat (2004).
\includegraphics[width=11.5cm]{art/mapreduce2.eps}

In general, MapReduce breaks a large computing problem into smaller parts by recasting it in terms of manipulation of key-value pairs, and therefore more complex than in single-machine indexing. A simple solution is to maintain a (perhaps precomputed) mapping for frequent terms that is copied to all nodes and to use terms $\$t$ as keys (instead of termIDs) for infrequent terms. We do not address this problem here and assume that all nodes share a consistent term $\$t$ to $\$r$ key-value 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

local intermediate files, the segment files (shown as \100X(a-\\$m)eos100) \100X(g-\\$m)eos100 \100X(q-\\$m)eos100 in Figure 4.5).

For the reduce phase, we want all values for a given key to be stored close together so that they can be read and processed quickly. This is achieved by partitioning the keys into $\$j$ partitions and having the parsers write key-value pairs for each term partition into a separate segment file. In Figure 4.5, the term partitions are corresponding 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

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 a list is the task of the inverters in the reduce phase. The

Block 4

Block 5

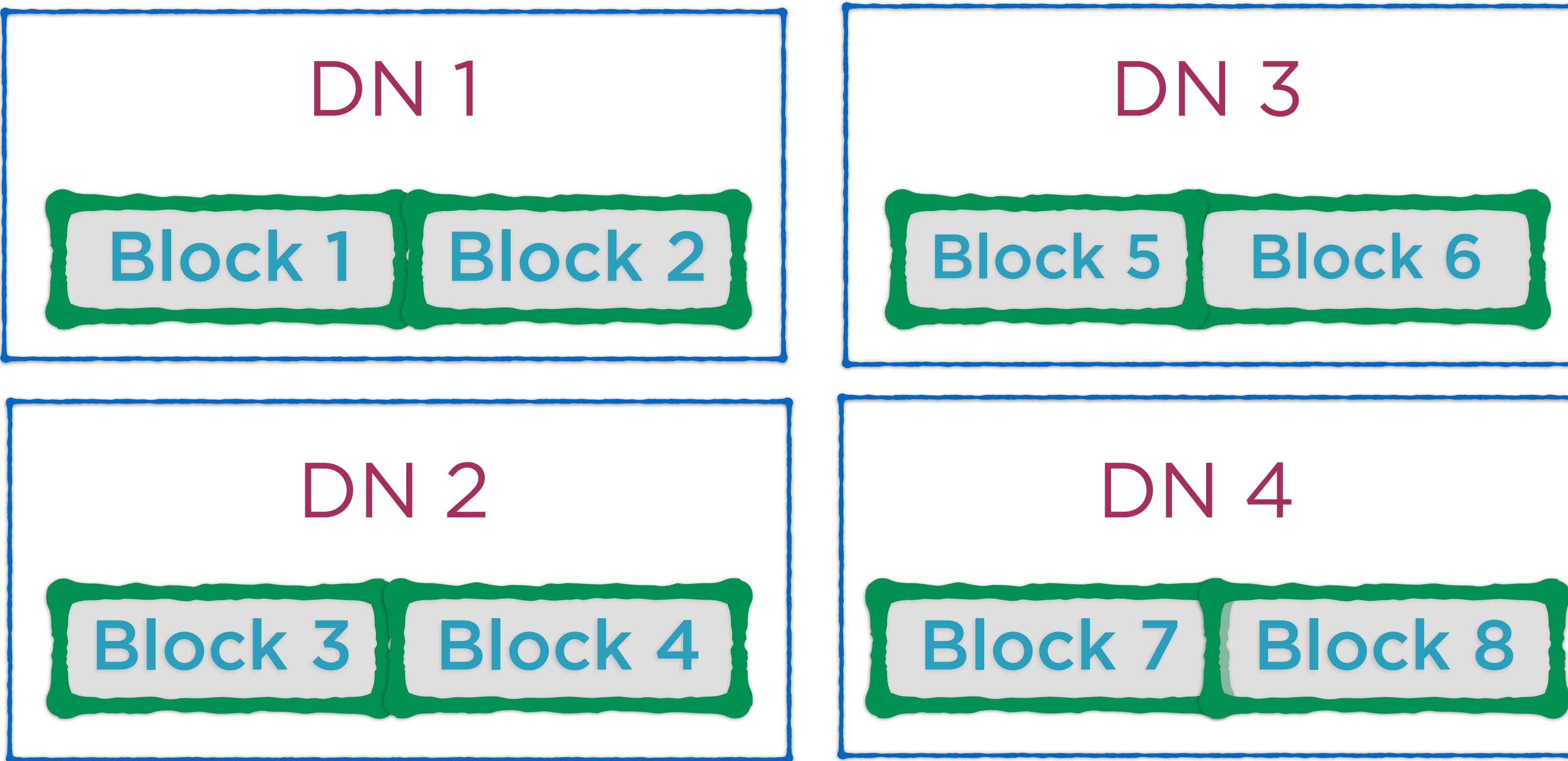
Block 6

Block 7

Block 8

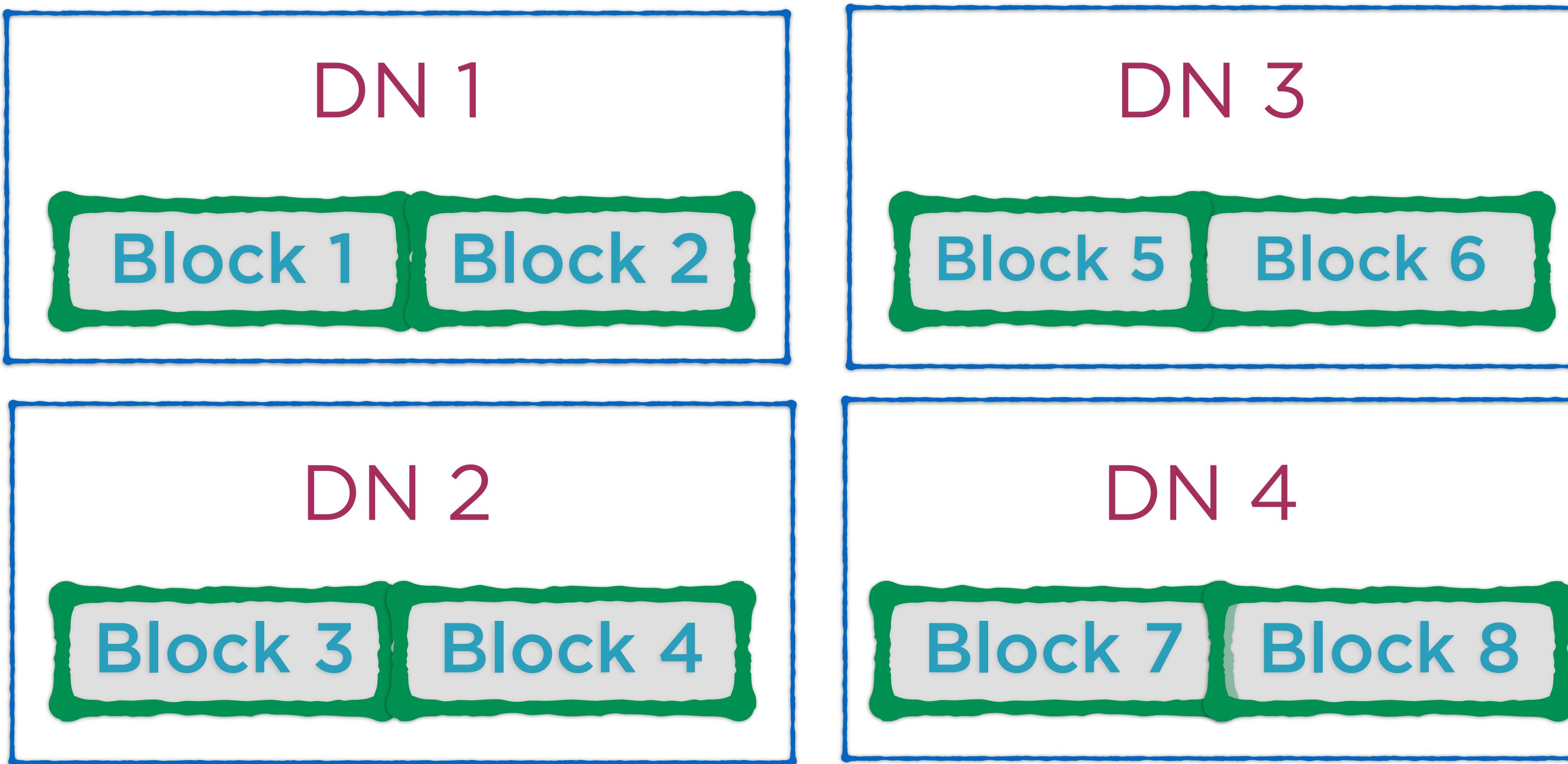
Store the blocks
across the data
nodes

Storing a File in HDFS



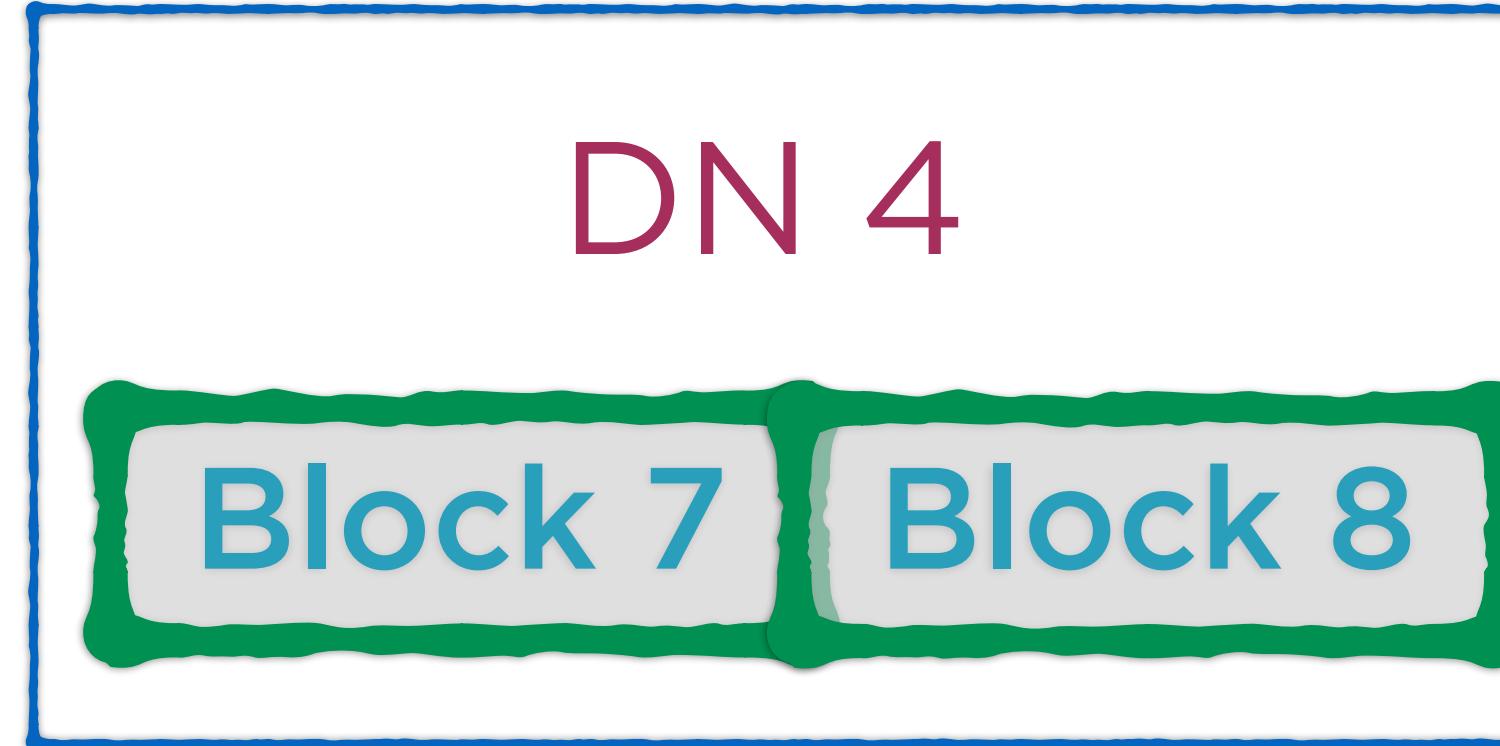
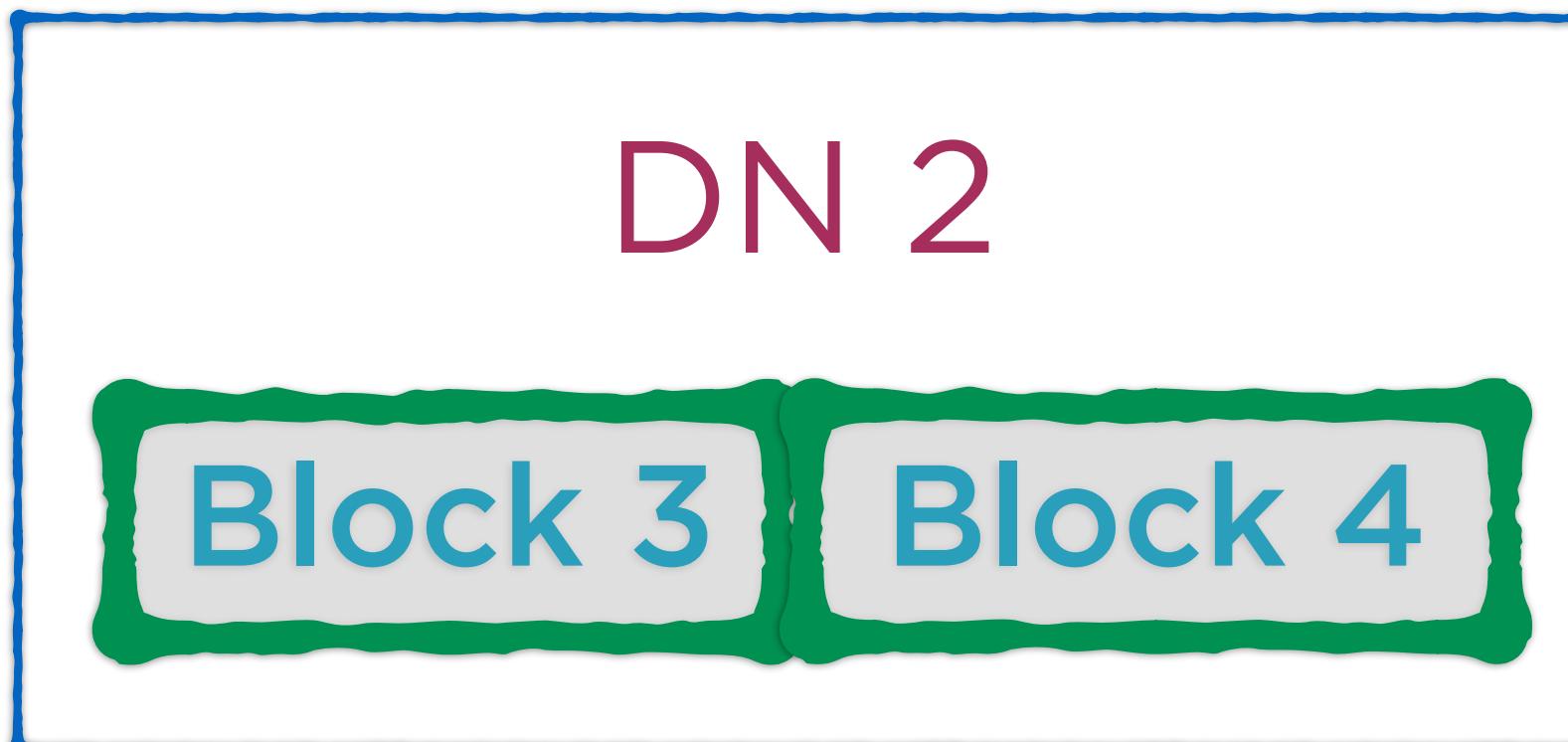
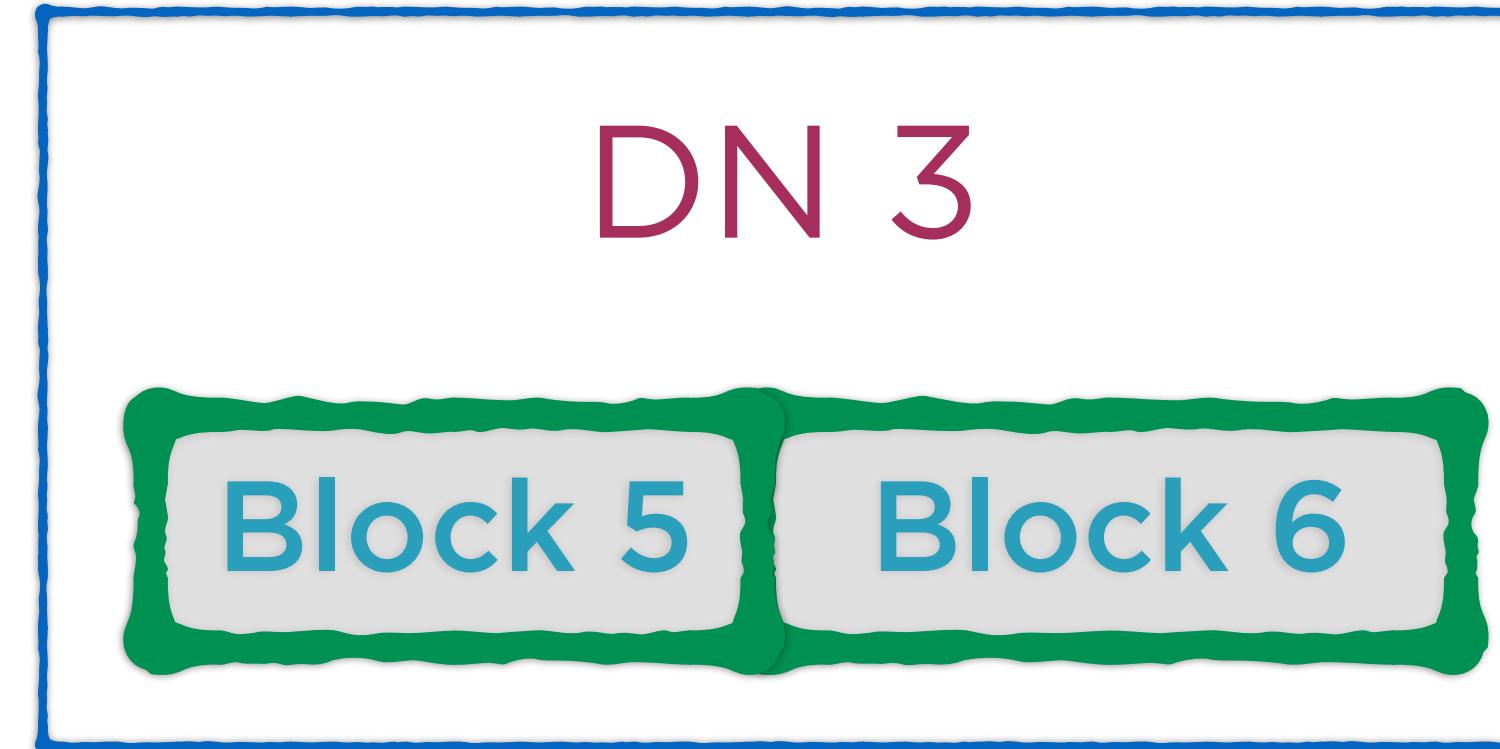
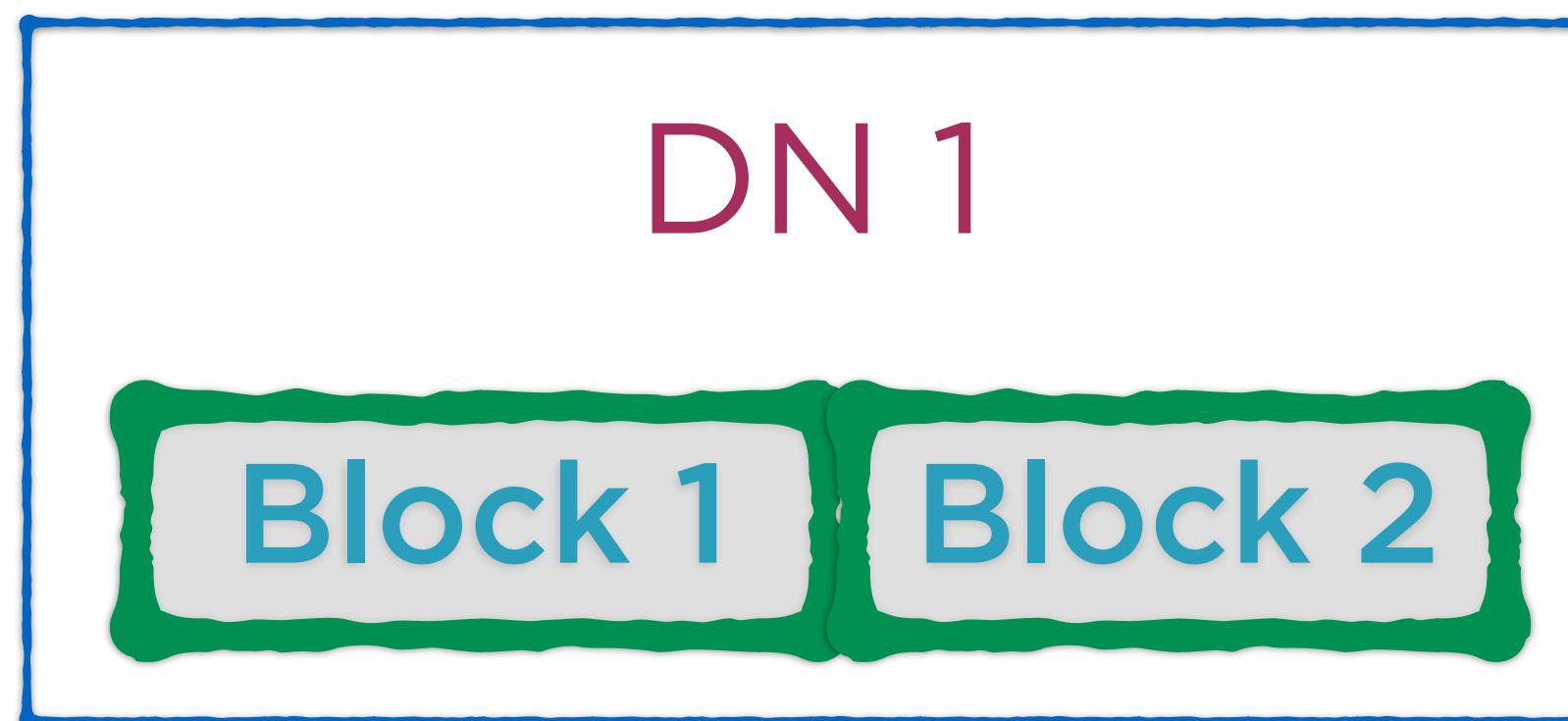
Each node contains a partition or a split of data

Storing a File in HDFS



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

Storing a File in HDFS



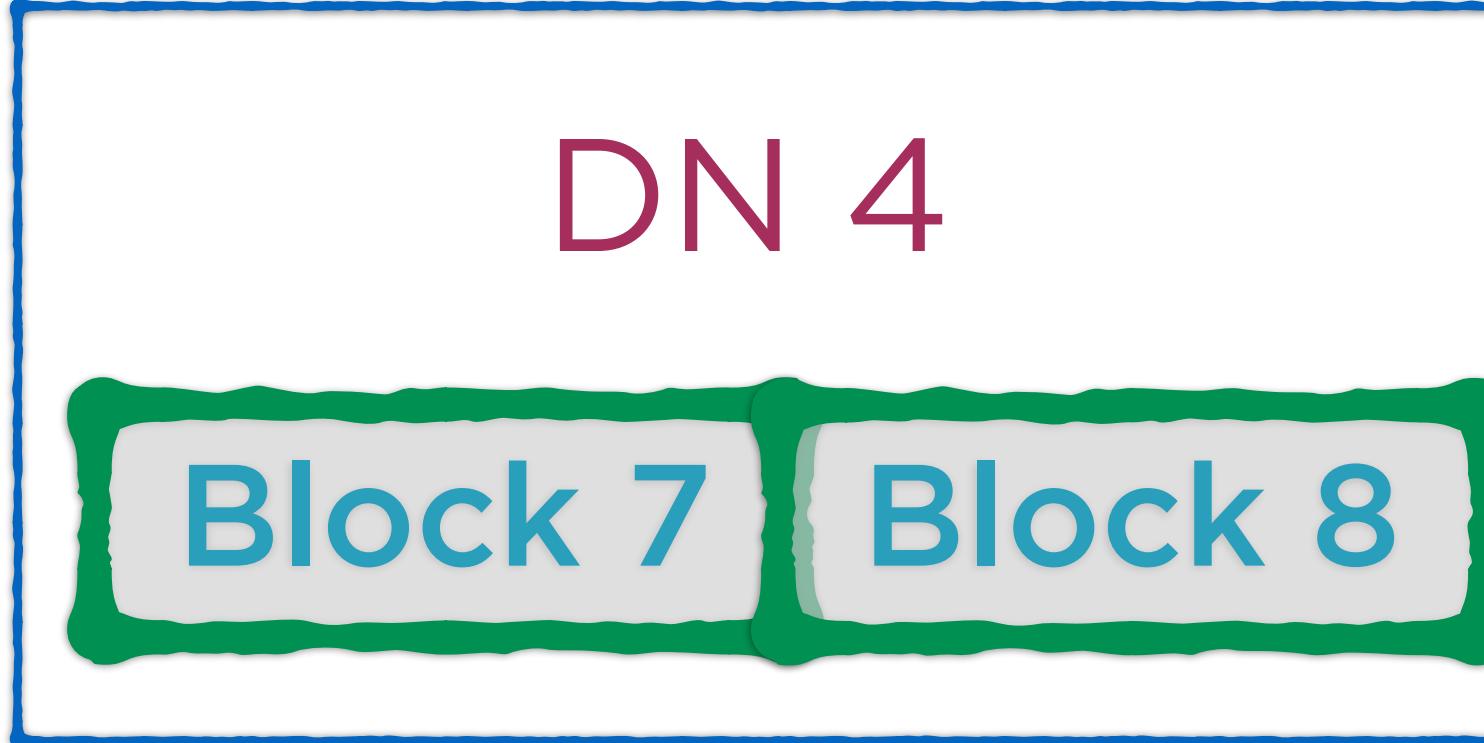
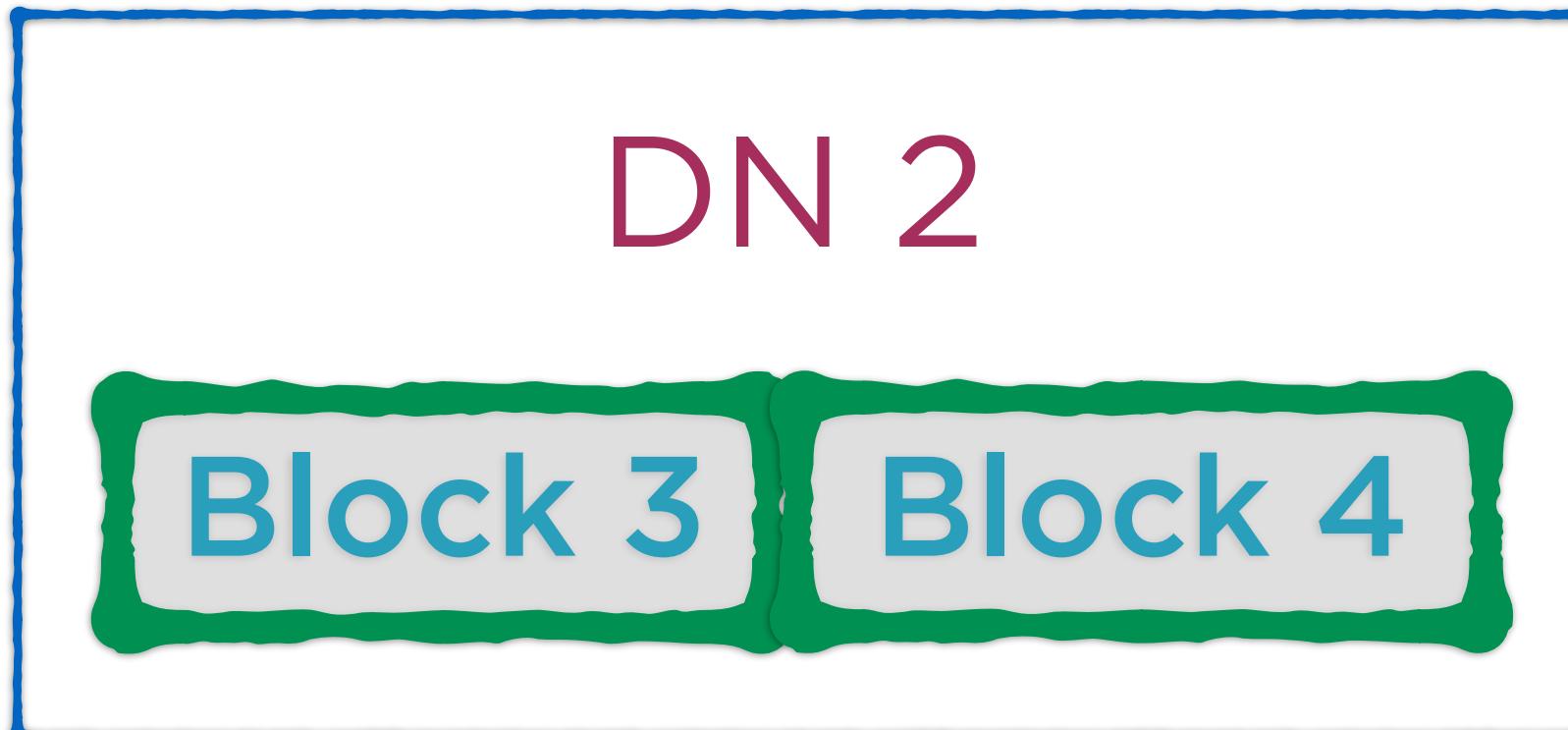
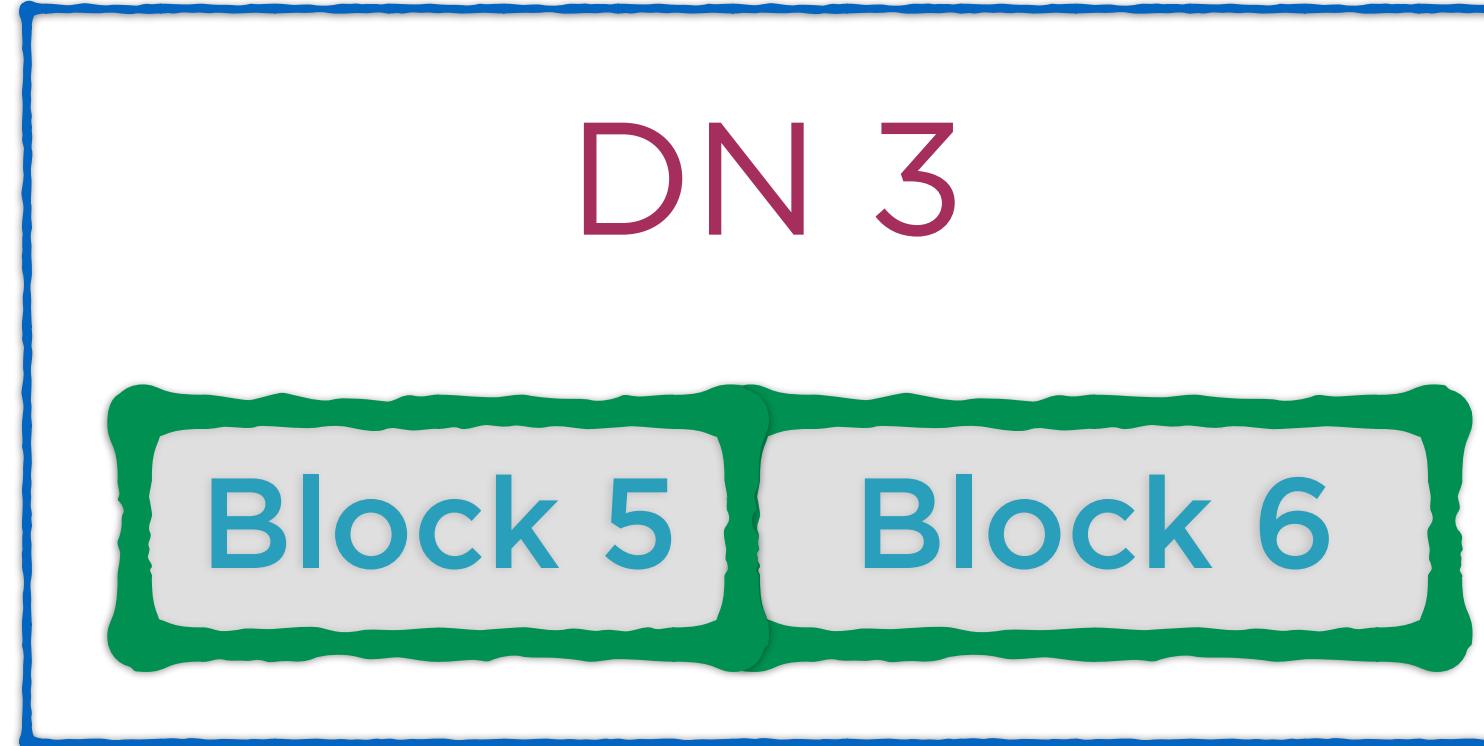
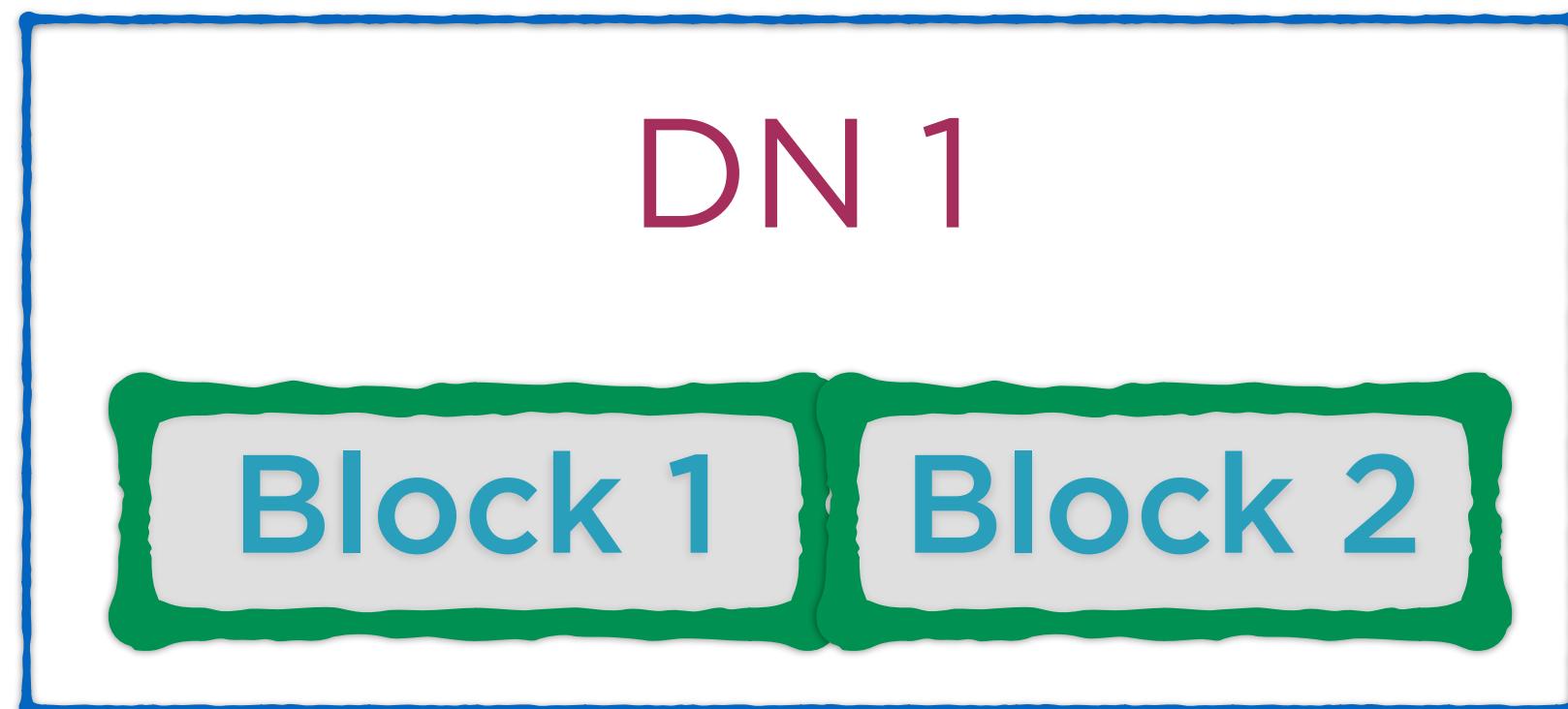
Name node

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

Reading a File in HDFS

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

Reading a File in HDFS

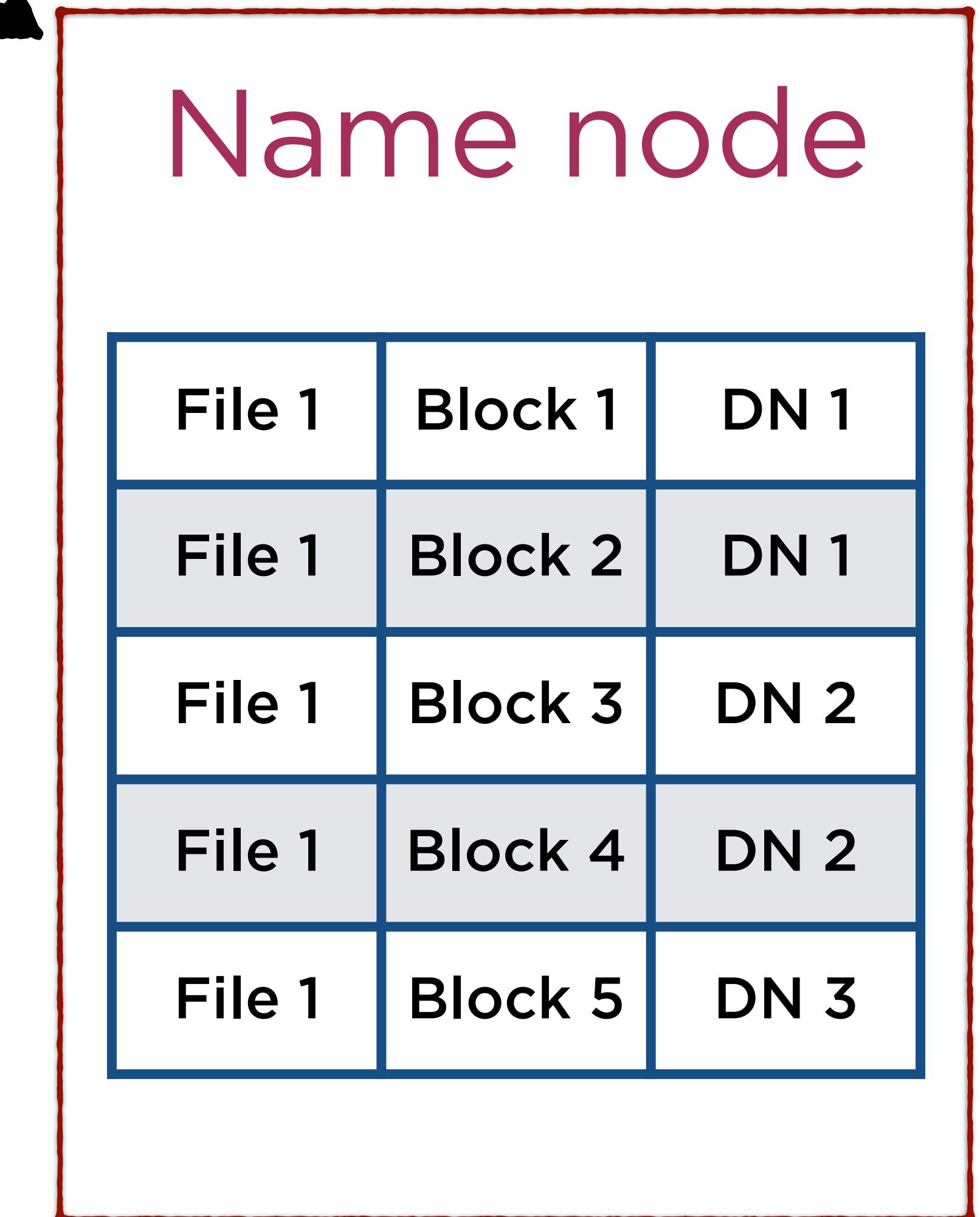
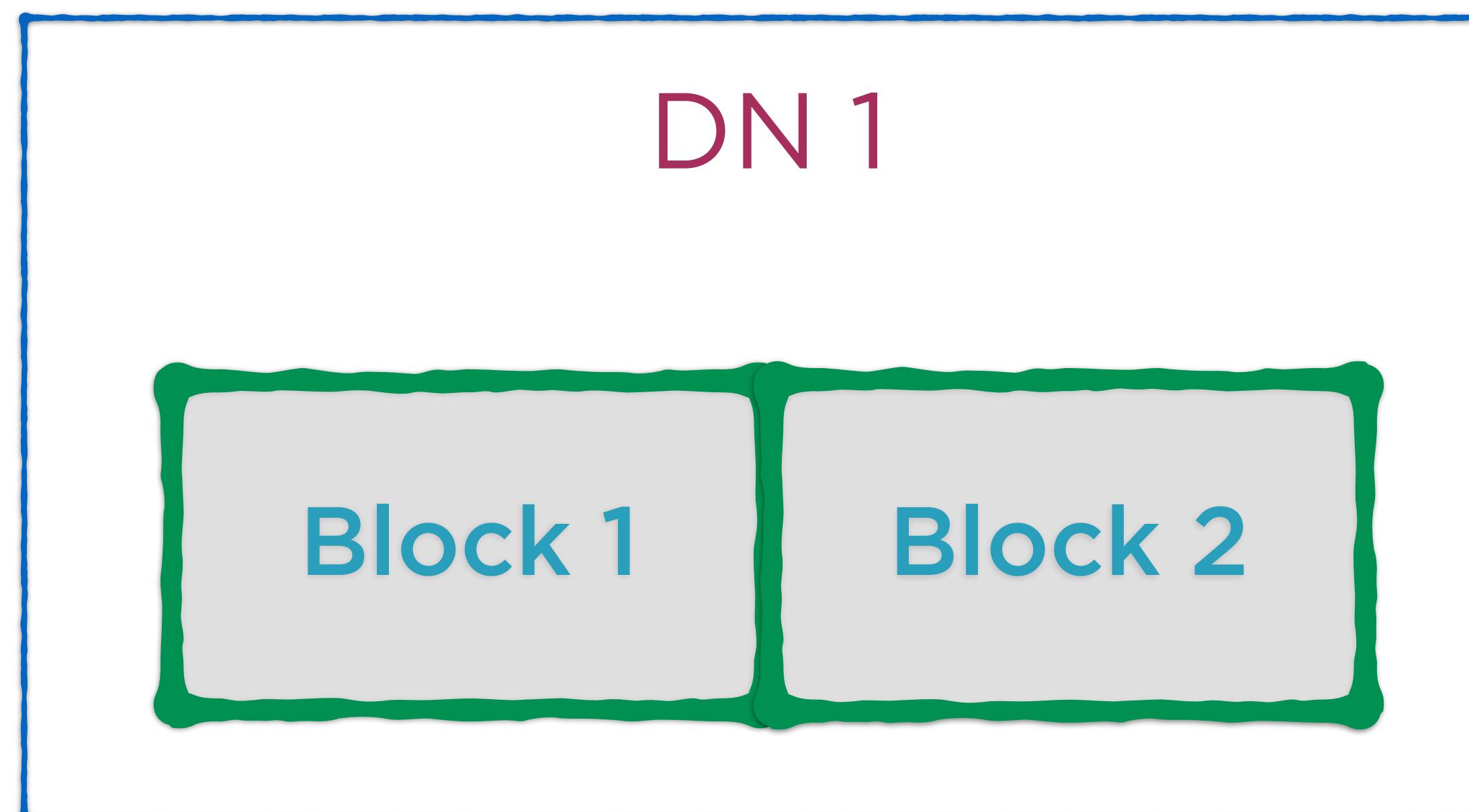


Name node

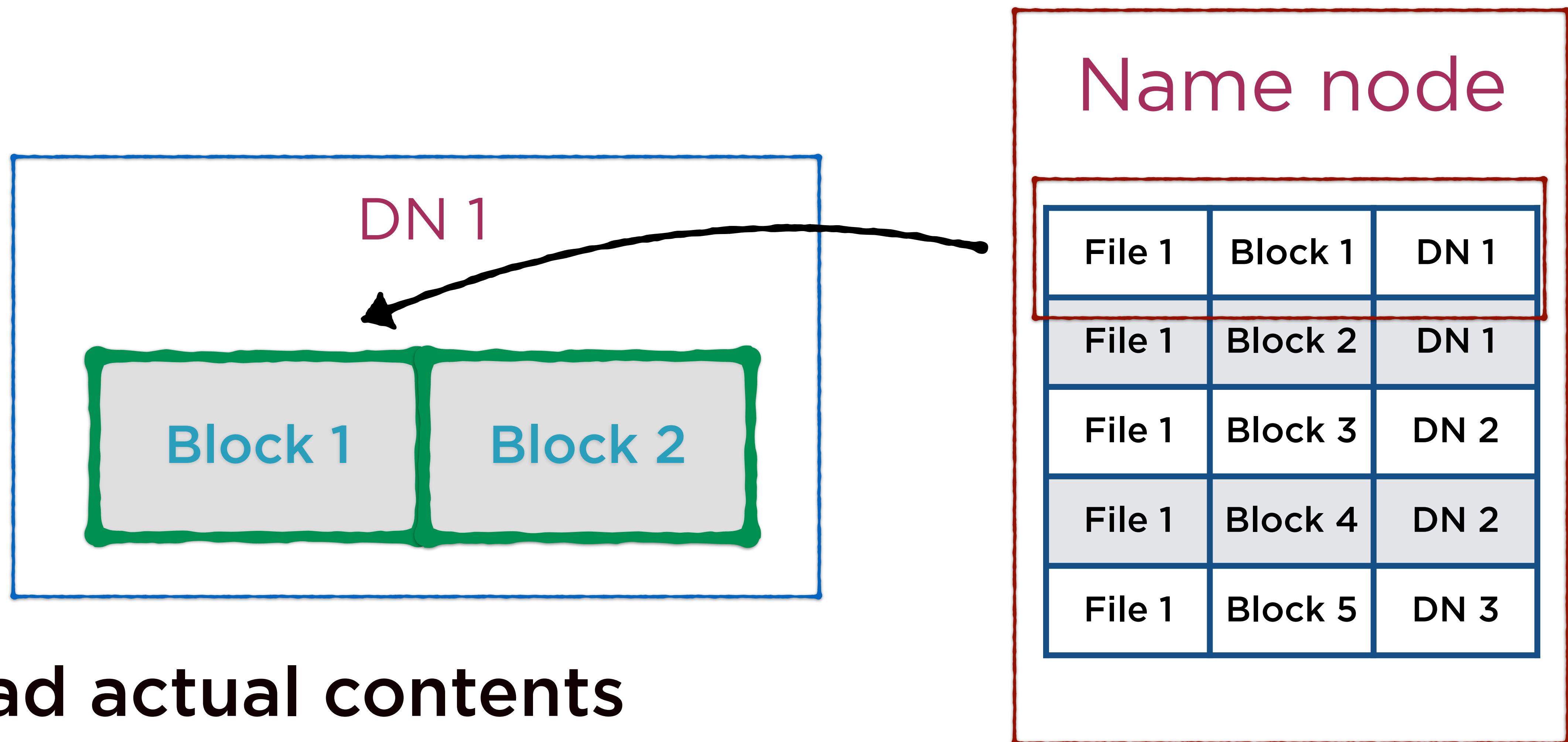
File 1	Block 1	DN 1
File 1	Block 2	DN 1
File 1	Block 3	DN 2
File 1	Block 4	DN 2
File 1	Block 5	DN 3

Reading a File in HDFS

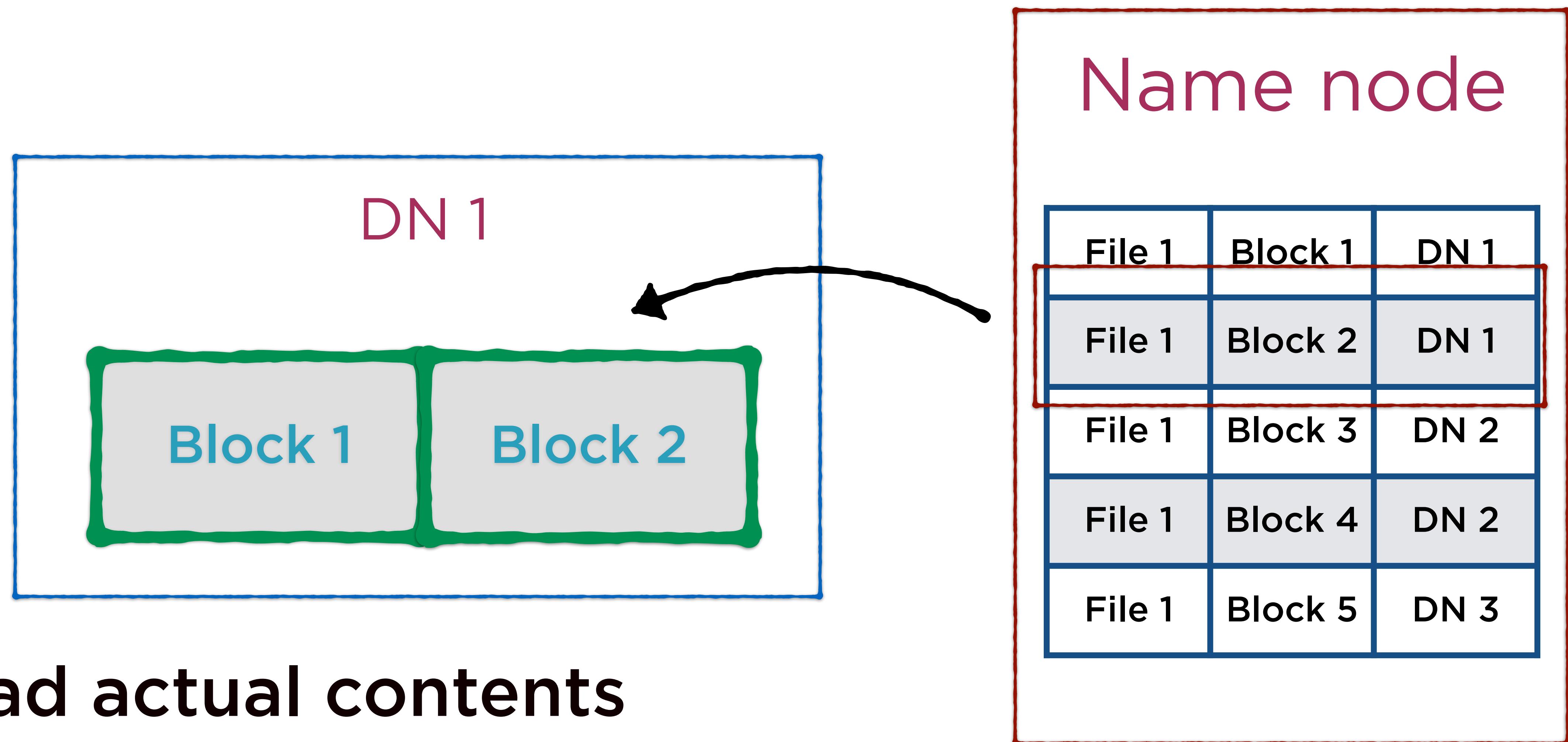
Request to the name node



Reading a File in HDFS

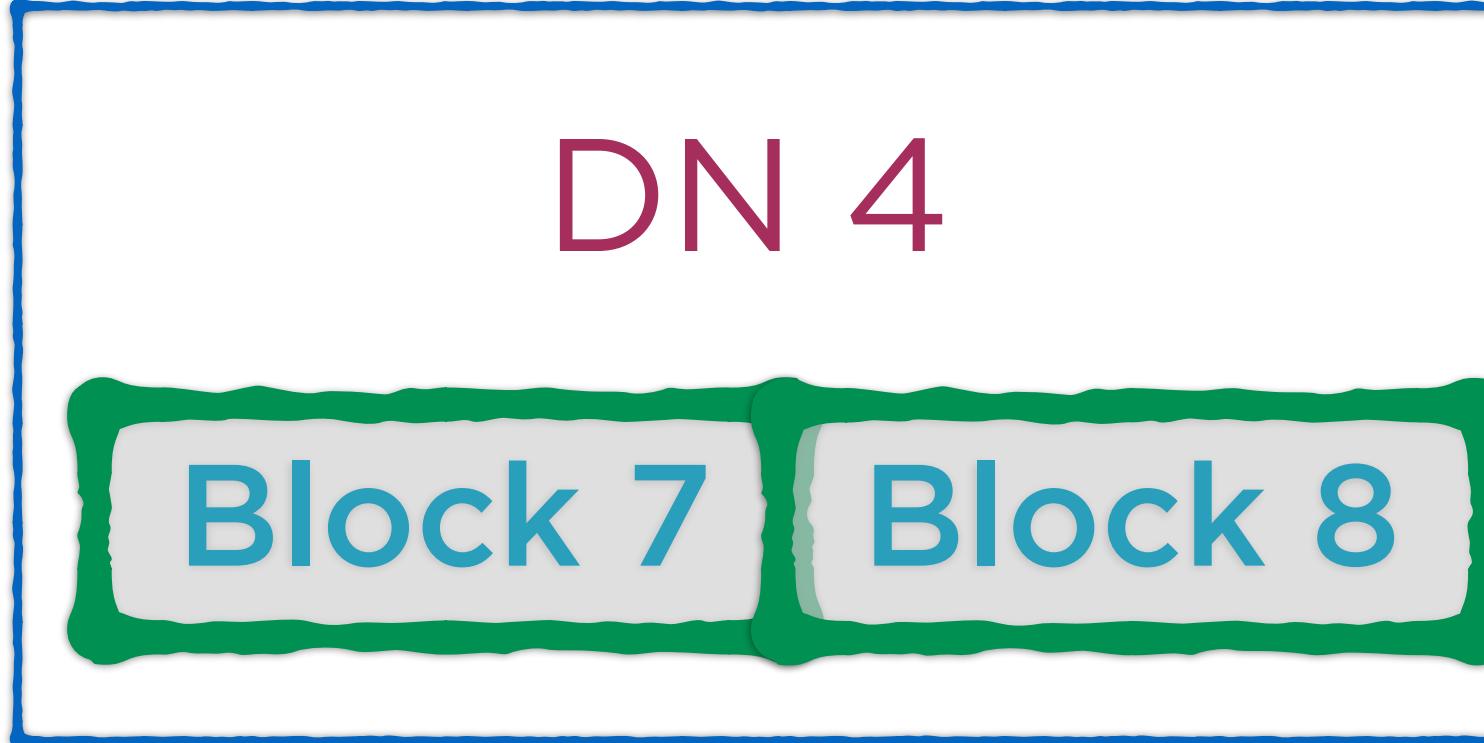
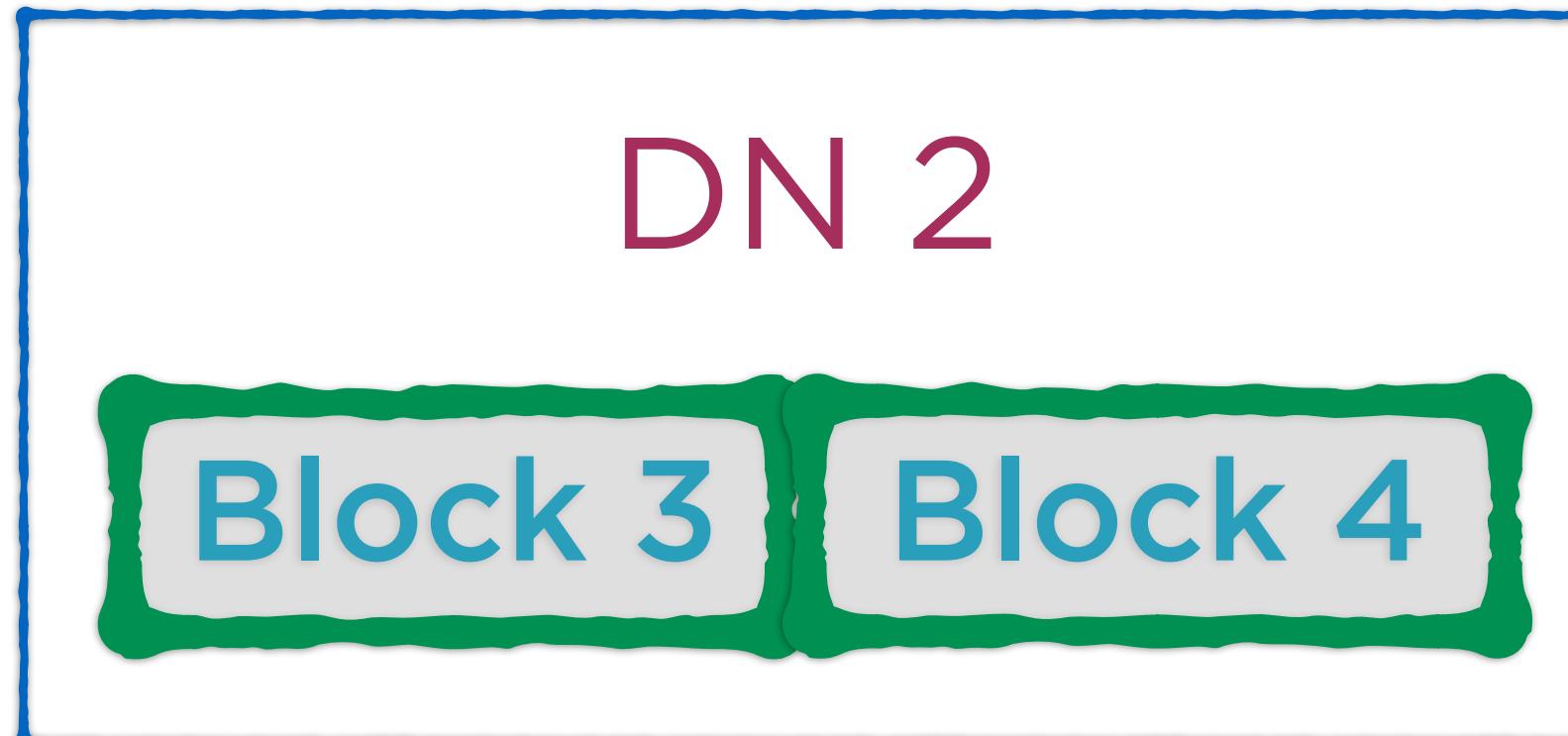
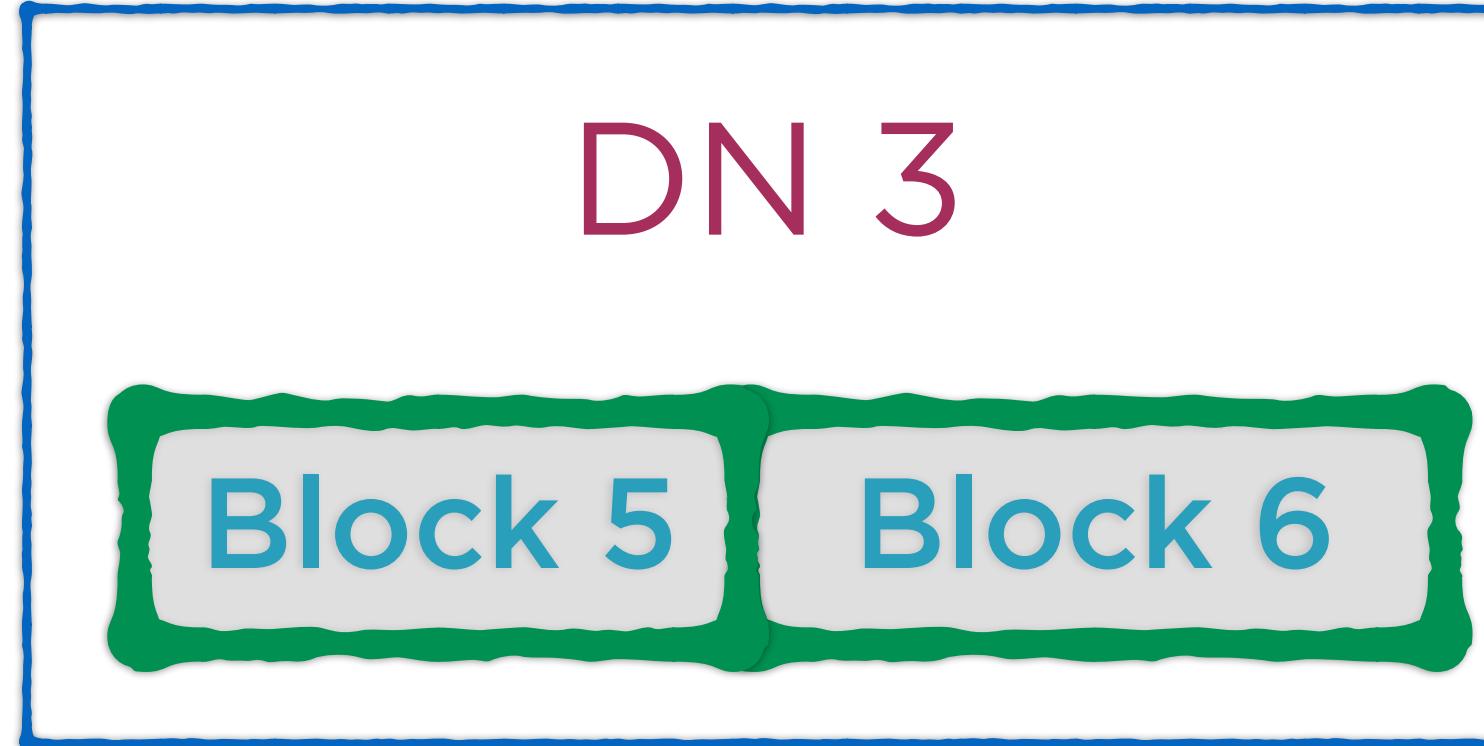
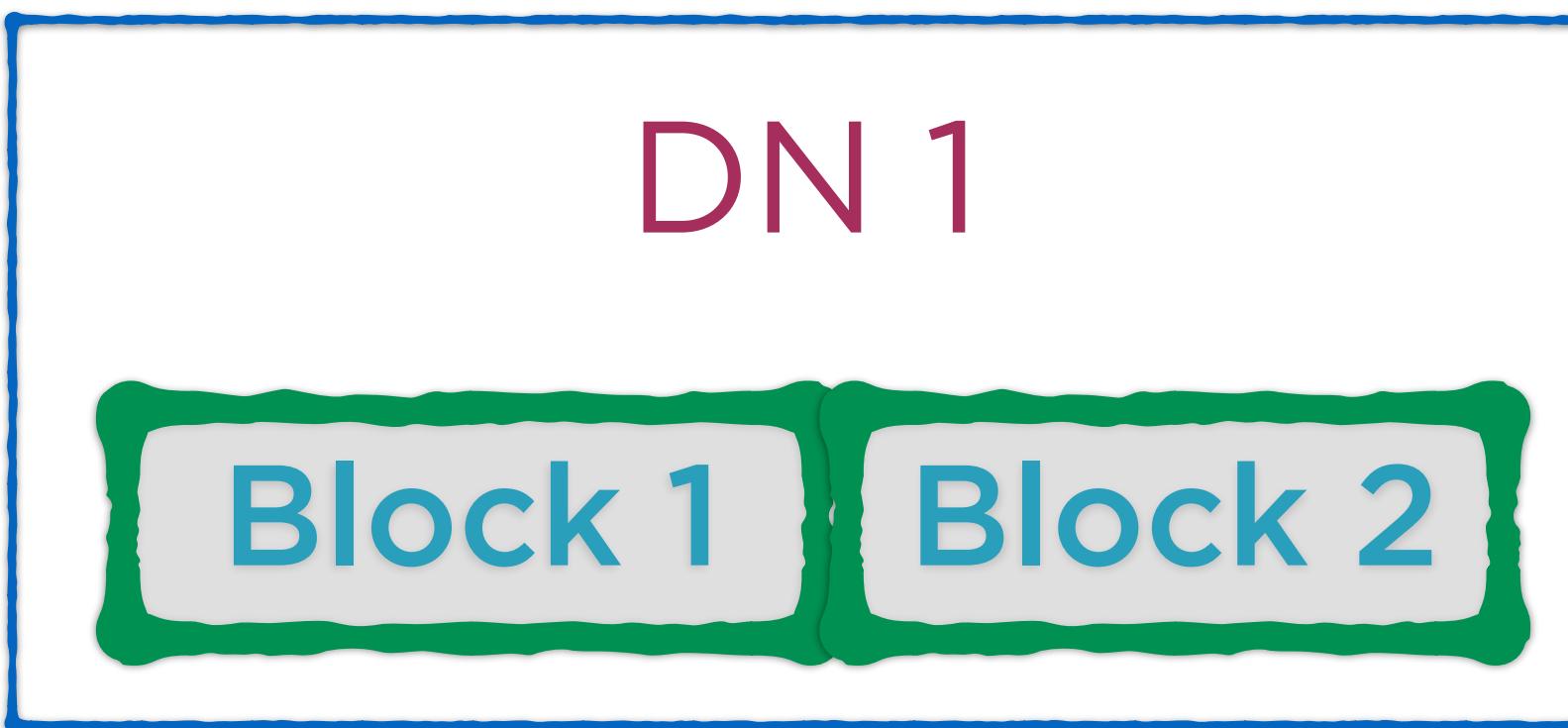


Reading a File in HDFS



Read actual contents
from the block

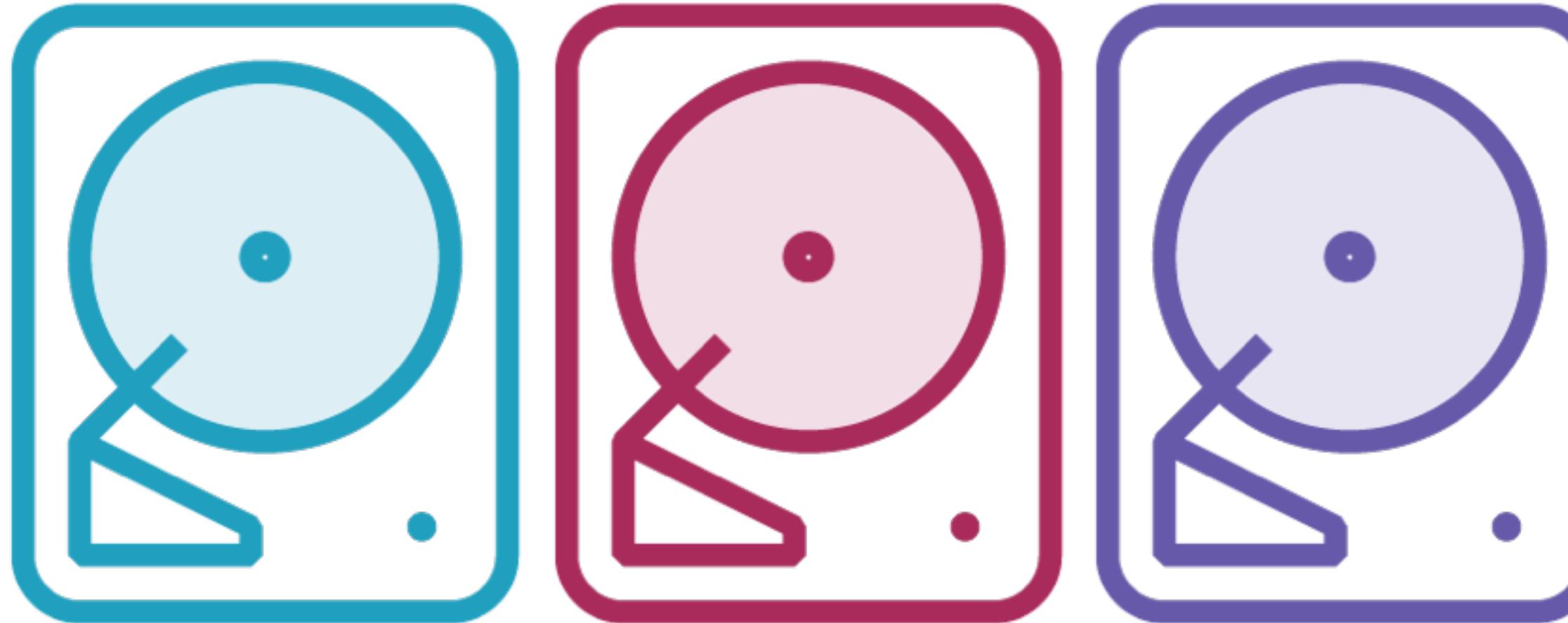
A File Stored in HDFS



Name node

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

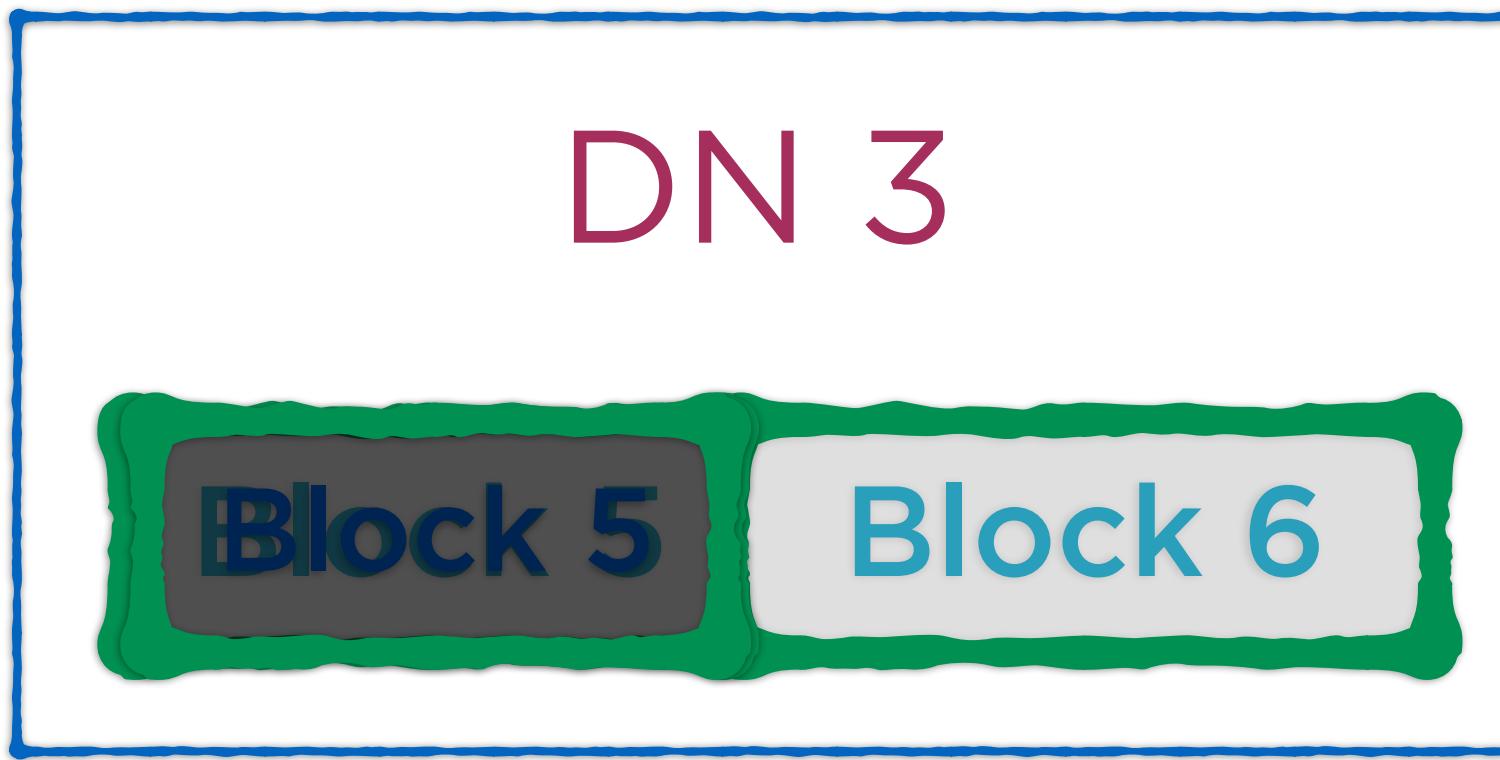
Challenges of Distributed Storage



**Failure management
in the data nodes**

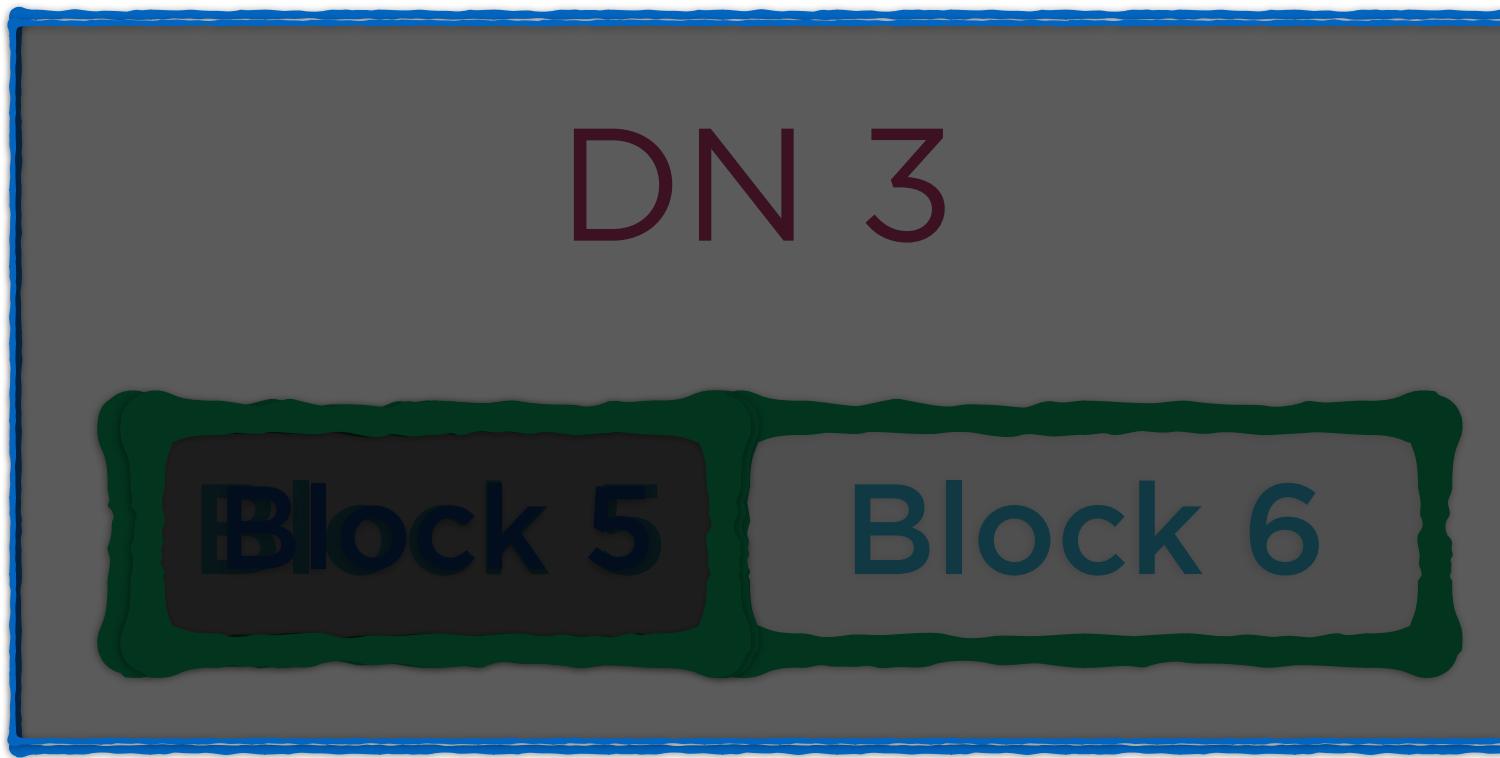
**Failure management
for the name node**

A File Stored in HDFS



What if one of the
blocks gets corrupted?

A File Stored in HDFS



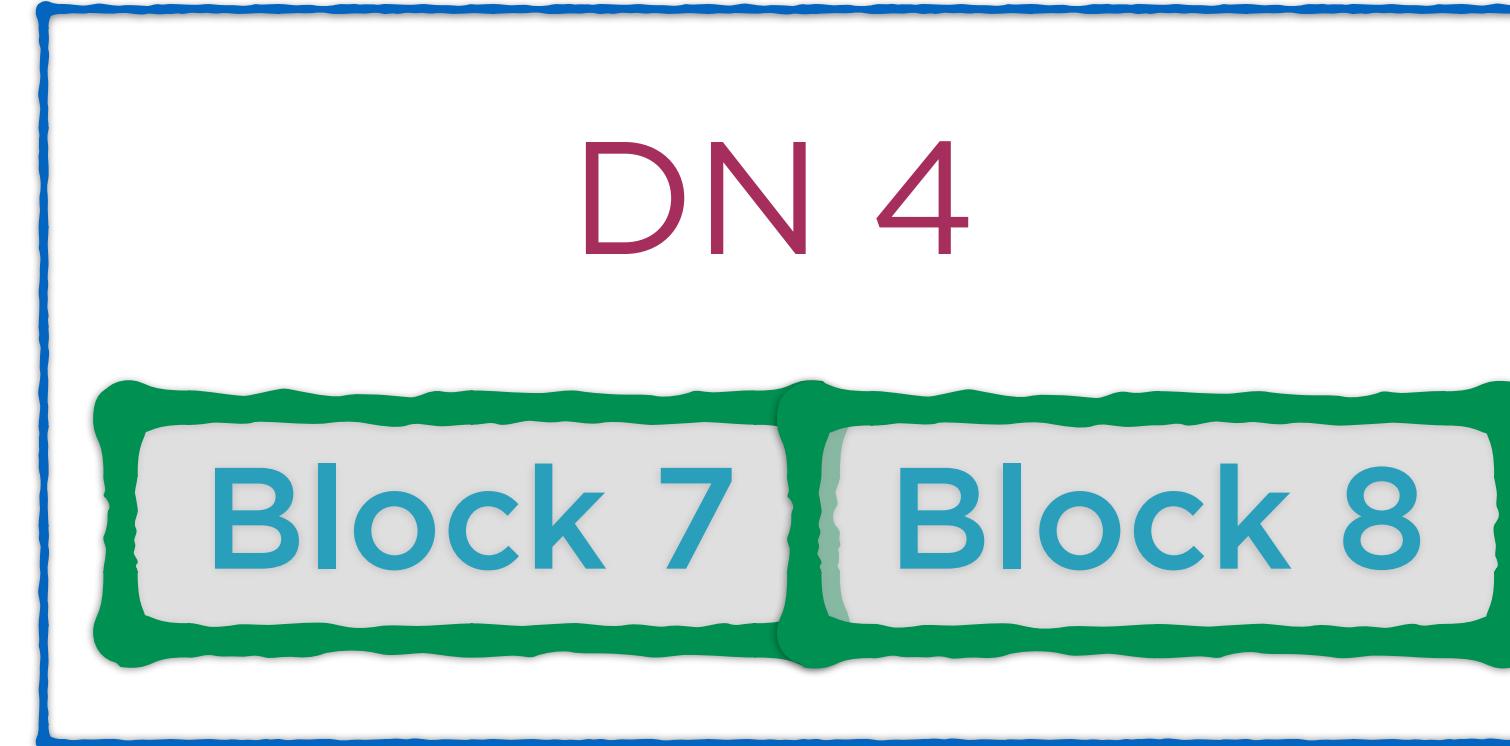
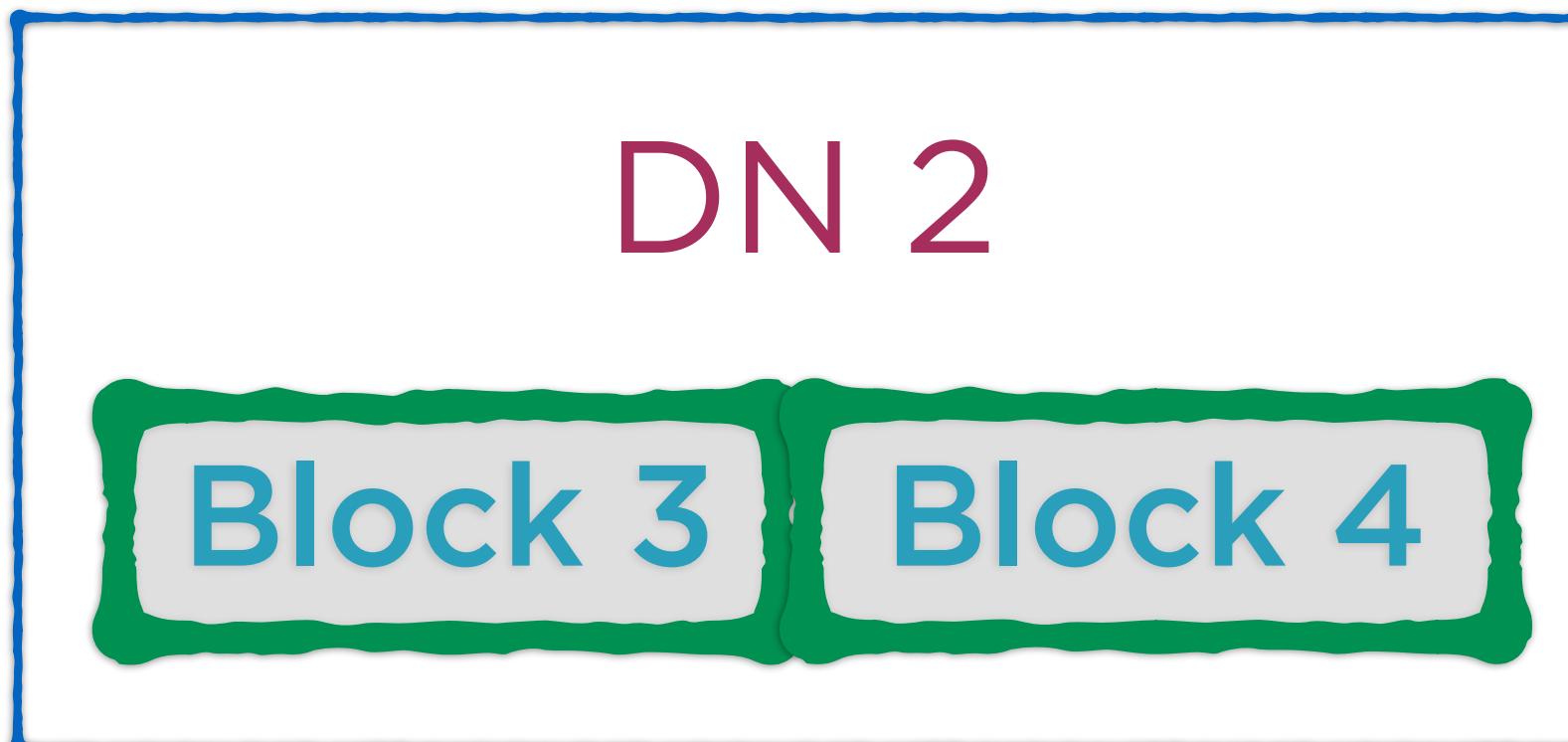
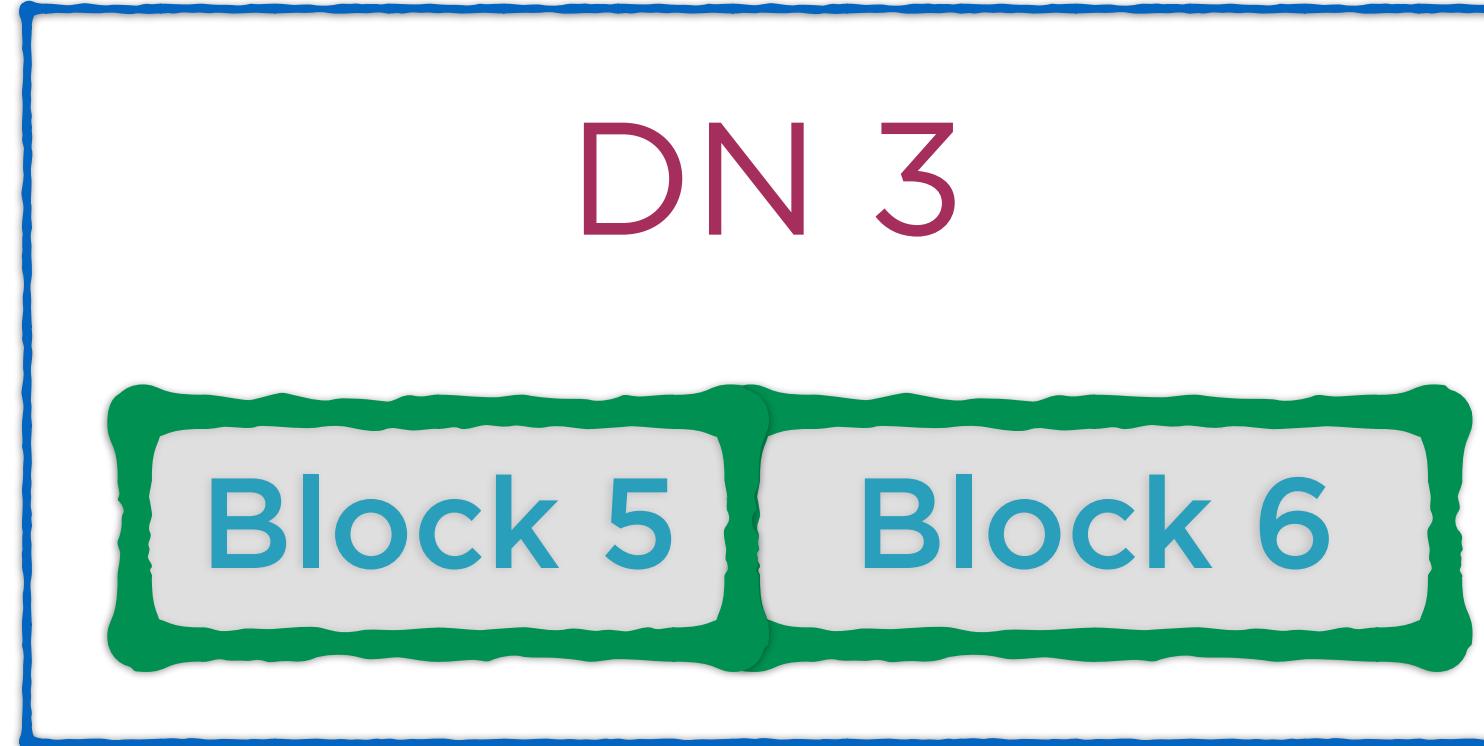
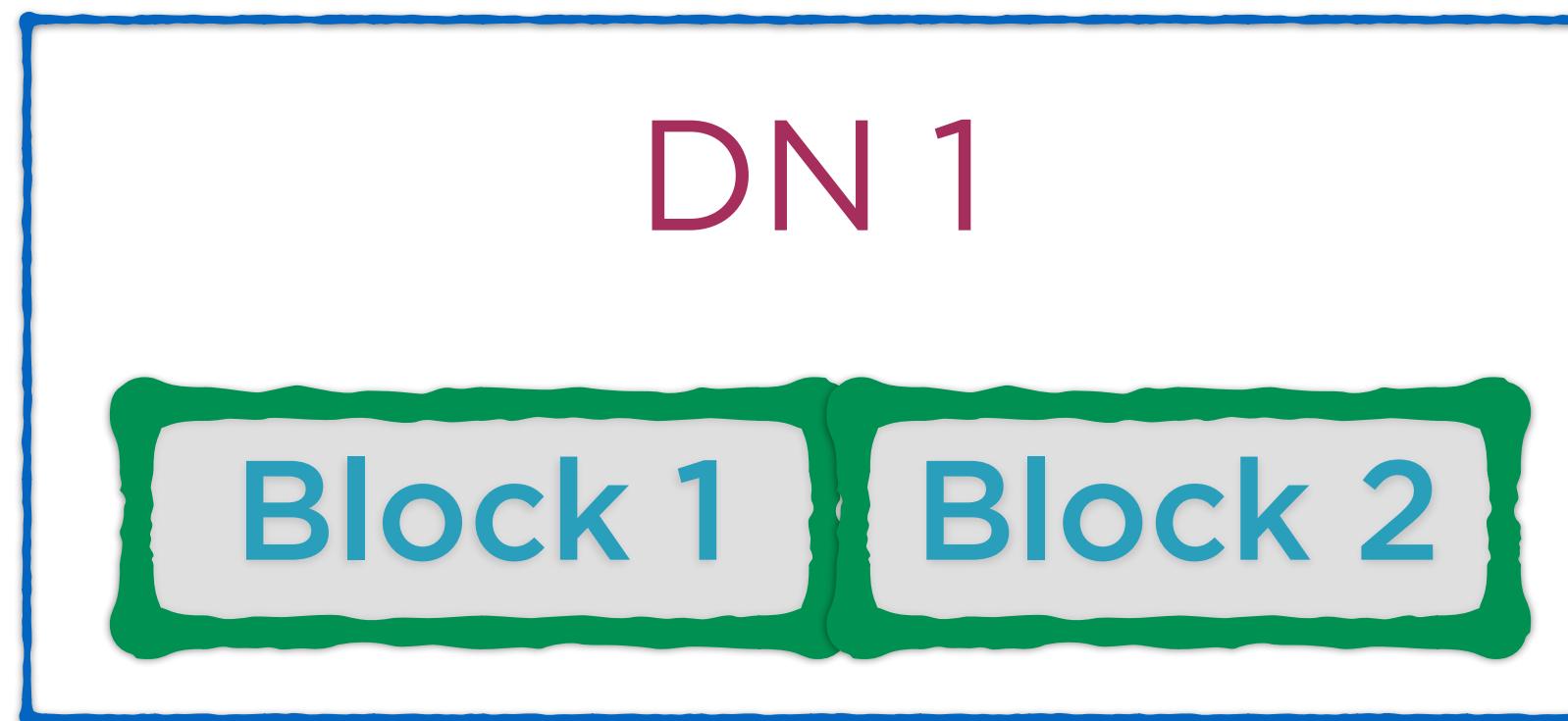
**What if a data node
containing some blocks
crashes?**

Managing Failures in Data Nodes



**Define a
replication factor**

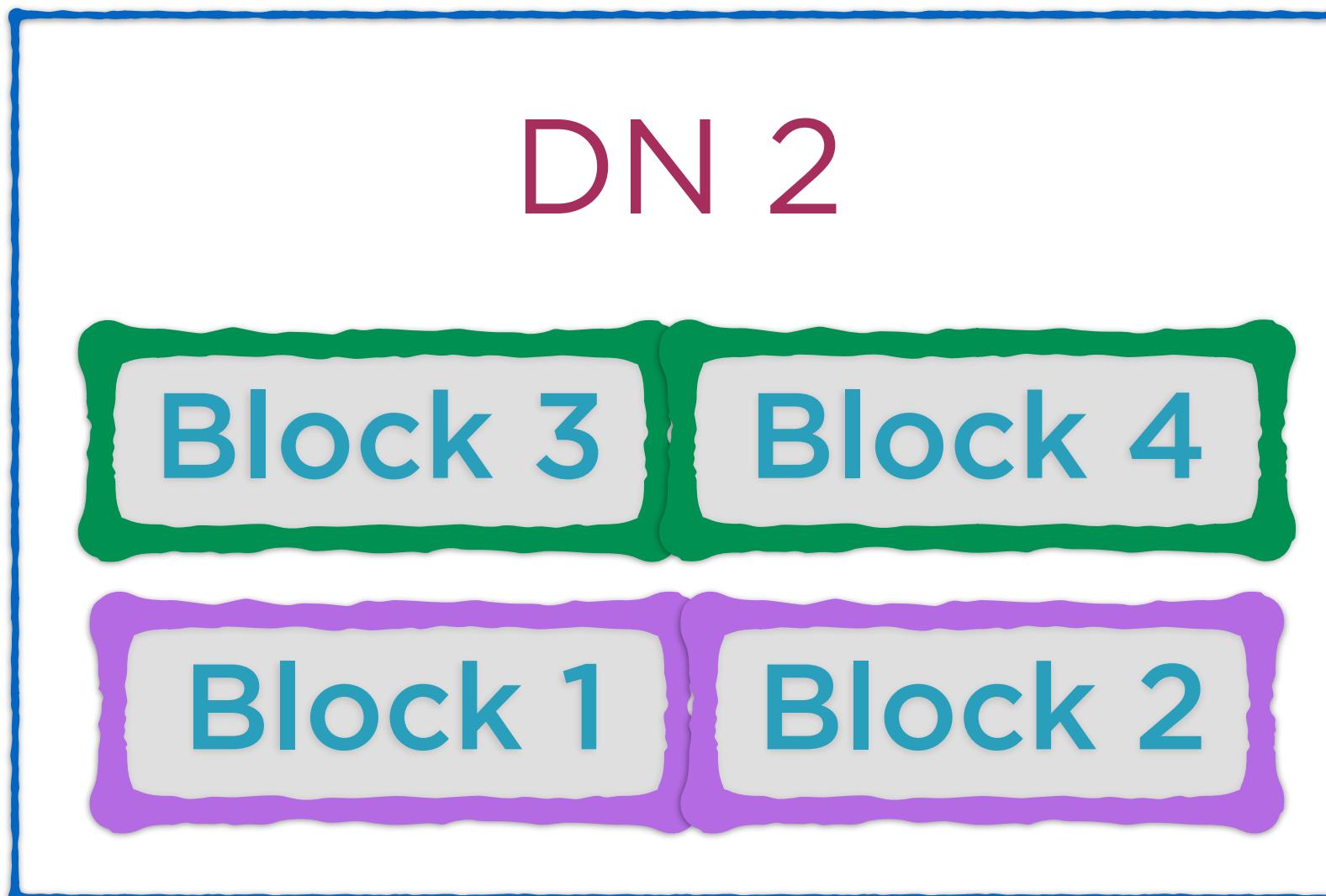
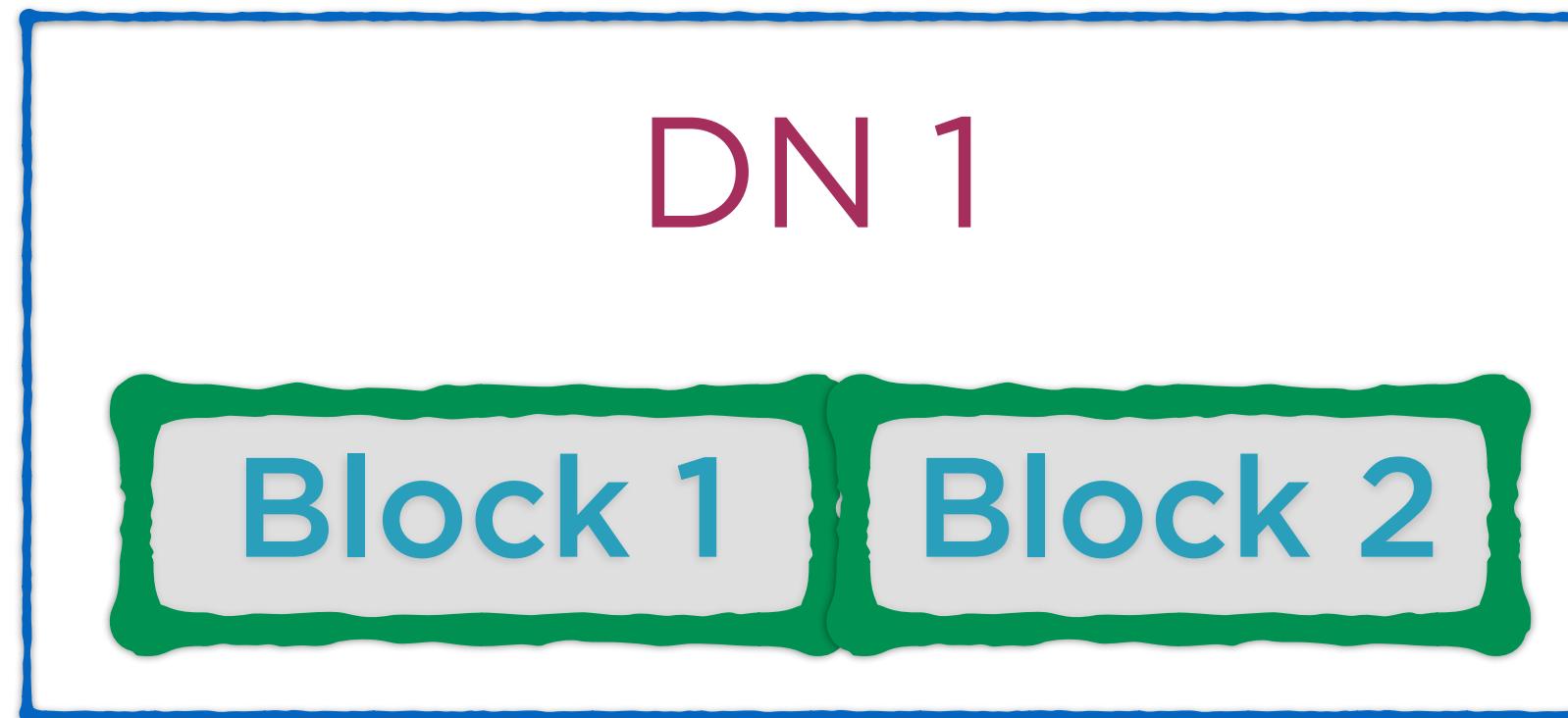
A File Stored in HDFS



Name node

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

Replication



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

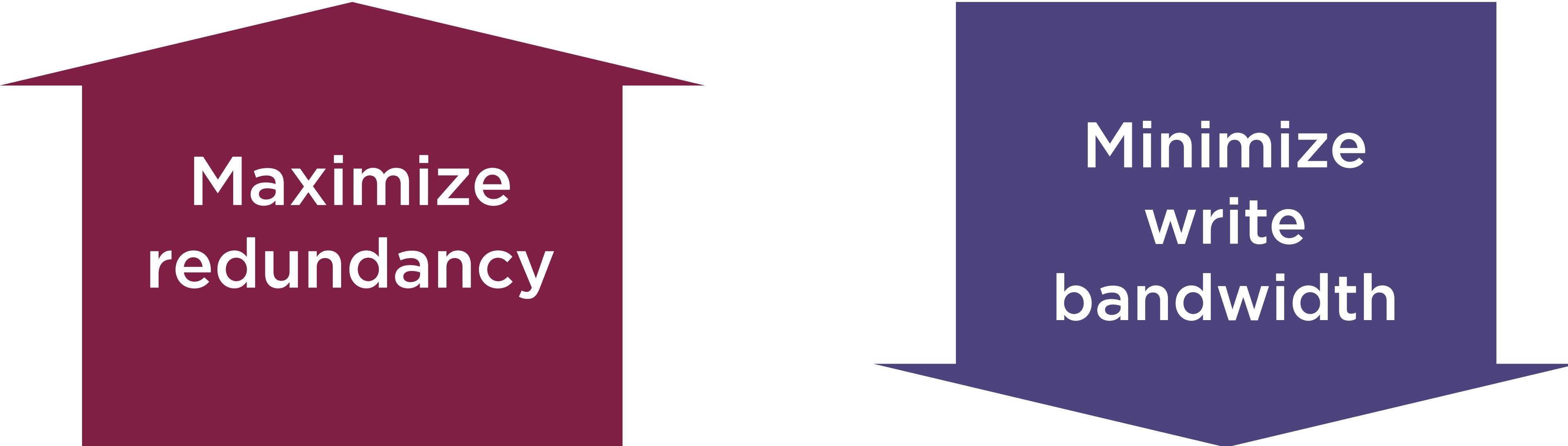
Replication

The replica locations are also stored in the name node

Name node

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

Choosing Replica Locations

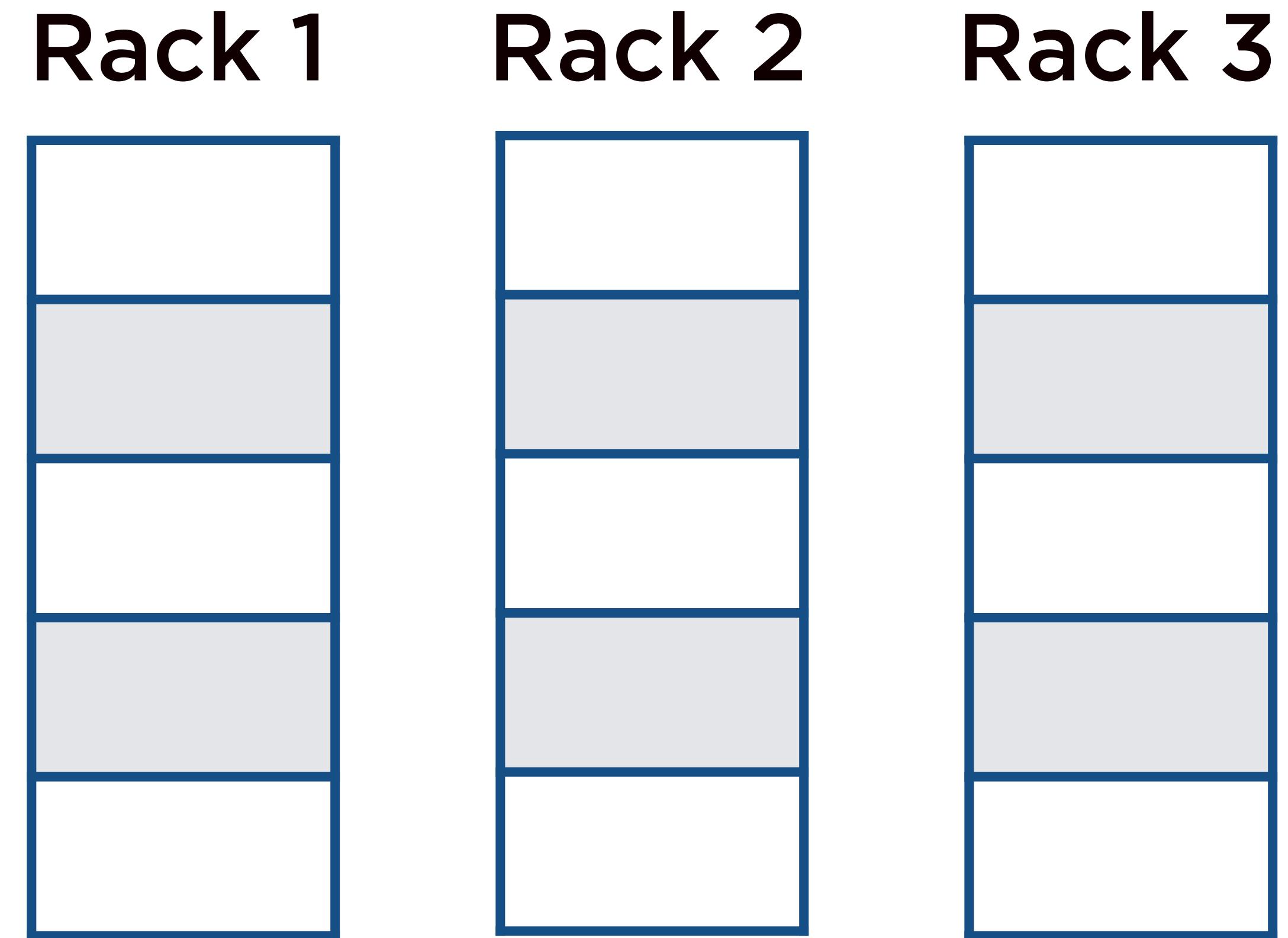


Maximize
redundancy

Minimize
write
bandwidth



**Maximize
redundancy**

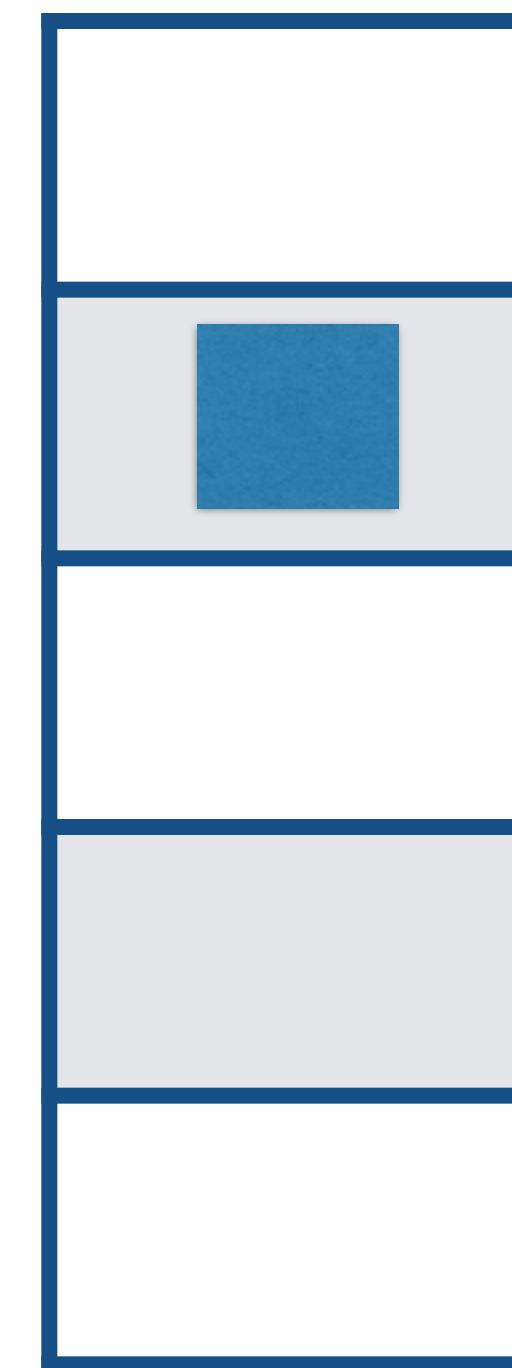
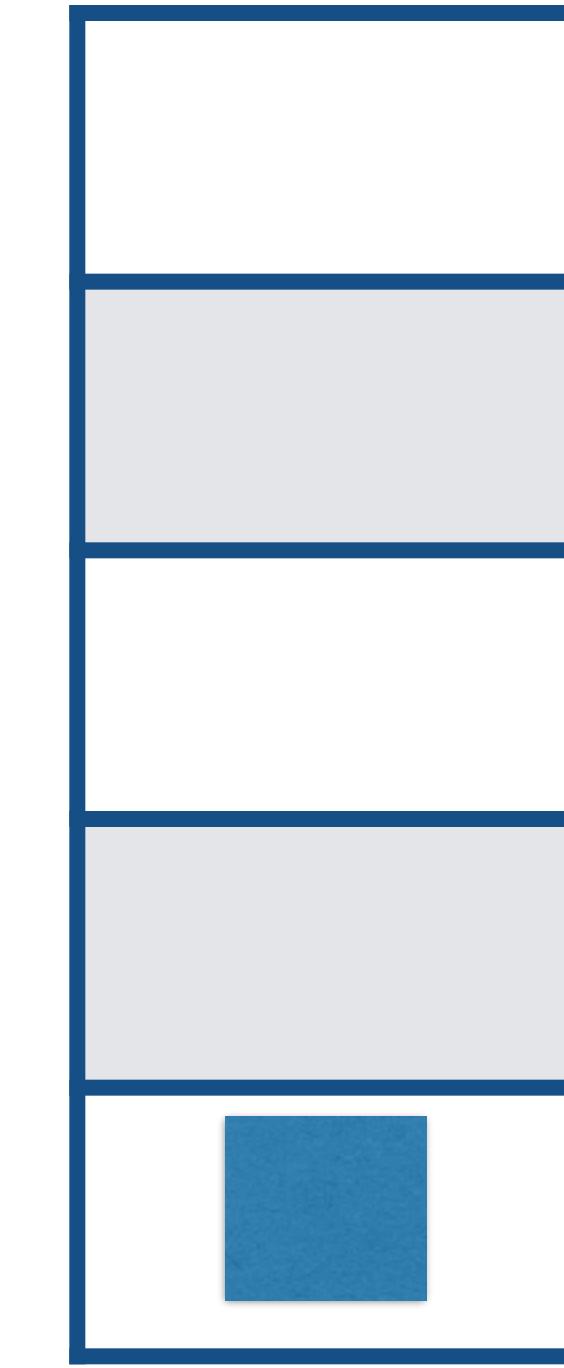
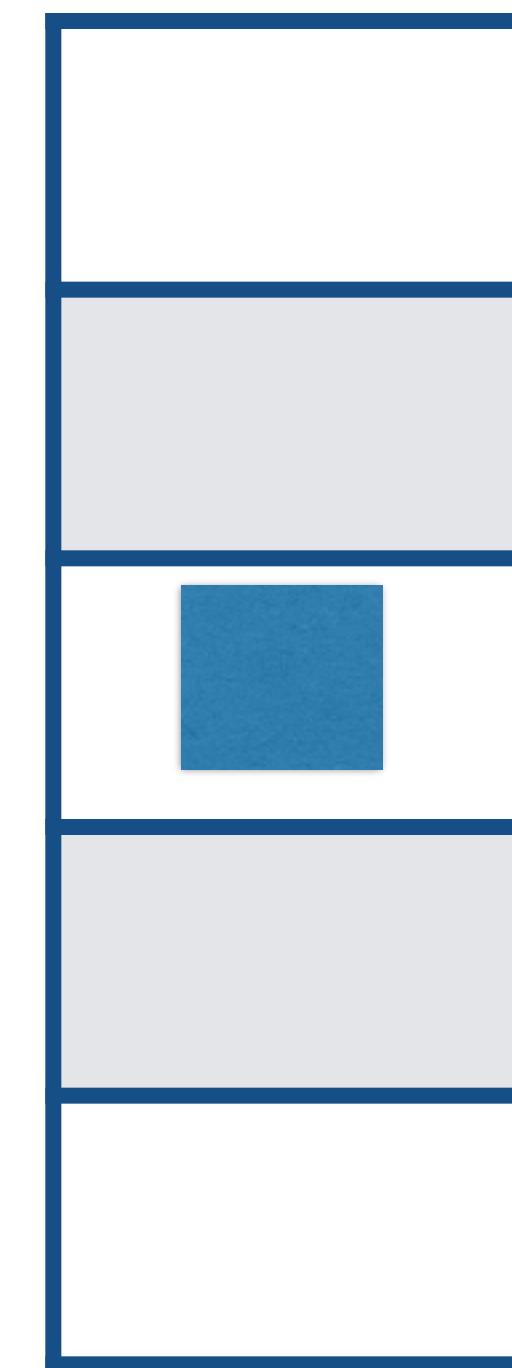


**Servers in a
data center**



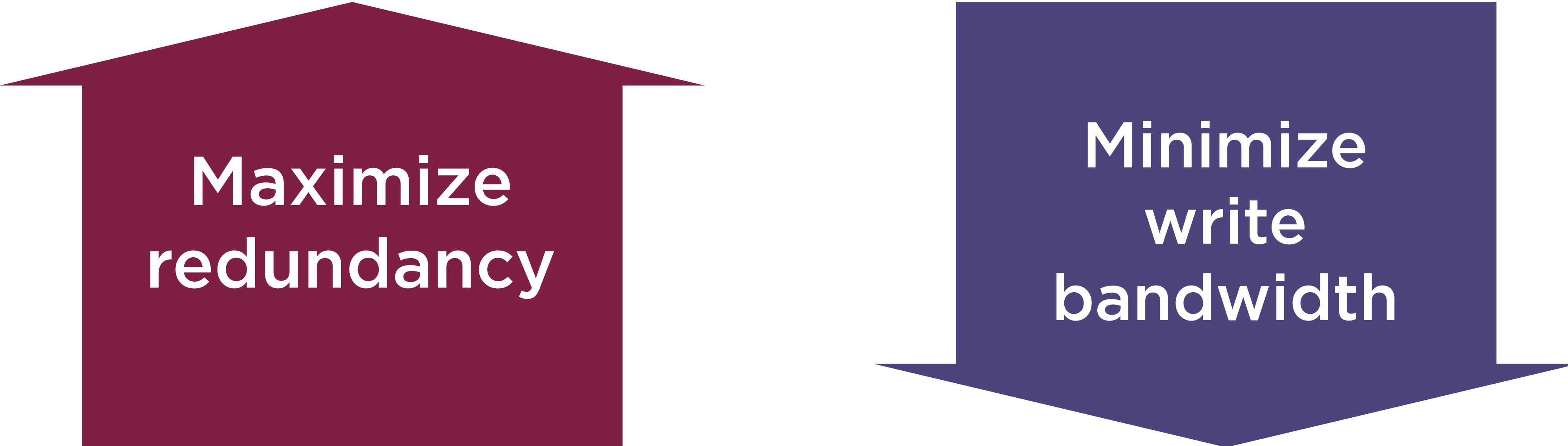
**Maximize
redundancy**

Rack 1 Rack 2 Rack 3



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

Choosing Replica Locations



Maximize
redundancy

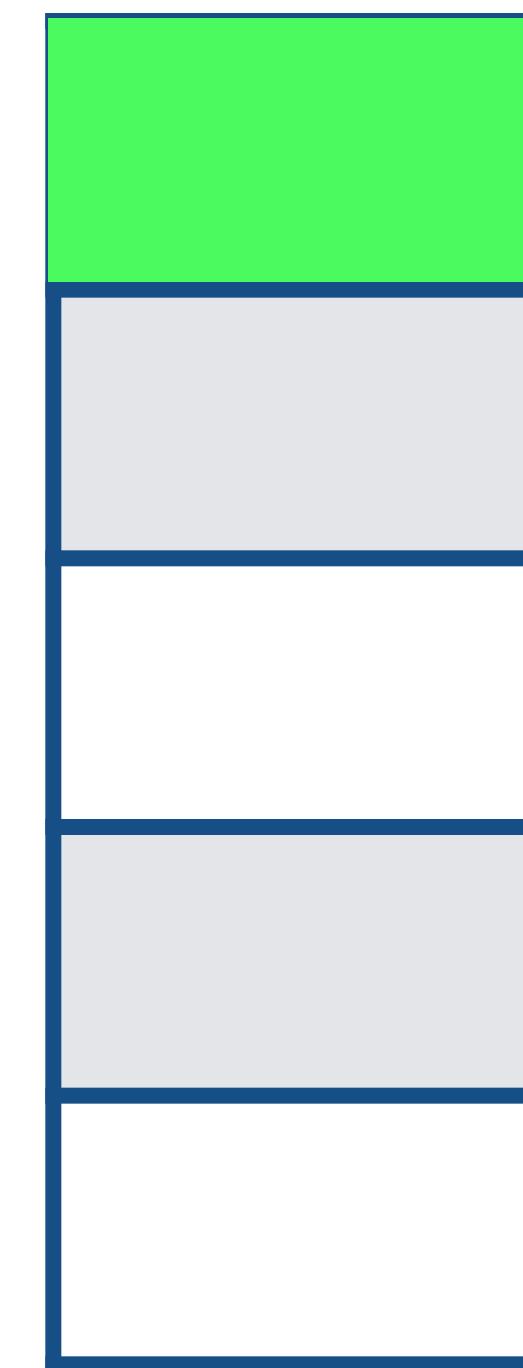
Minimize
write
bandwidth

Minimize
write
bandwidth

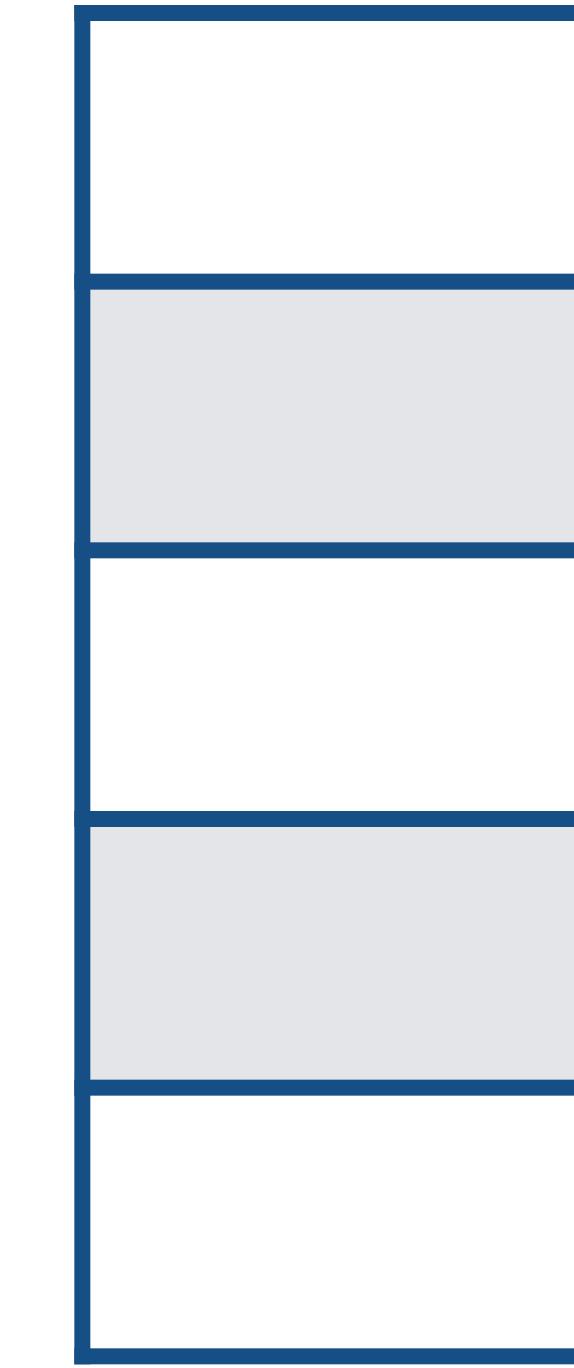
This requires that
replicas be stored
close to each other

Minimize
write
bandwidth

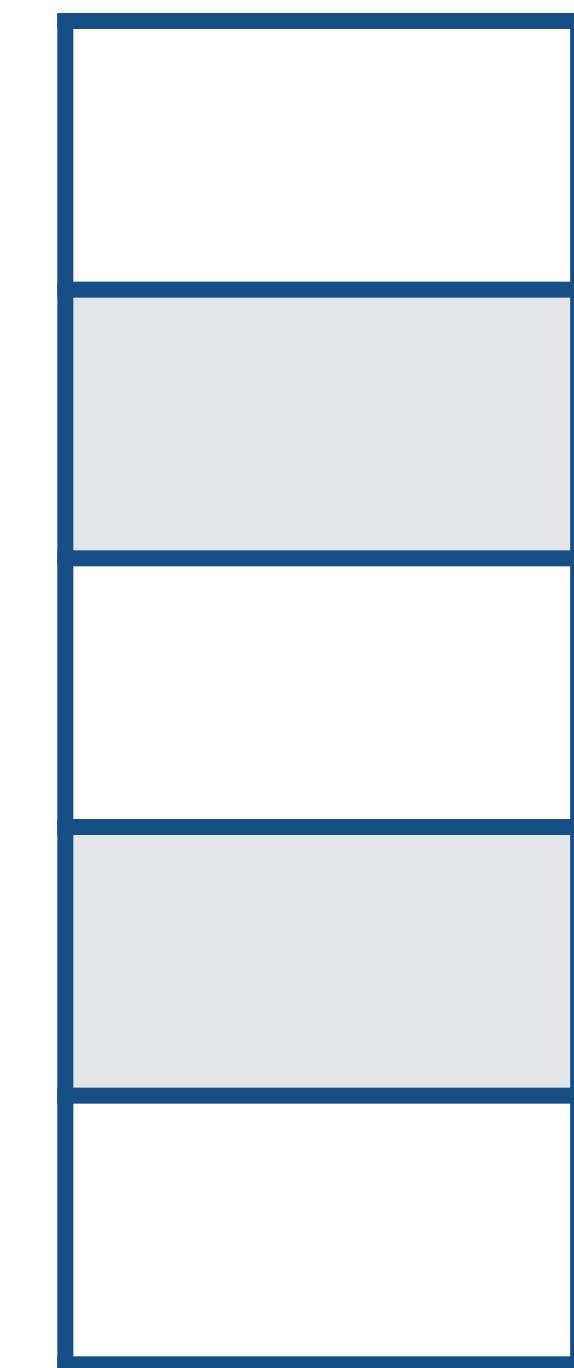
Rack 1



Rack 2

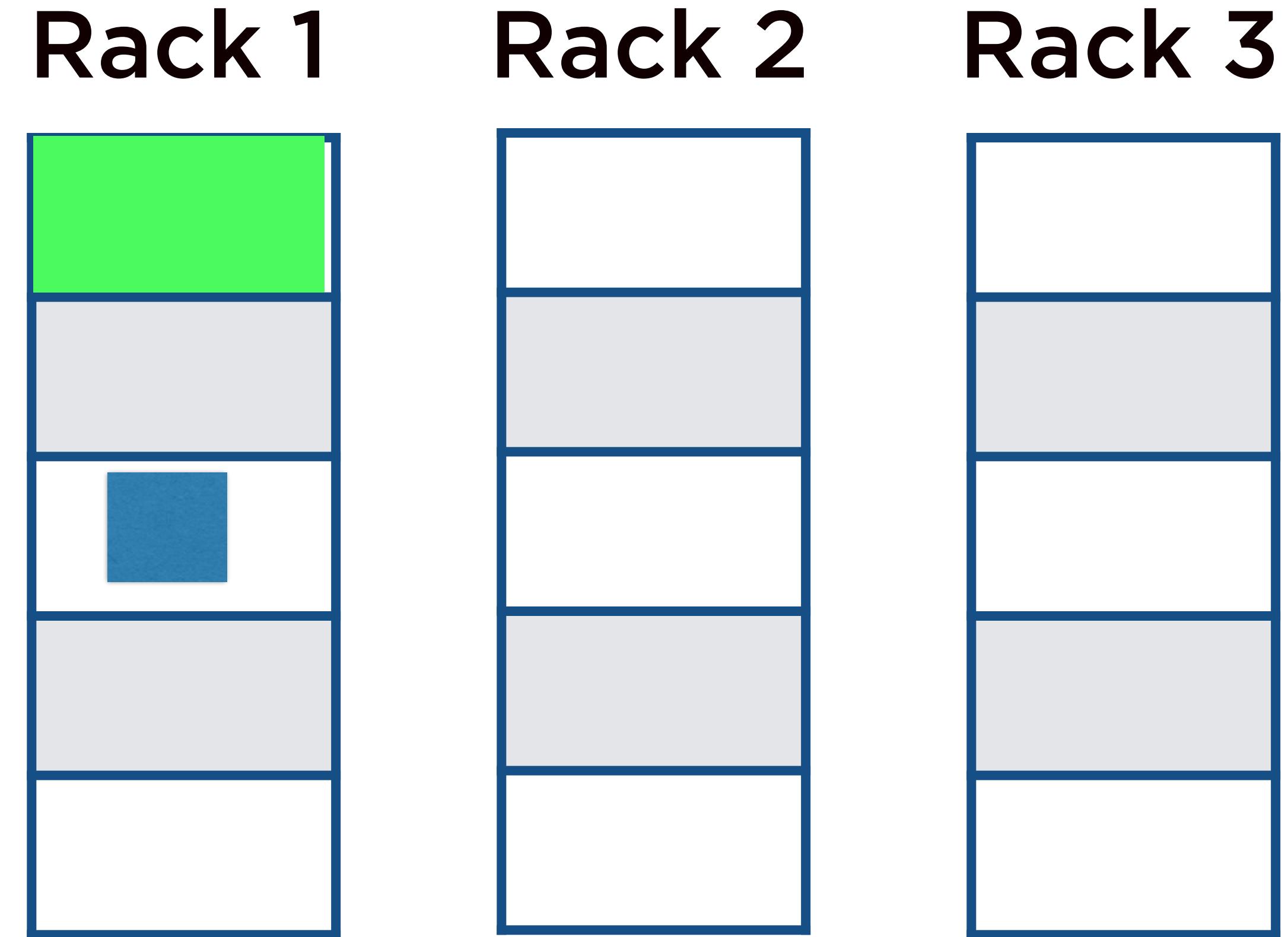


Rack 3



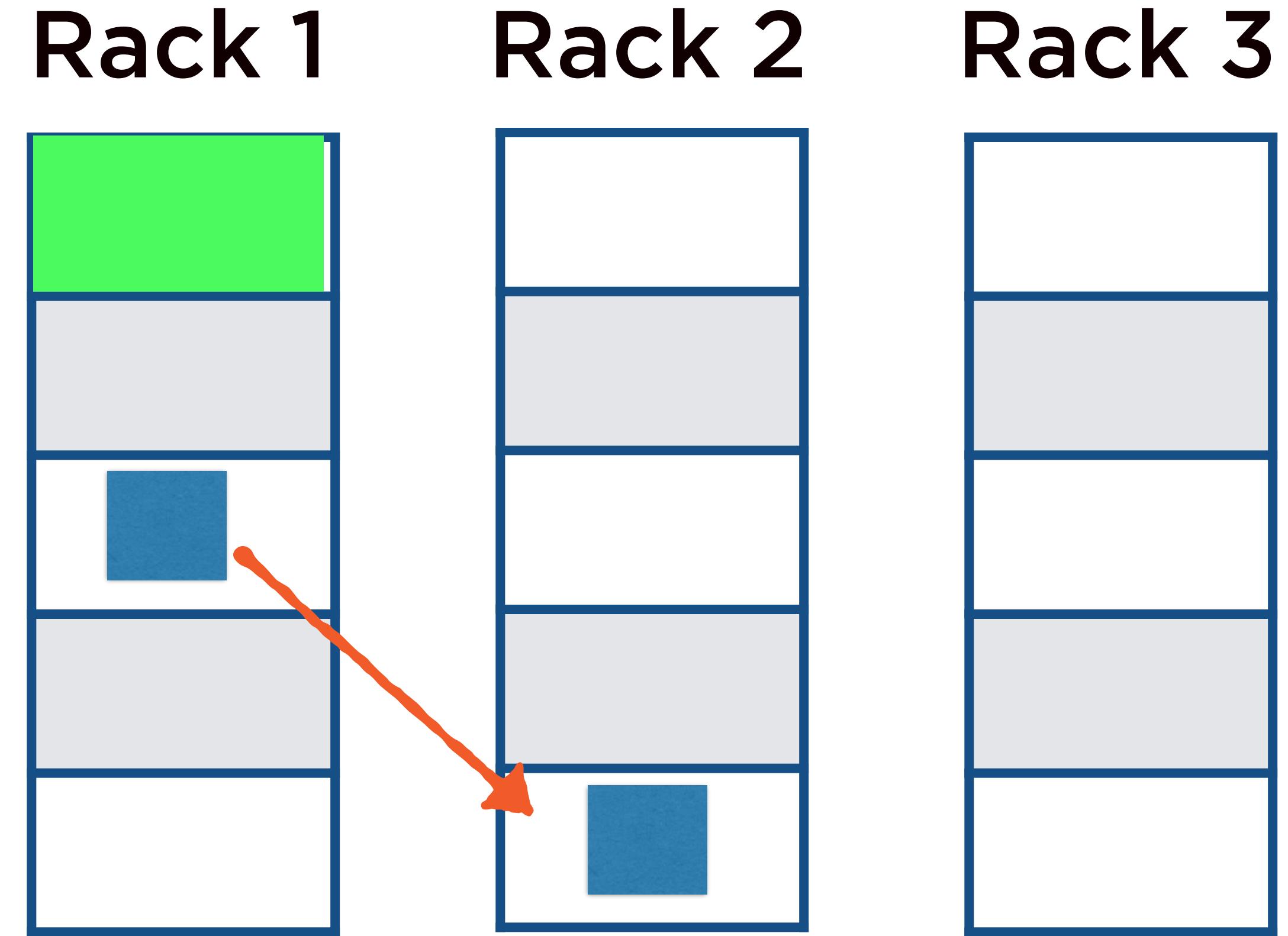
Node that runs the
replication pipeline

Minimize
write
bandwidth



This node chooses
the location for the
first replica

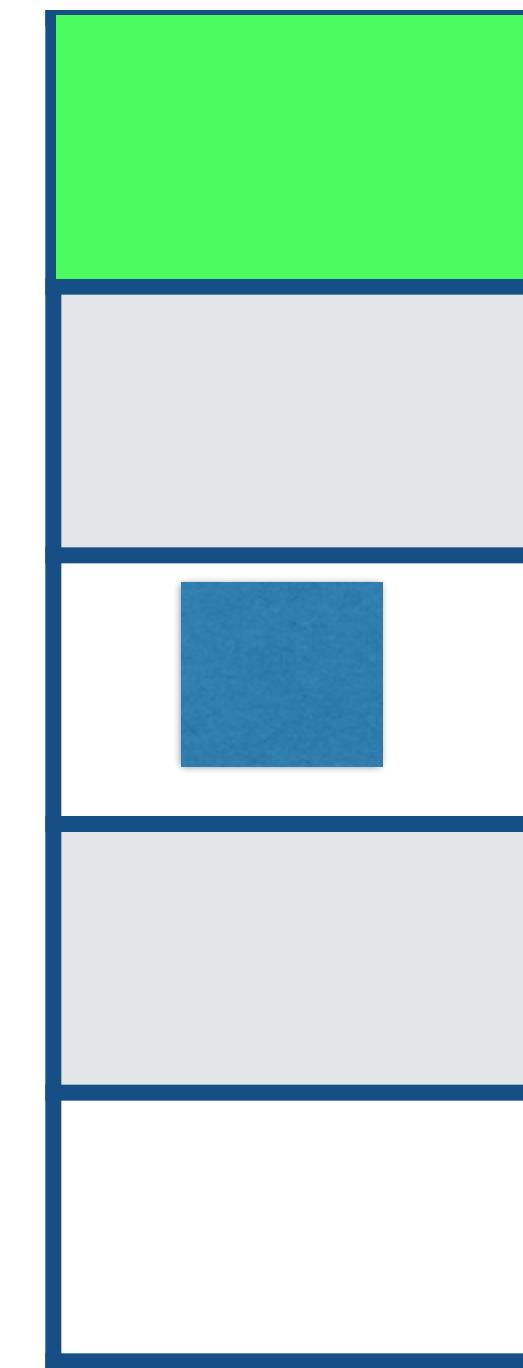
Minimize
write
bandwidth



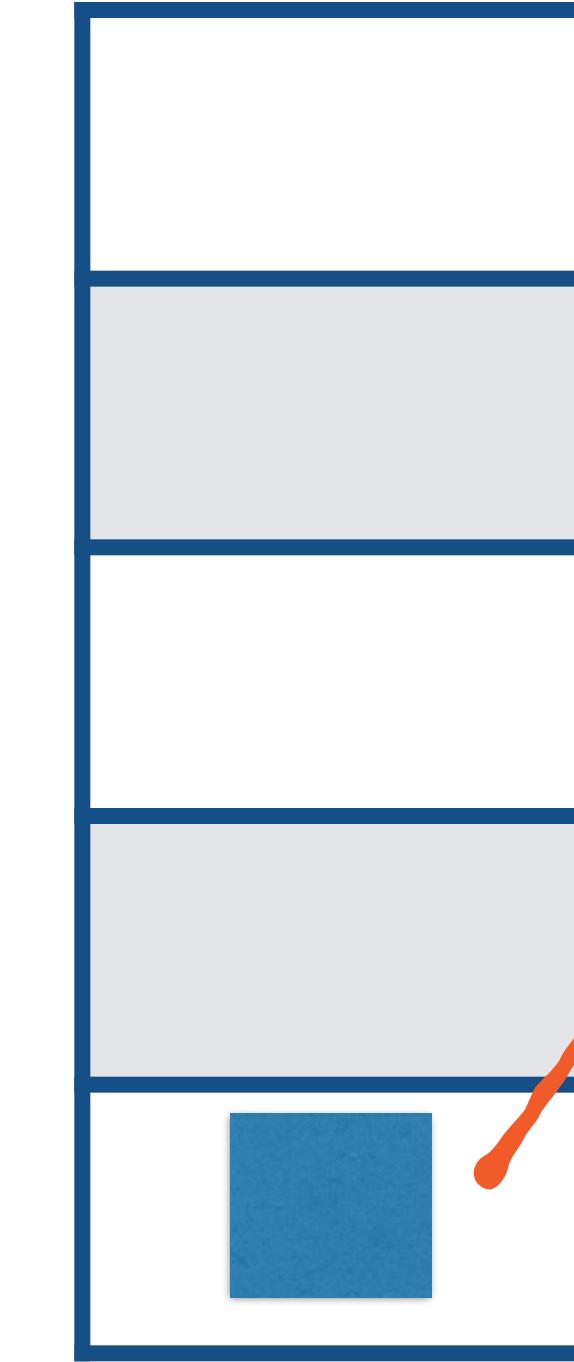
Data is forwarded
from here to the next
replica location

Minimize
write
bandwidth

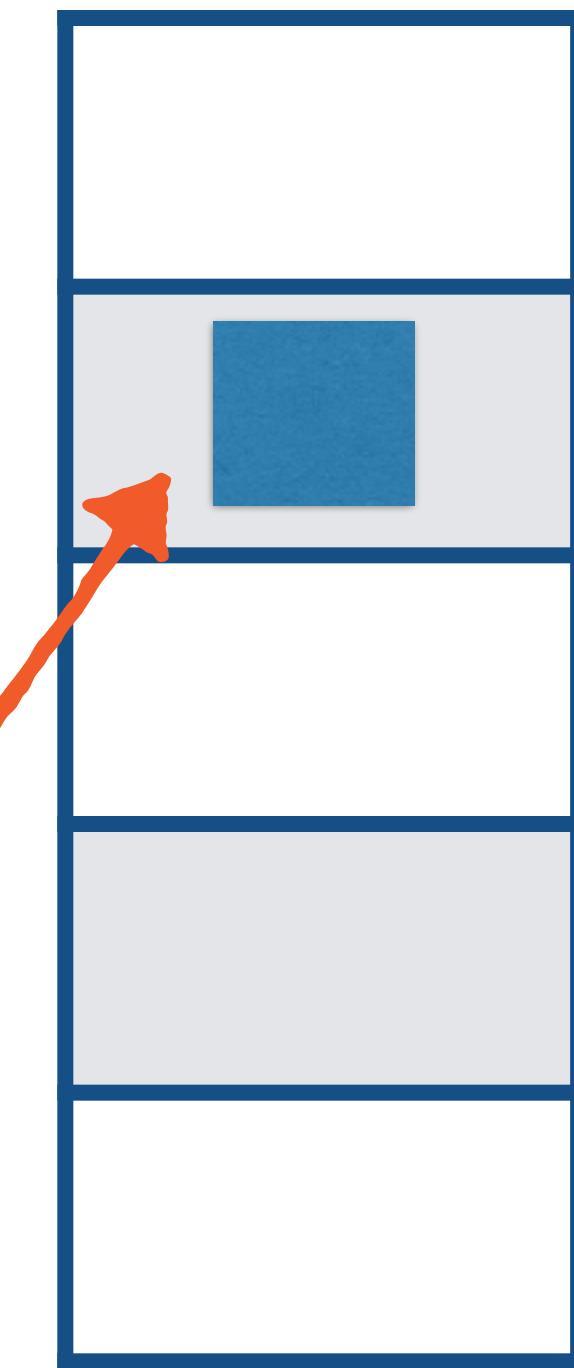
Rack 1



Rack 2



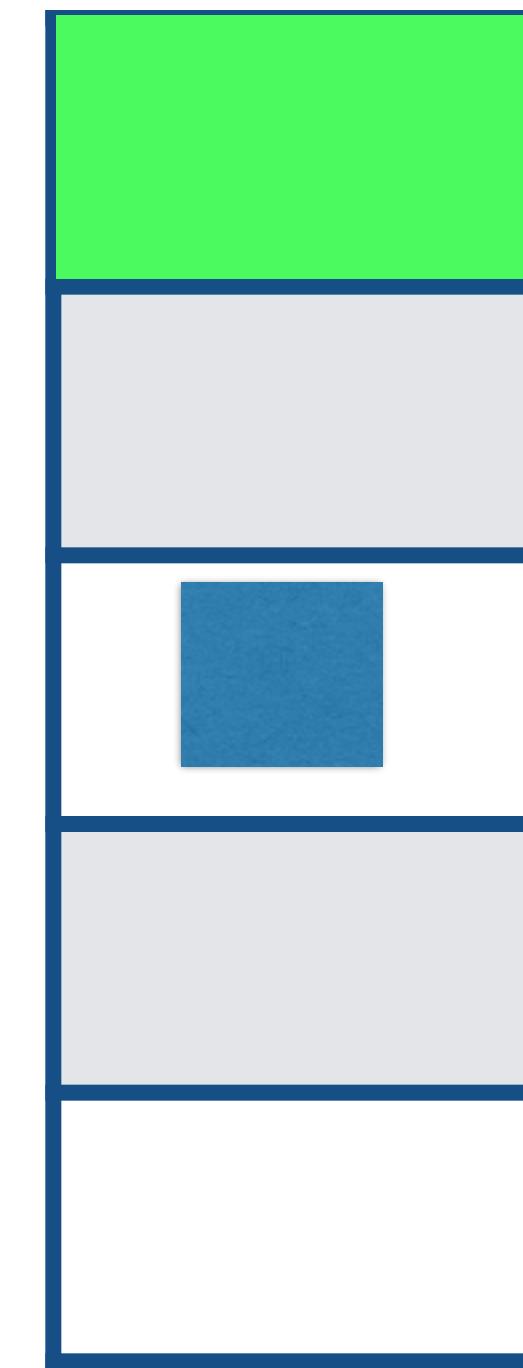
Rack 3



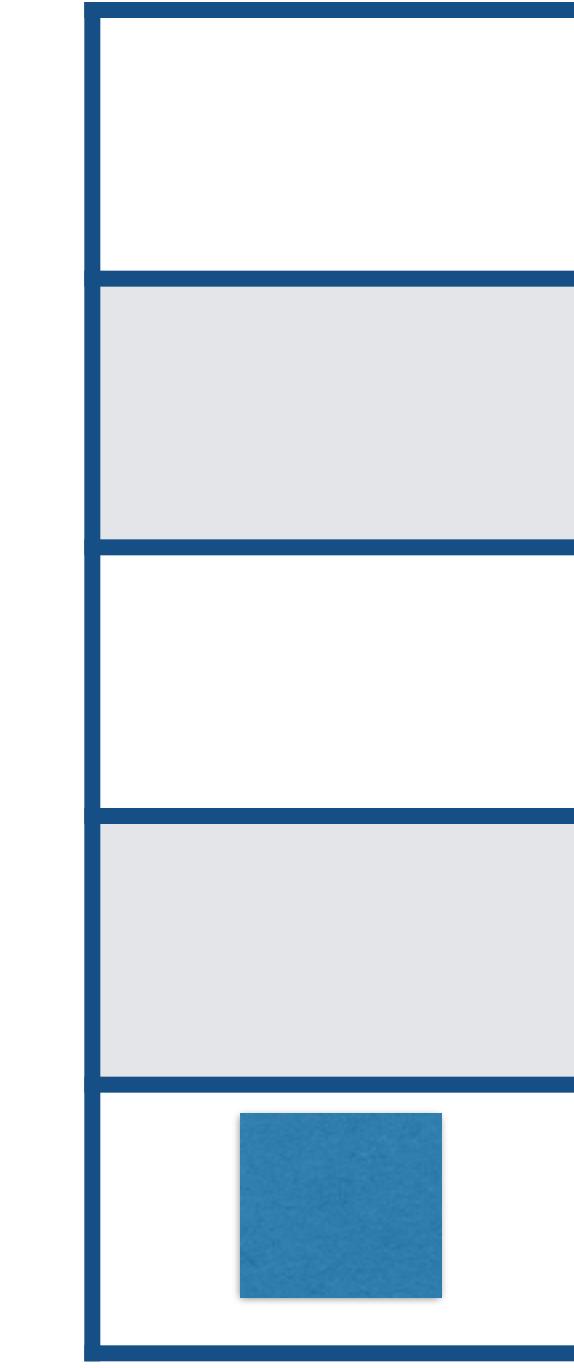
Forwarded further to
the next replica
location

Minimize
write
bandwidth

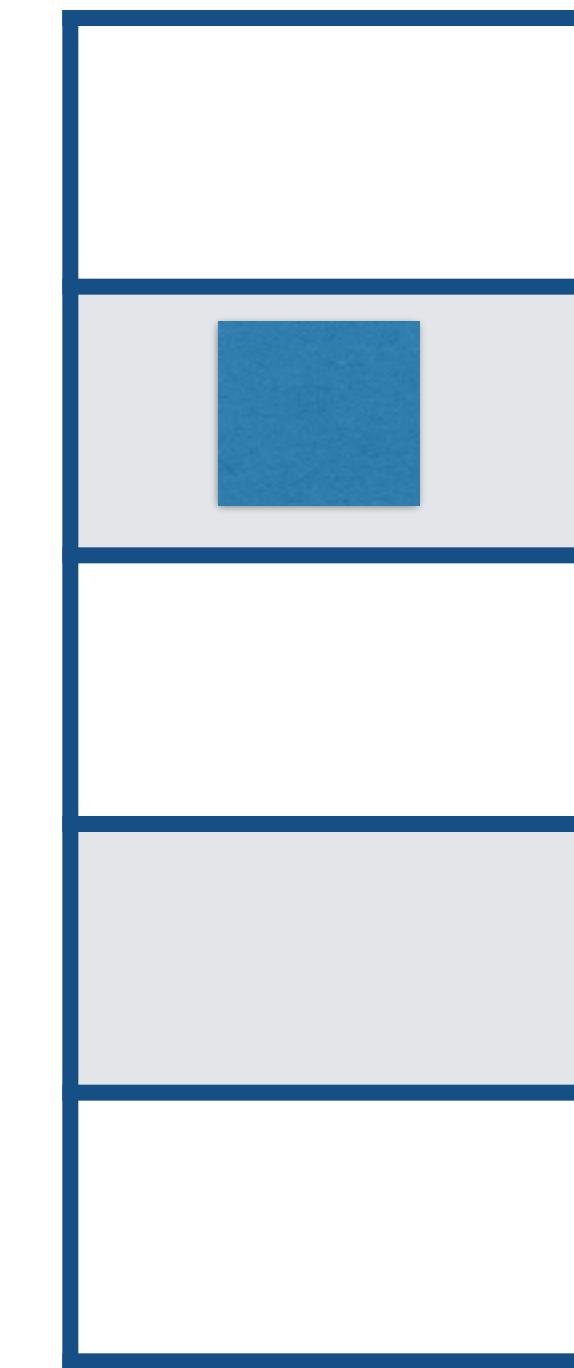
Rack 1



Rack 2



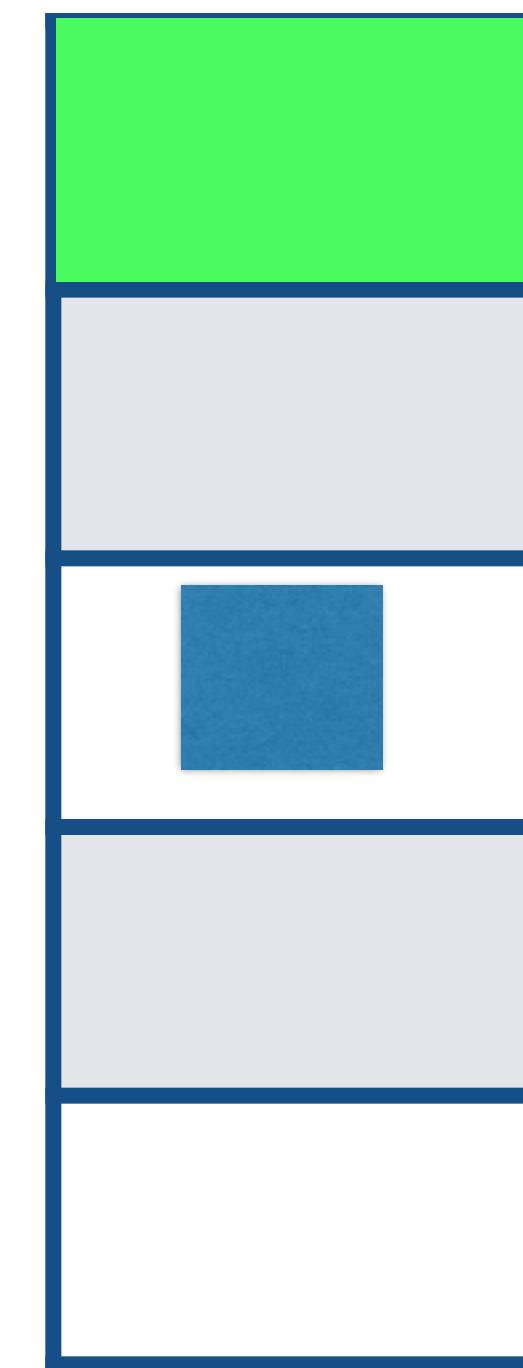
Rack 3



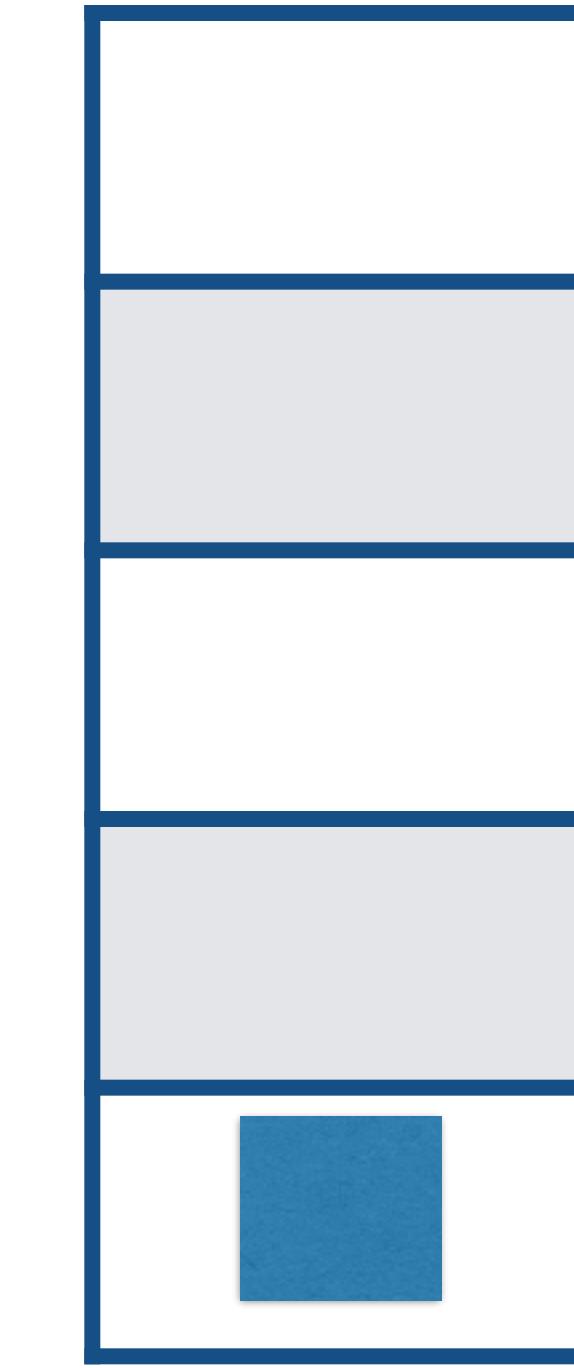
Forwarding requires
a large amount of
bandwidth

Minimize
write
bandwidth

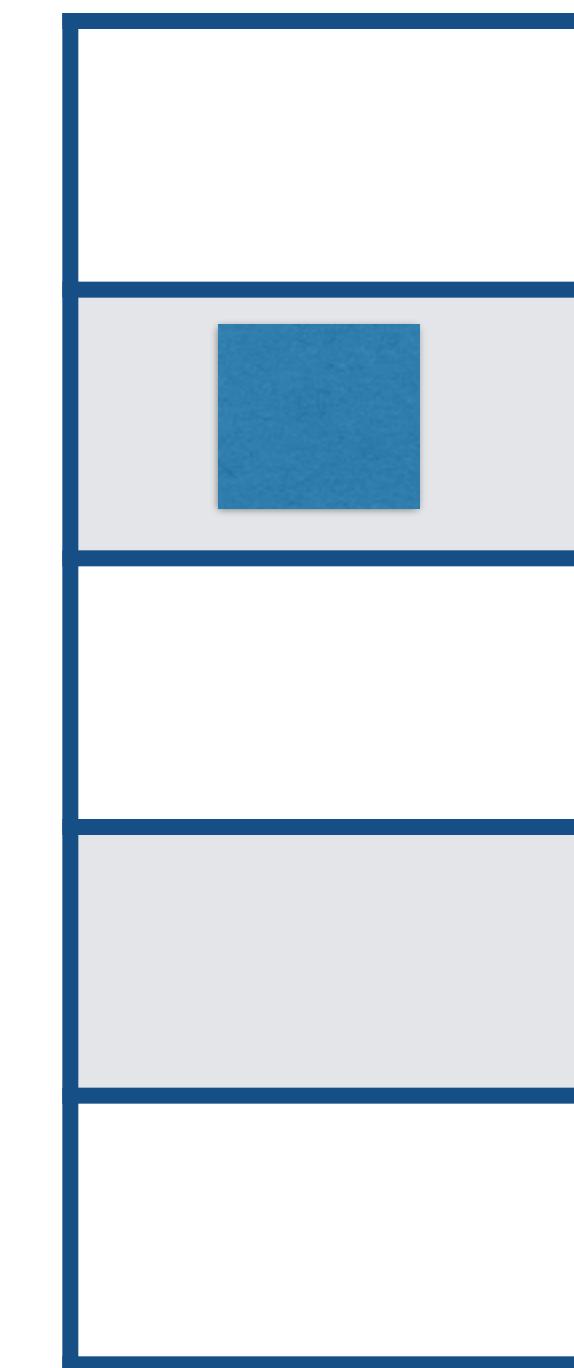
Rack 1



Rack 2



Rack 3



Increases the cost
of write operations

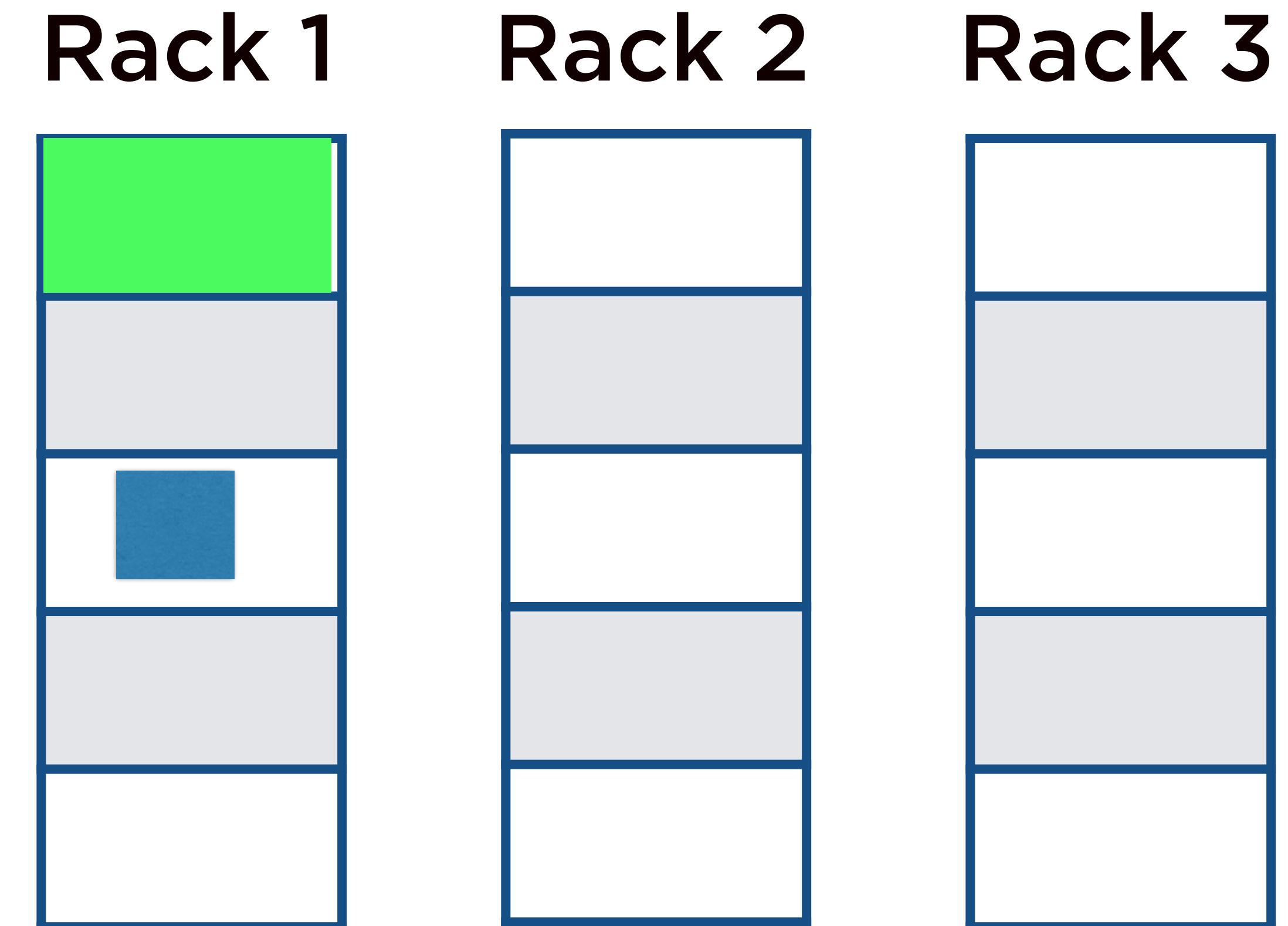
Default Hadoop Replication Strategy



Balancing both needs

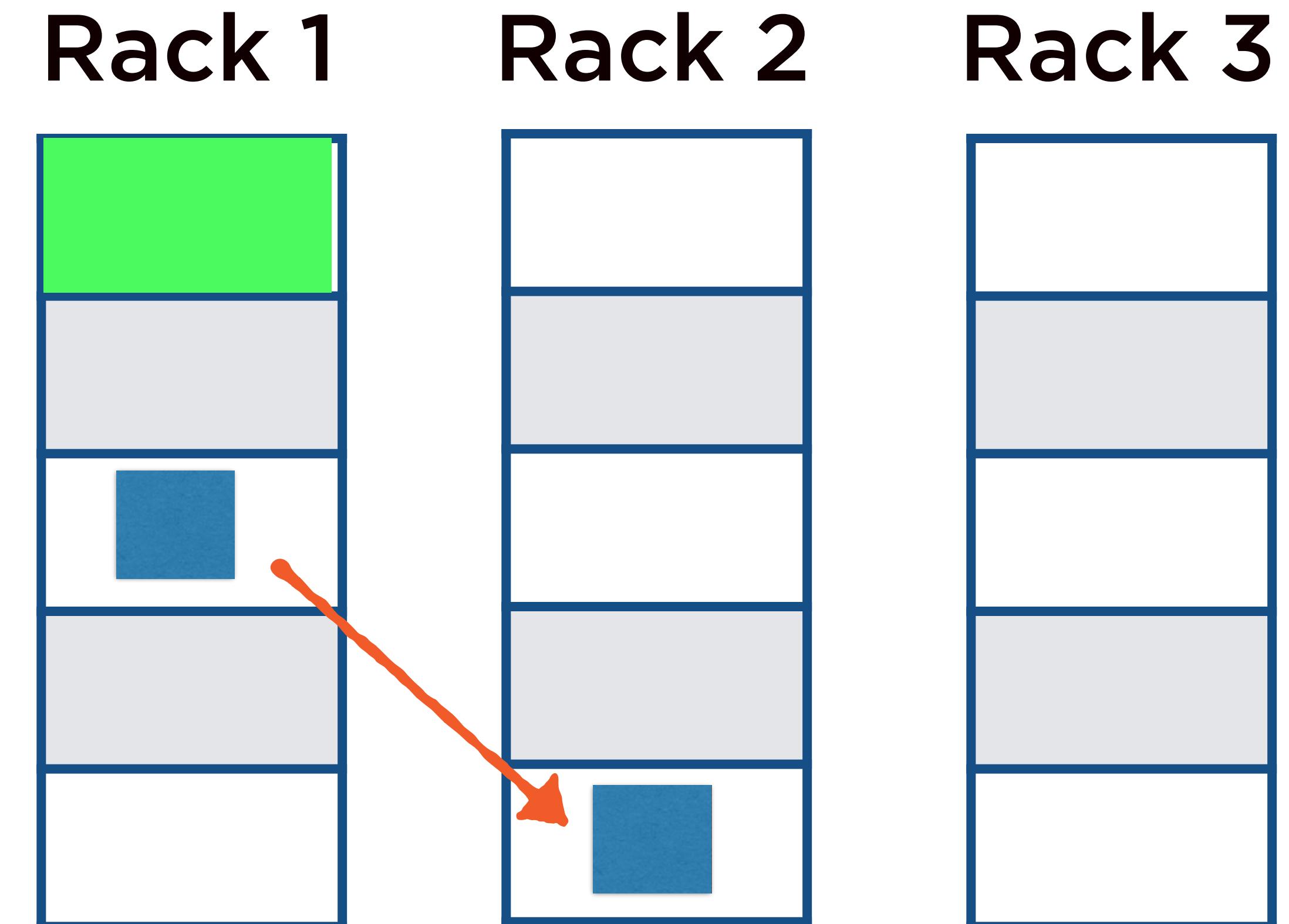
Default Hadoop Replication Strategy

**First location
chosen at random**



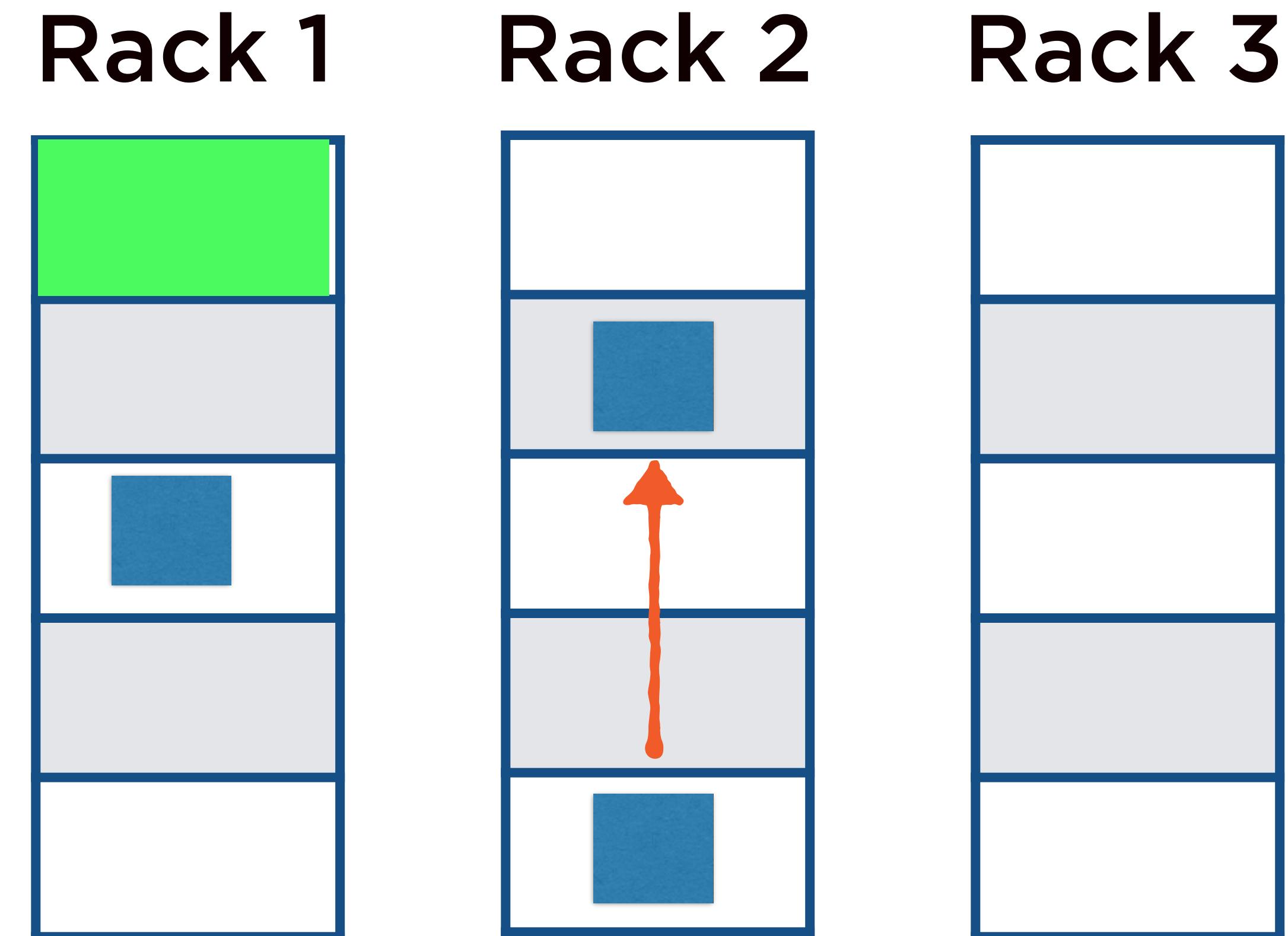
Default Hadoop Replication Strategy

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



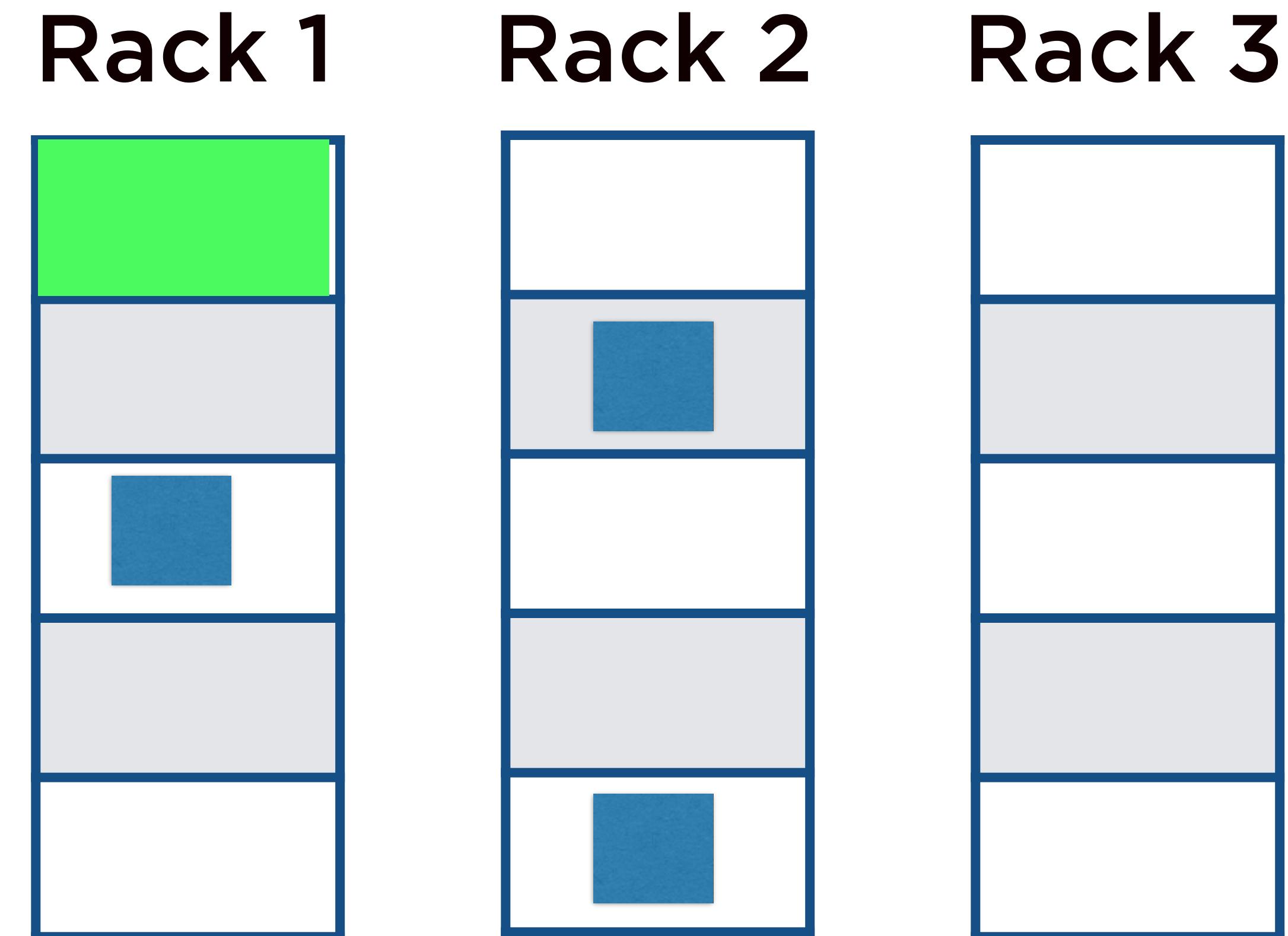
Default Hadoop Replication Strategy

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



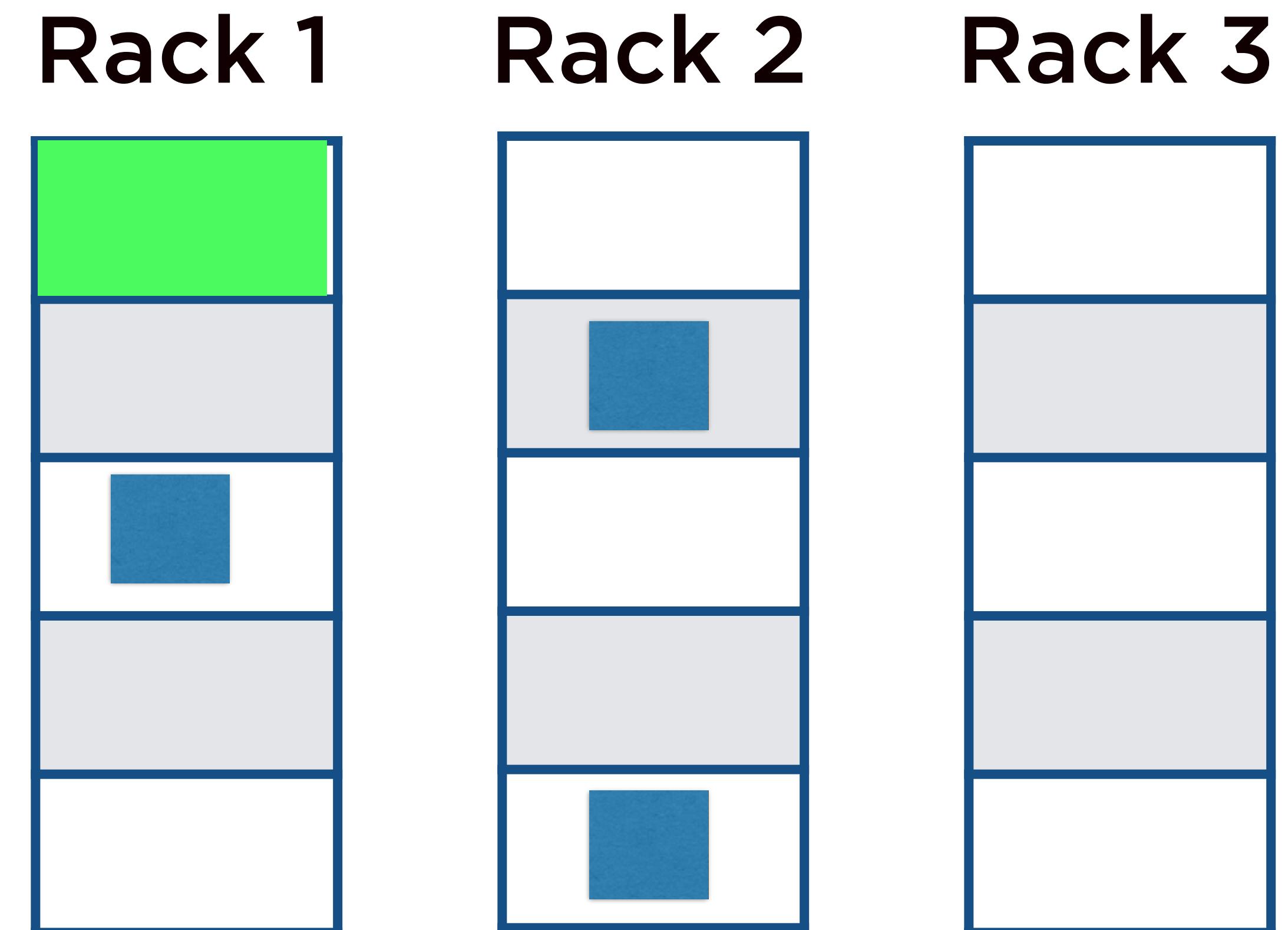
Default Hadoop Replication Strategy

Reduces inter-rack traffic and improves write performance



Default Hadoop Replication Strategy

Read operations
are sent to the
rack closest to
the client

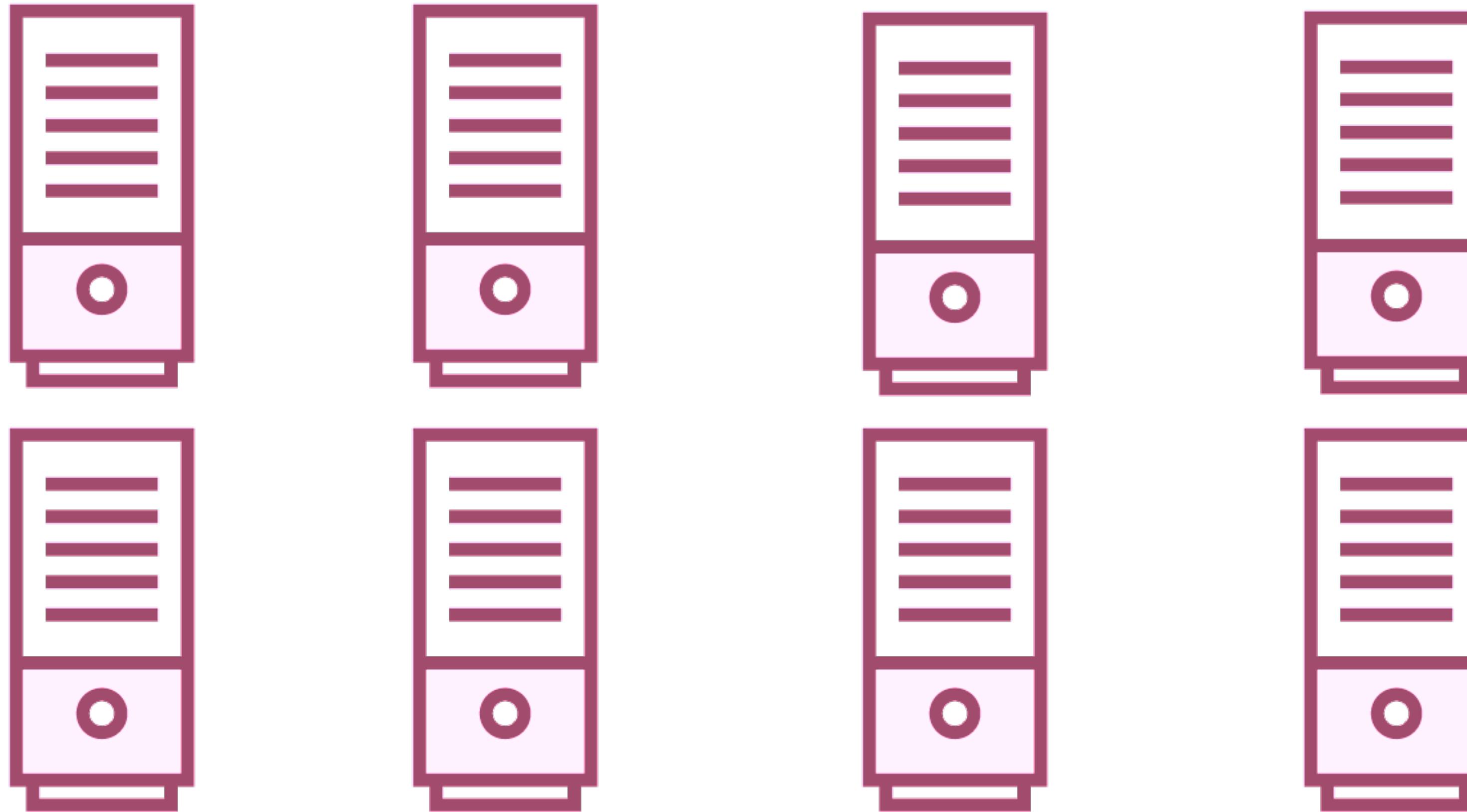


Understanding MapReduce

MapReduce

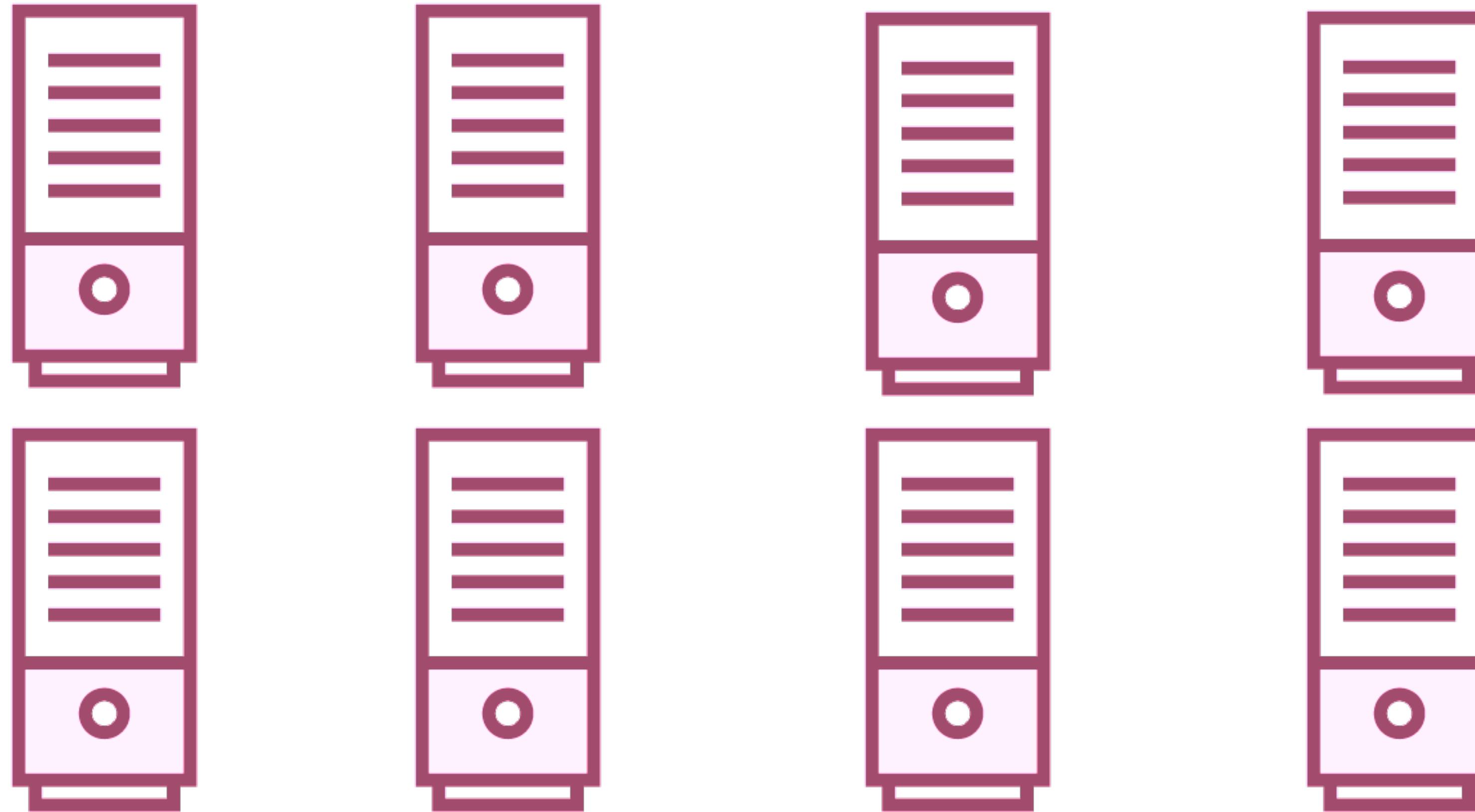
Processing huge amounts of data

MapReduce



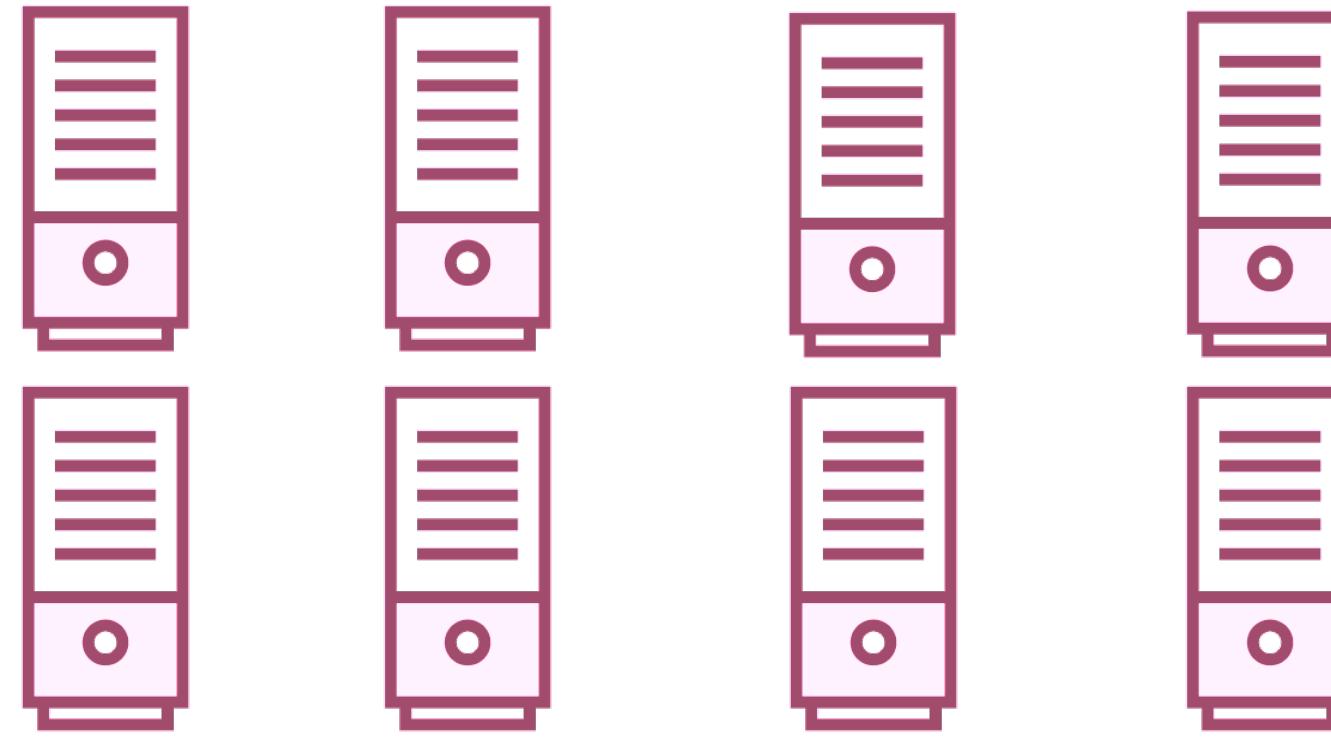
**Requires running processes on many
machines**

MapReduce



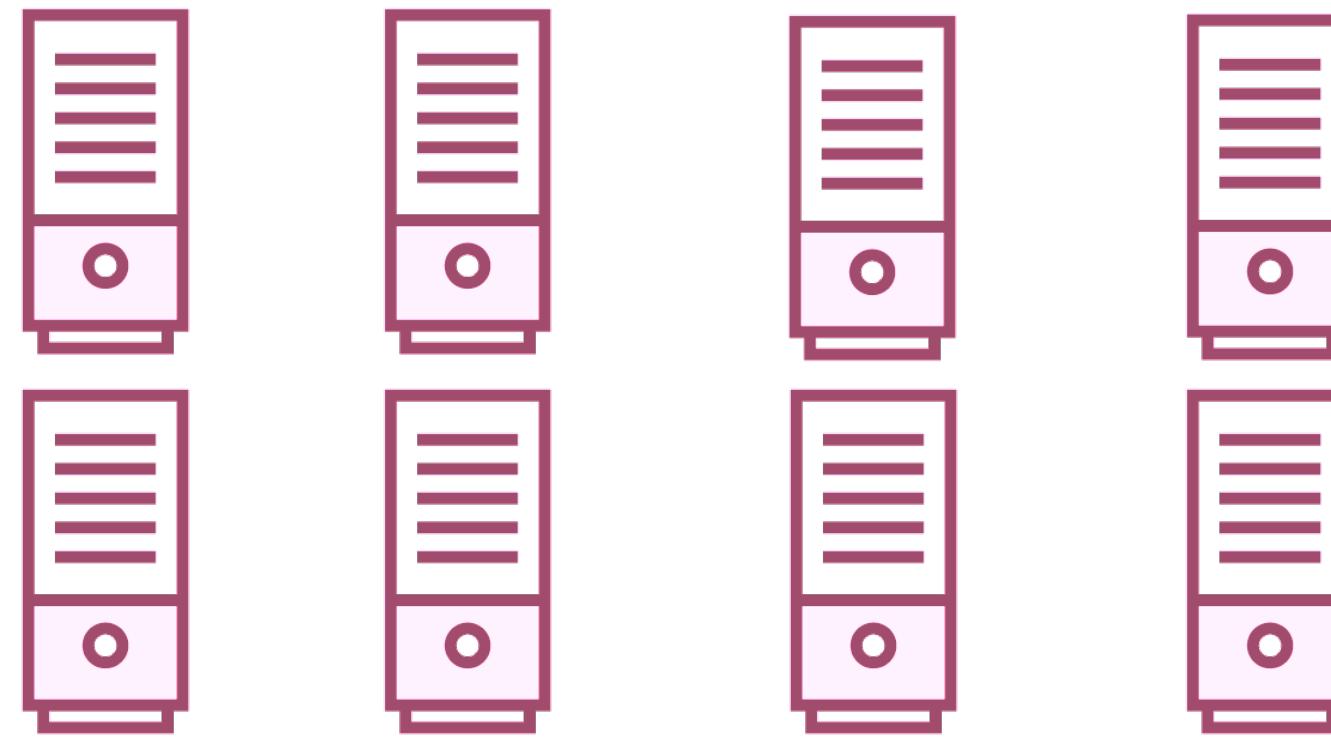
A distributed system

MapReduce



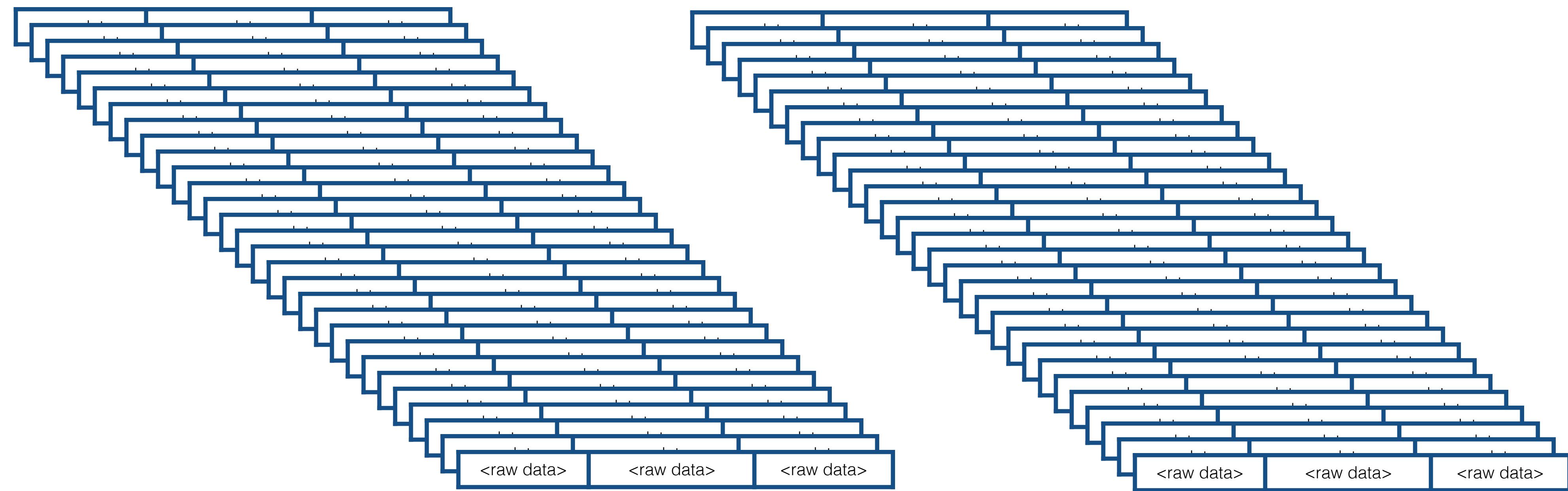
**MapReduce is a programming
paradigm**

MapReduce



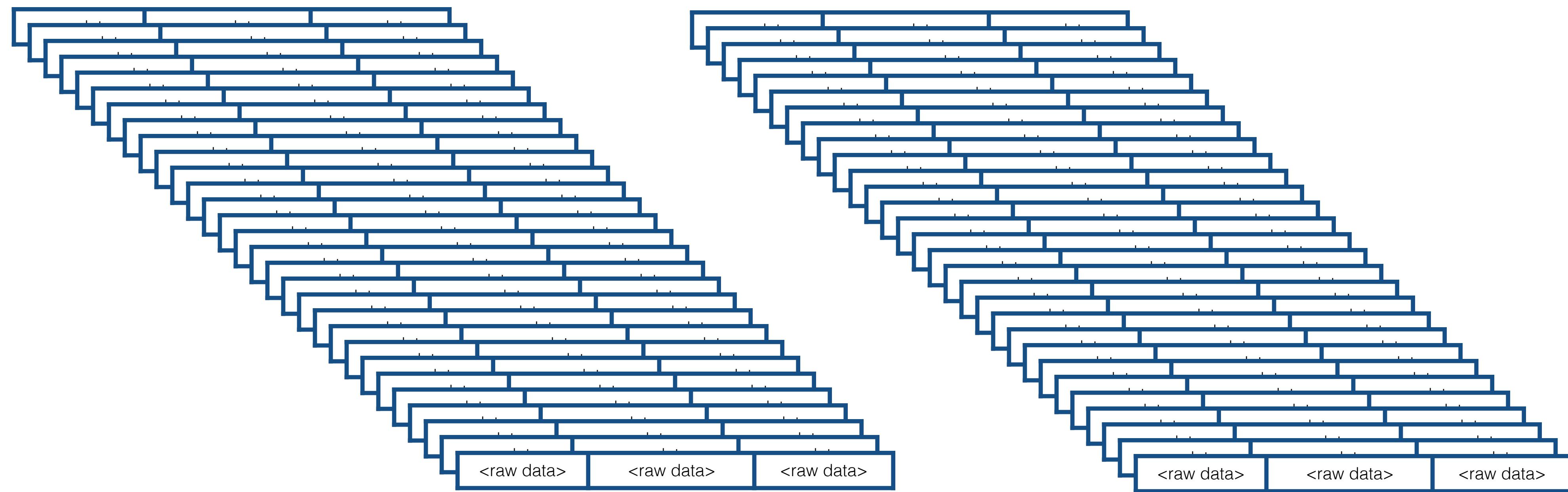
**Takes advantage of the inherent
parallelism in data processing**

MapReduce



Modern systems generate millions of records of raw data

MapReduce



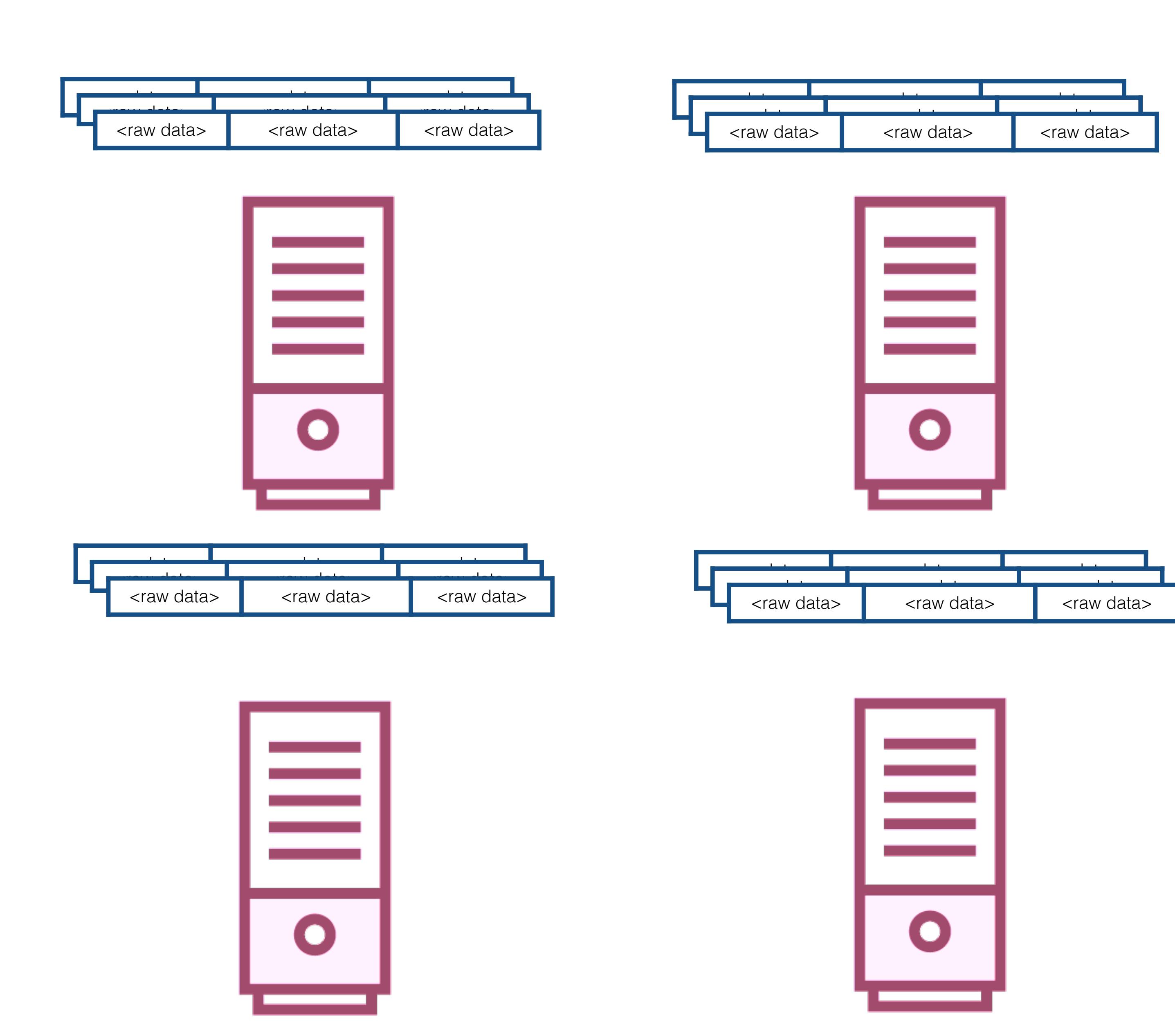
A task of this scale is processed in

two stages

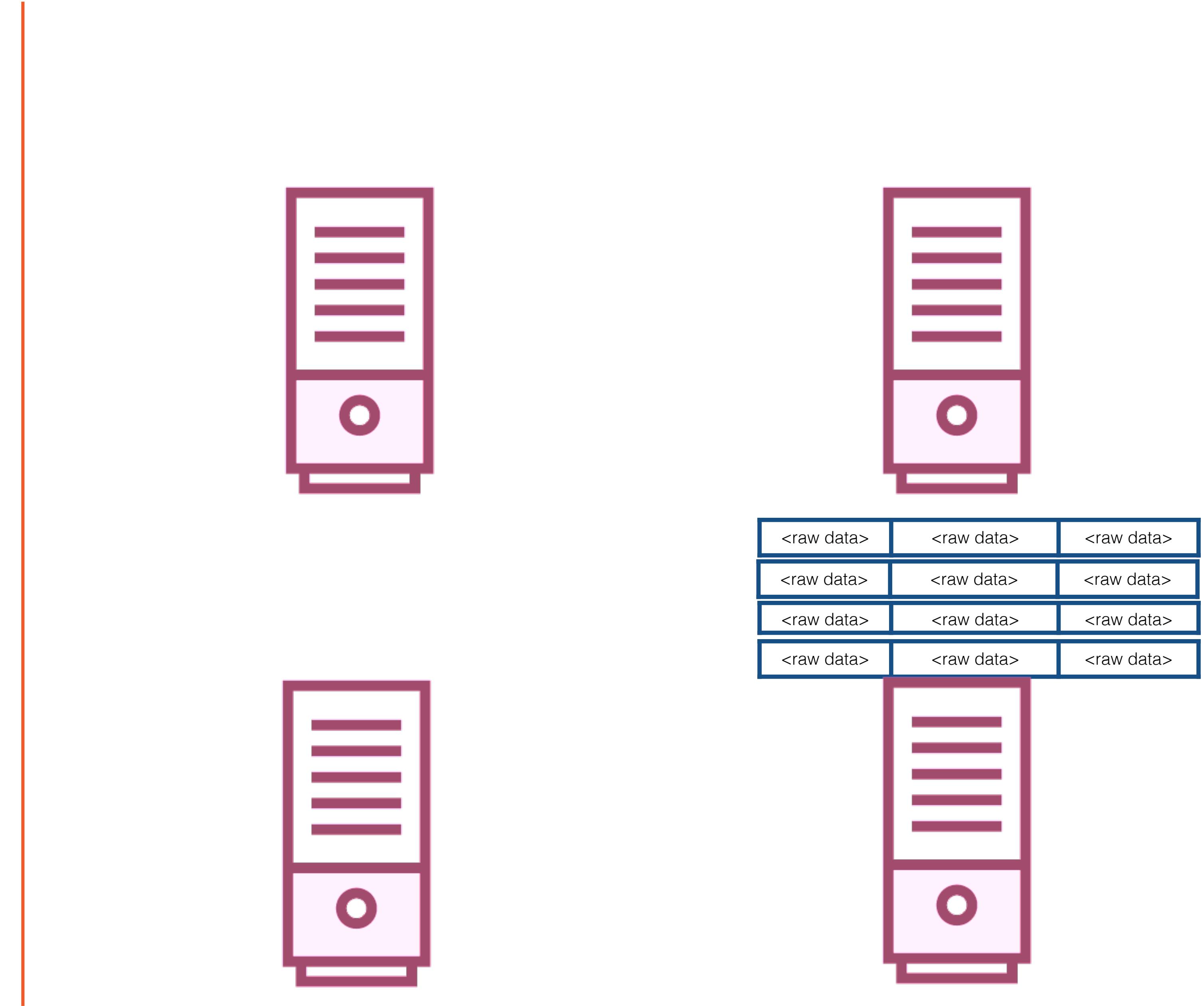
map

reduce

map



reduce



MapReduce

map reduce

The programmer defines
these 2 functions

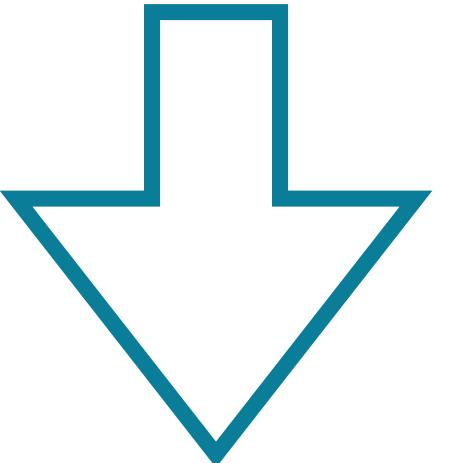
Hadoop does the rest -
behind the scenes

map

An operation performed in parallel, on small portions of the dataset

map

One Record

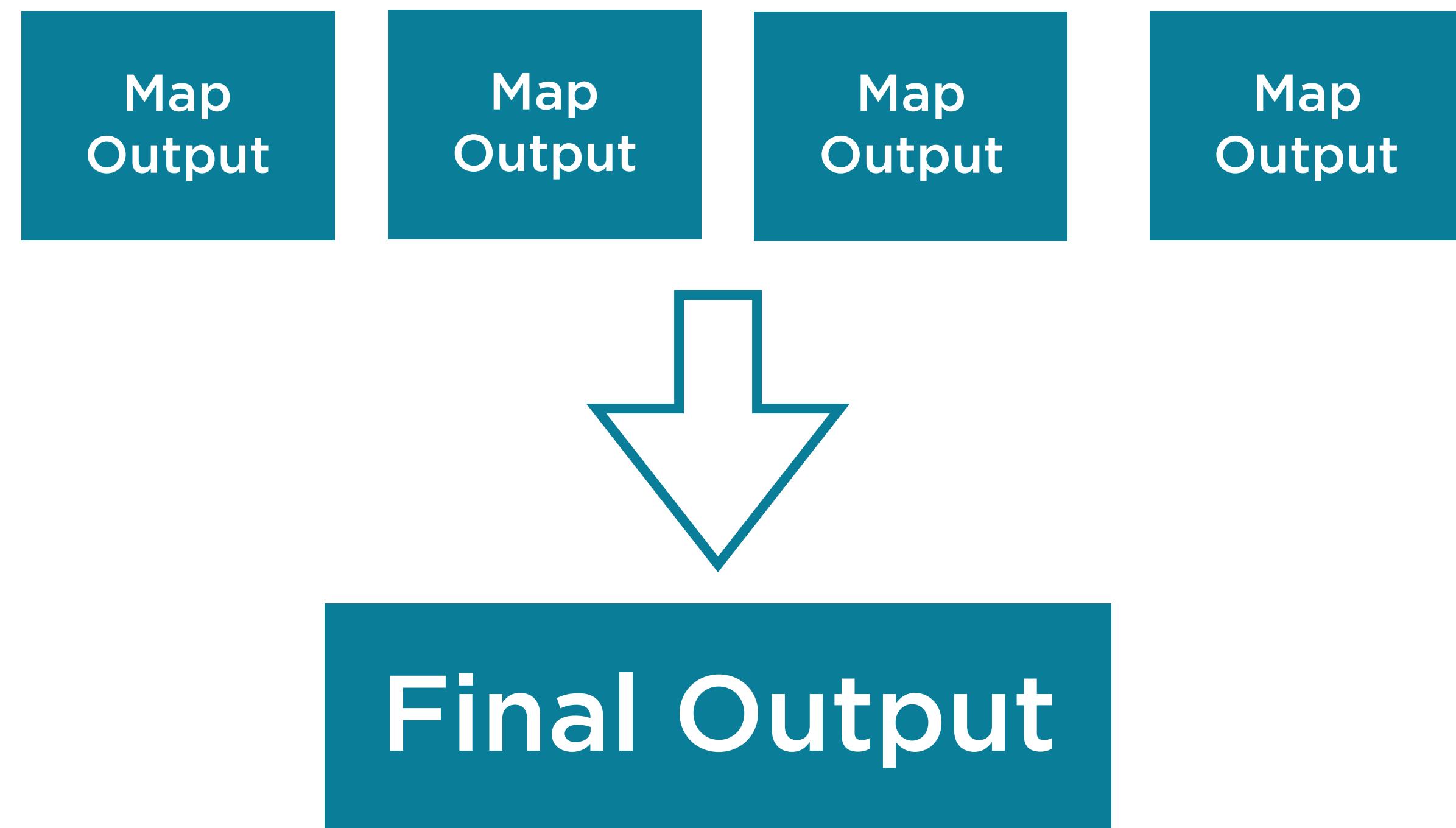


Key-Value Output

reduce

An operation to
combine the results of
the map step

reduce



map

A step that can be performed in parallel

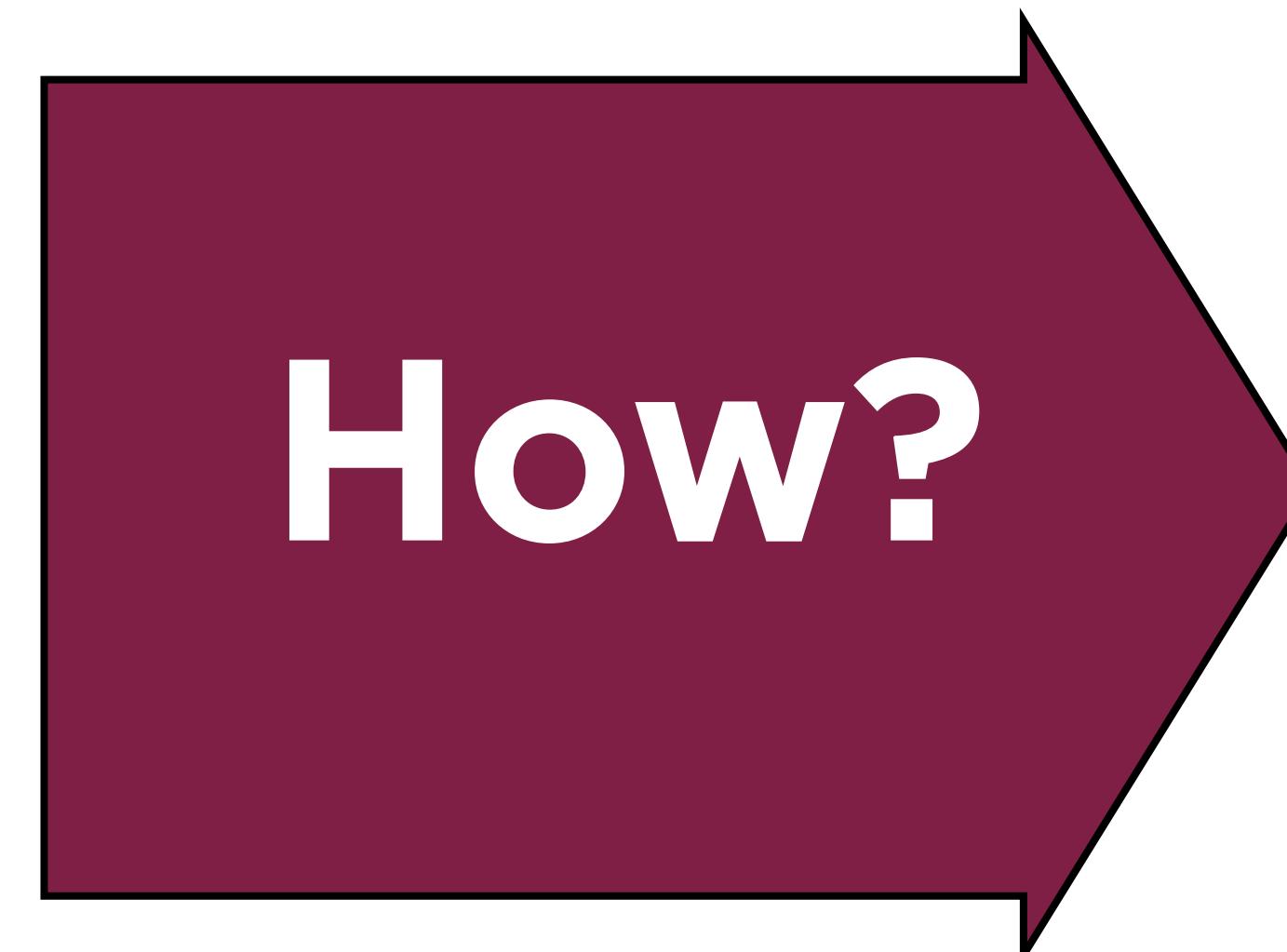
reduce

A step to combine the intermediate results

Counting Word Frequencies

Consider a large text file

Twinkle twinkle little star
How I wonder what you are
Up above the world so high
Like a diamond in the sky
Twinkle twinkle little star
How I wonder what you are
.....



Word	Frequency
above	14
are	20
how	21
star	22
twinkle	32
...	...

MapReduce Flow

Twinkle twinkle little star

How I wonder what you are

Up above the world so high

Like a diamond in the sky

Twinkle twinkle little star

How I wonder what you are

.....

**The raw data is really large
(potentially in PetaBytes)**

**It's distributed across many
machines in a cluster**

**Each machine holds a partition of
data**

MapReduce Flow

Twinkle twinkle little star
How I wonder what you are



Up above the world so high
Like a diamond in the sky



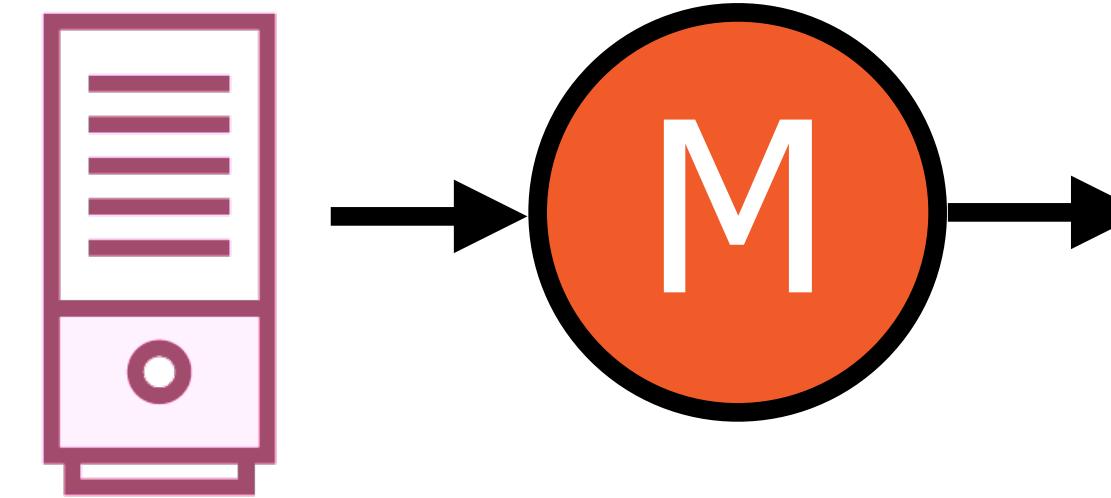
Twinkle twinkle little star
How I wonder what you are



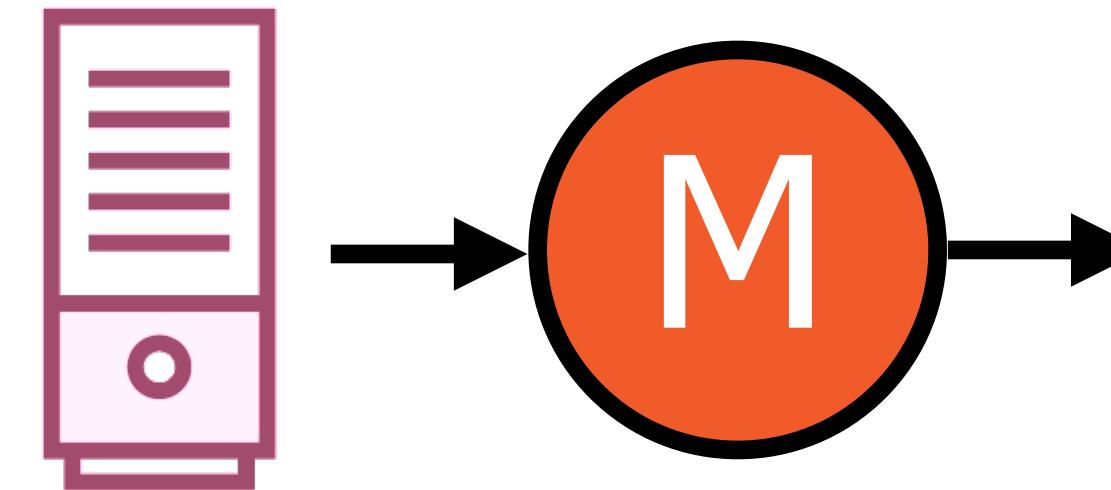
Each partition is given to a different process i.e. to mappers

MapReduce Flow

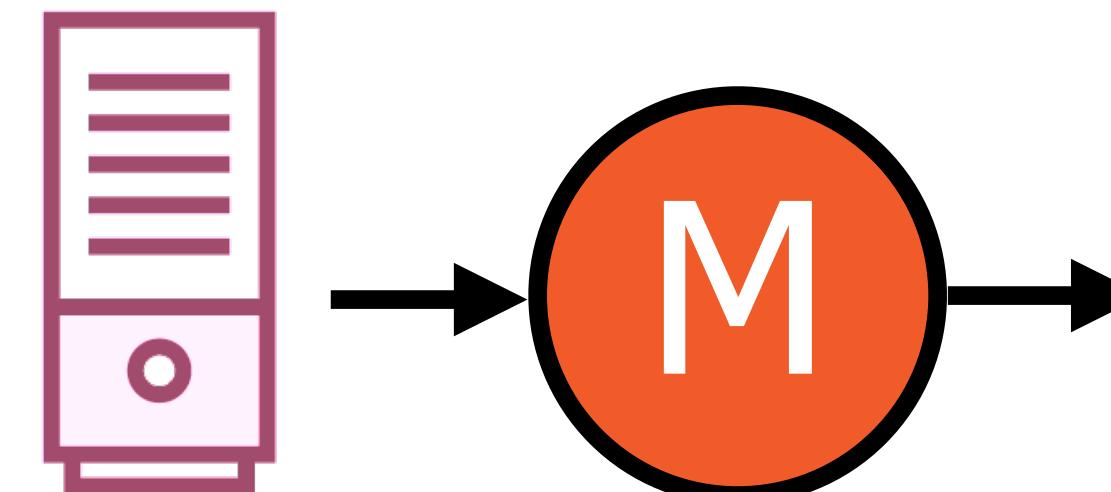
Twinkle twinkle little star
How I wonder what you are



Up above the world so high
Like a diamond in the sky



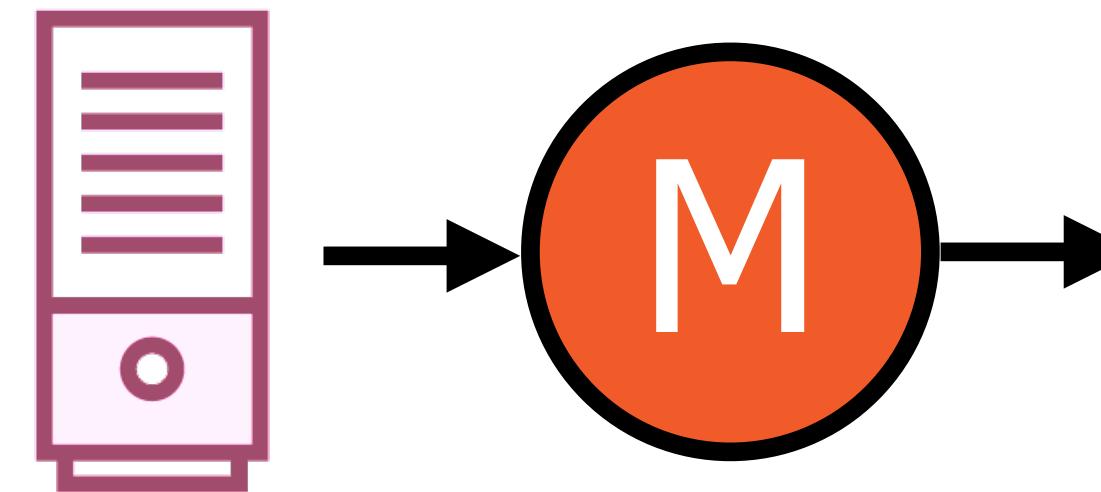
Twinkle twinkle little star
How I wonder what you are



Each mapper works in parallel

Map Flow

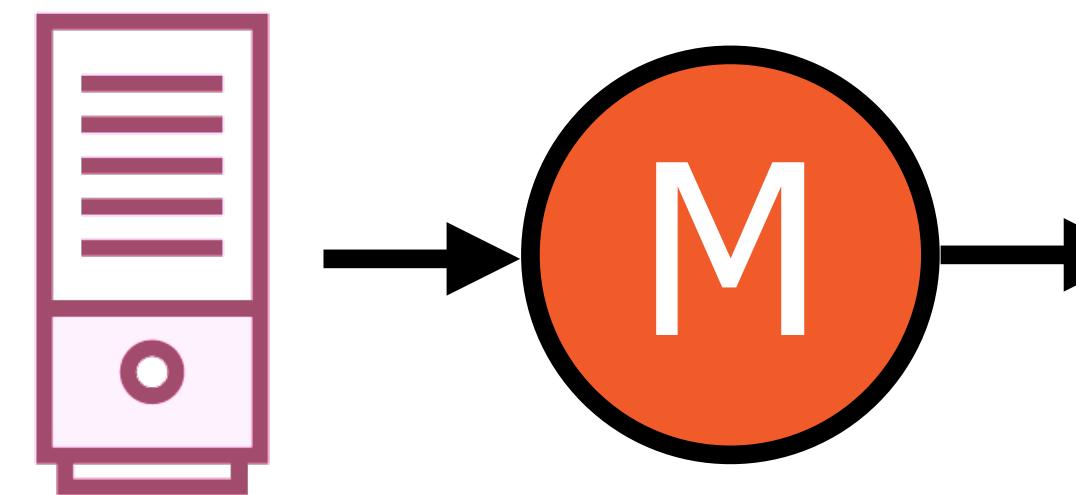
```
Twinkle twinkle little star  
How I wonder what you are
```



**Within each mapper, the rows
are processed serially**

Map Flow

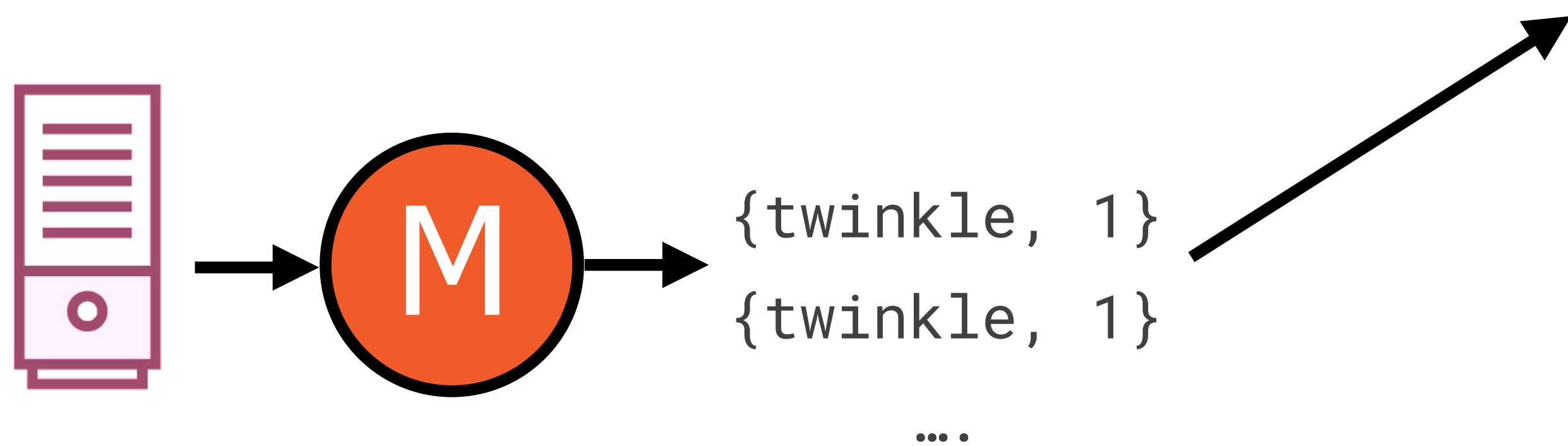
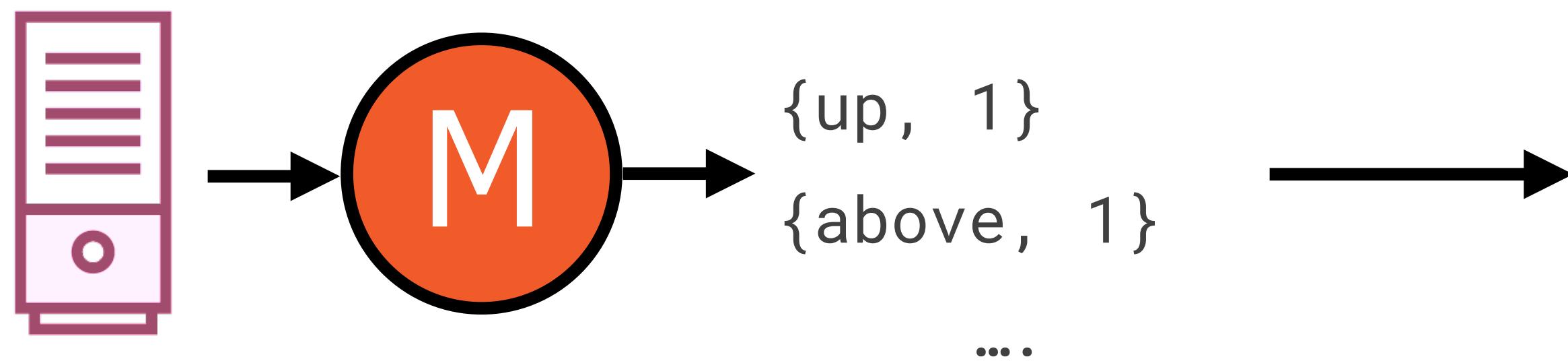
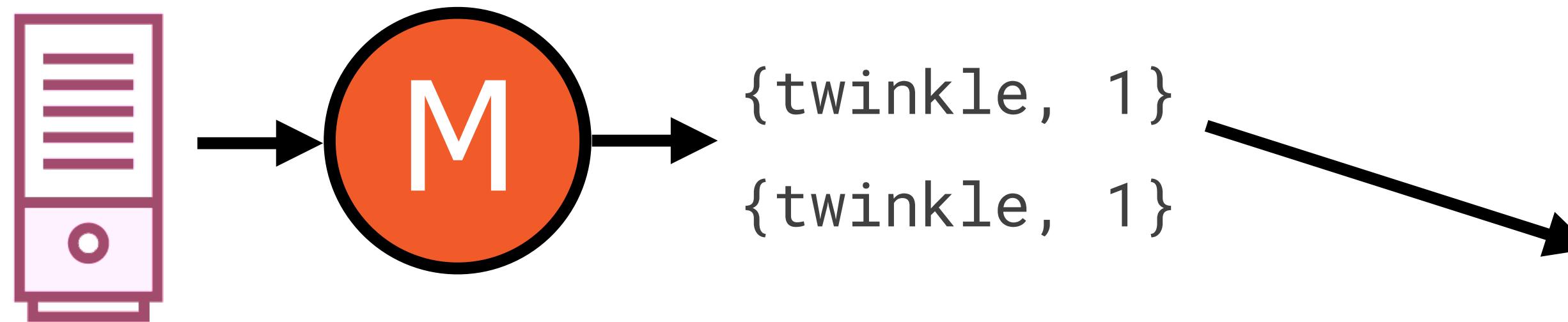
```
Twinkle twinkle little star  
How I wonder what you are
```



Word	# Count
{twinkle, 1}	
{twinkle, 1}	
{little, 1}	
{star, 1}	

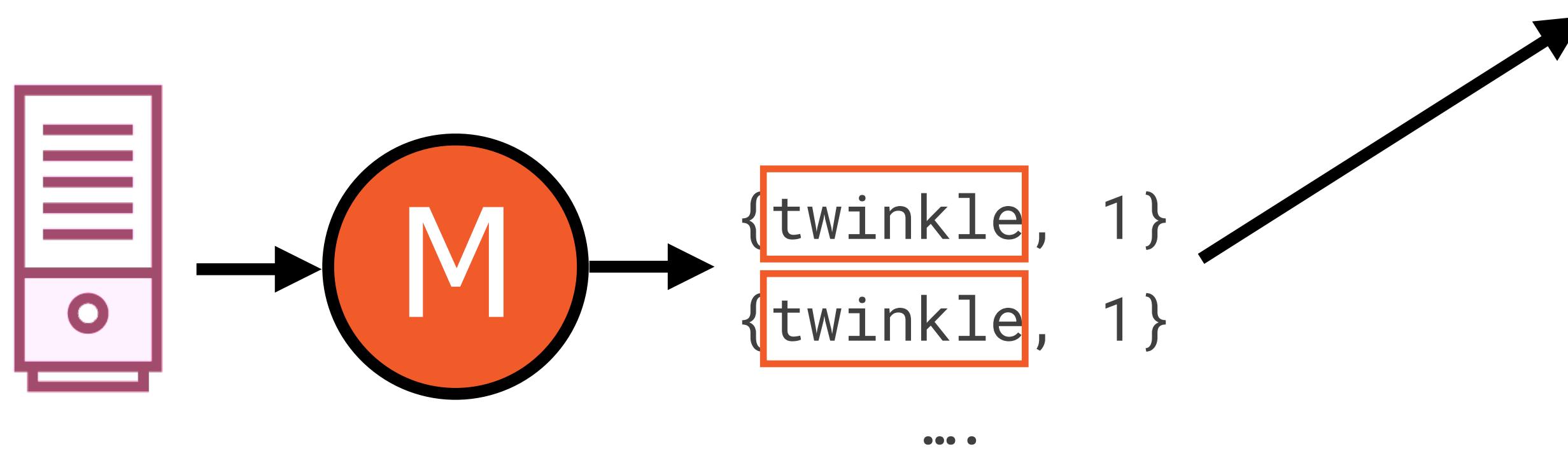
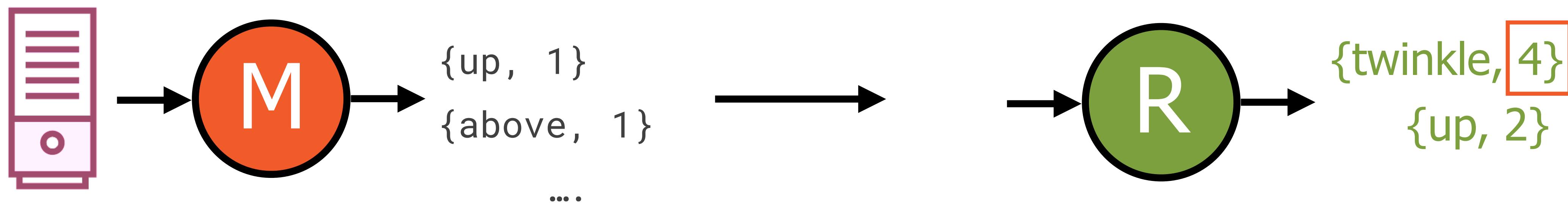
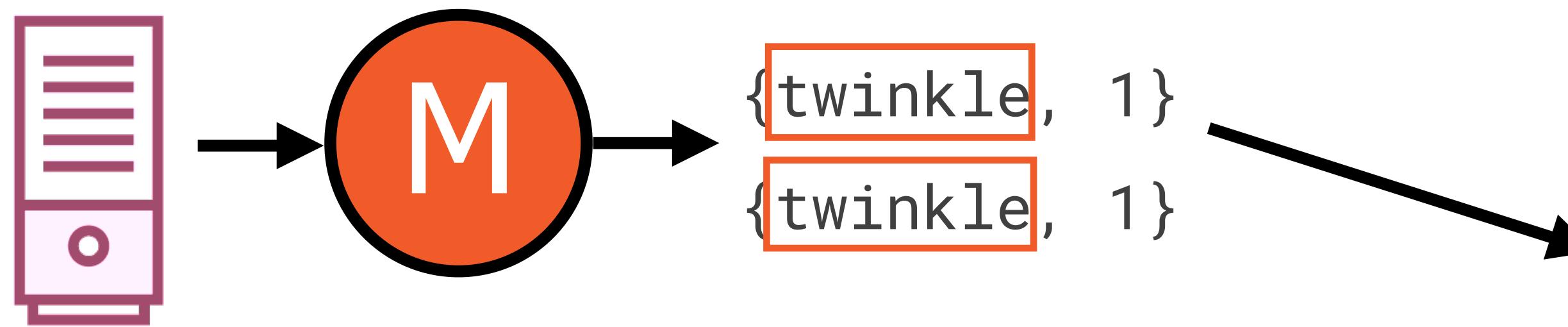
Each row emits {key, value} pairs

Reduce Flow



The results are
passed on to another
process i.e. a reducer

Reduce Flow



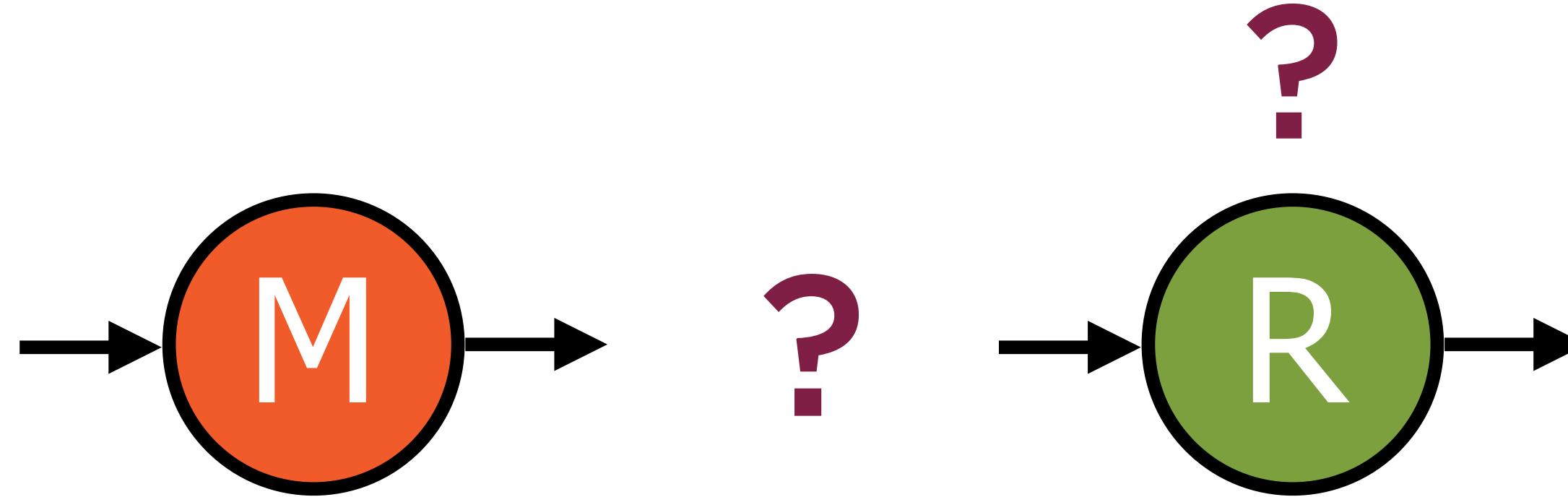
The reducer
combines the values
with the same key

Key Insight Behind MapReduce



Many data processing tasks can be expressed in this form

Answer Two Questions



1. What {key, value} pairs should be emitted in the map step?
2. How should values with the same key be combined?

Counting Word Frequencies

```
Twinkle twinkle little star  
How I wonder what you are  
Up above the world so high  
Like a diamond in the sky
```



For each word
in each line

{twinkle, 1}

{twinkle, 1}

{little, 1}

{star, 1}

...

...

Word	Count
twinkle	2
little	1
...	...
...	...
...	...
...	...



Answer these to
parallelize any task :)

Implementing in Java

Map

A class where the map logic is implemented

Reduce

A class where the reduce logic is implemented

Main

A driver program that sets up the job

Implementing in Java

Map

**A class where the
map logic is
implemented**

Reduce

**A class where the
reduce logic is
implemented**

Main

**A driver program
that sets up the
job**

Map Step

Map Class

Mapper Class

The map logic is implemented in a class that extends the Mapper Class

Map Step

Map Class

<input key type,
input value type,
output key type,
output value type>

Mapper Class

This is a generic
class, with 4
type parameters

Implementing in Java

Map

**A class where the
map logic is
implemented**

Reduce

**A class where the
reduce logic is
implemented**

Main

**A driver program
that sets up the
job**

Implementing in Java

Map

A class where the
map logic is
implemented

Reduce

A class where the
reduce logic is
implemented

Main

A driver program
that sets up the
job

Reduce Step

Reduce Class

Reducer Class

The reduce logic is implemented in a class that extends the Reducer Class

Reduce Step

Reduce Class

<input key type,
input value type,
output key type,
output value type>

Reducer Class

This is also a
generic class, with
4 type parameters

Matching Data Types

Map Class

output key type,
output value type>

Mapper Class

Reduce Class

<input key type,
input value type,

Reducer Class

The output types of the Mapper should
match the input types of the Reducer

Implementing in Java

Map

A class where the
map logic is
implemented

Reduce

A class where the
reduce logic is
implemented

Main

A driver program
that sets up the
job

Implementing in Java

Map

A class where the
map logic is
implemented

Reduce

A class where the
reduce logic is
implemented

Main

**A driver program
that sets up the
job**

Setting up the Job

The Mapper and Reducer classes are used by a Job that is configured in the Main Class

Main Class

Job Object

Setting up the Job

The Job has a bunch of properties that need to be configured

Main Class Job Object

- Input filepath
- Output filepath
- Mapper class
- Reducer class
- Output data types

YARN

Yet Another Resource Negotiator



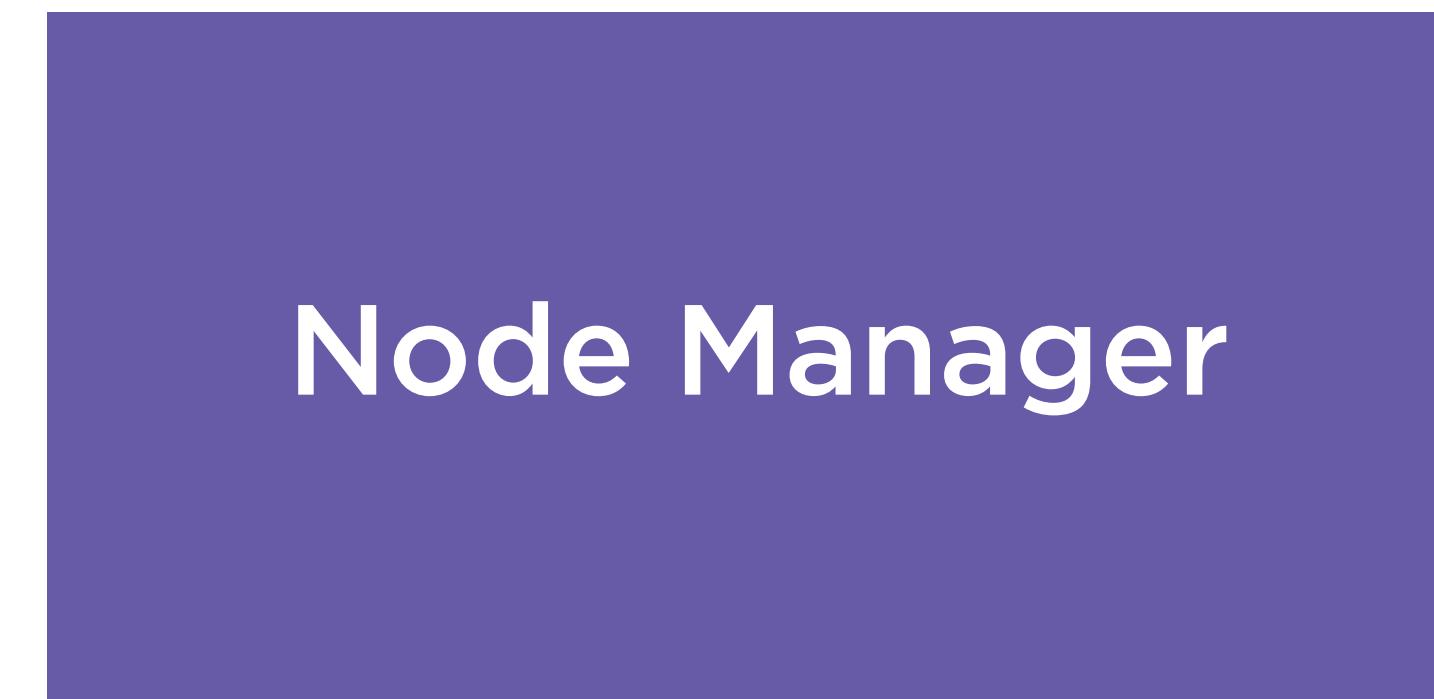
YARN



**Co-ordinates tasks running on
the cluster**

**Assigns new nodes in case of
failure**

YARN



Runs on a single master node

Schedules tasks across nodes

Run on all other nodes

Manages tasks on the individual node

YARN

ResourceManager

NodeManager

NodeManager

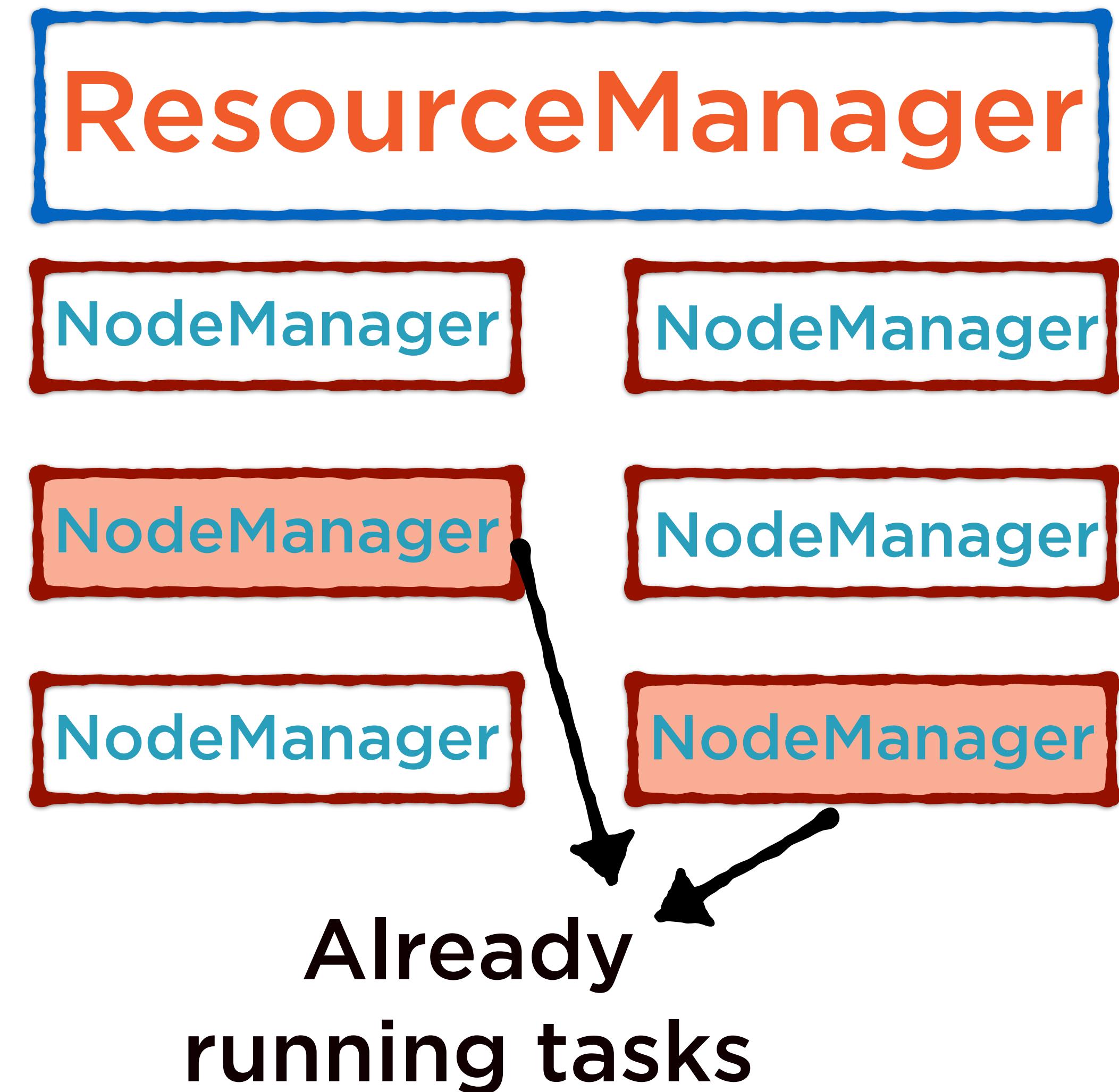
NodeManager

NodeManager

NodeManager

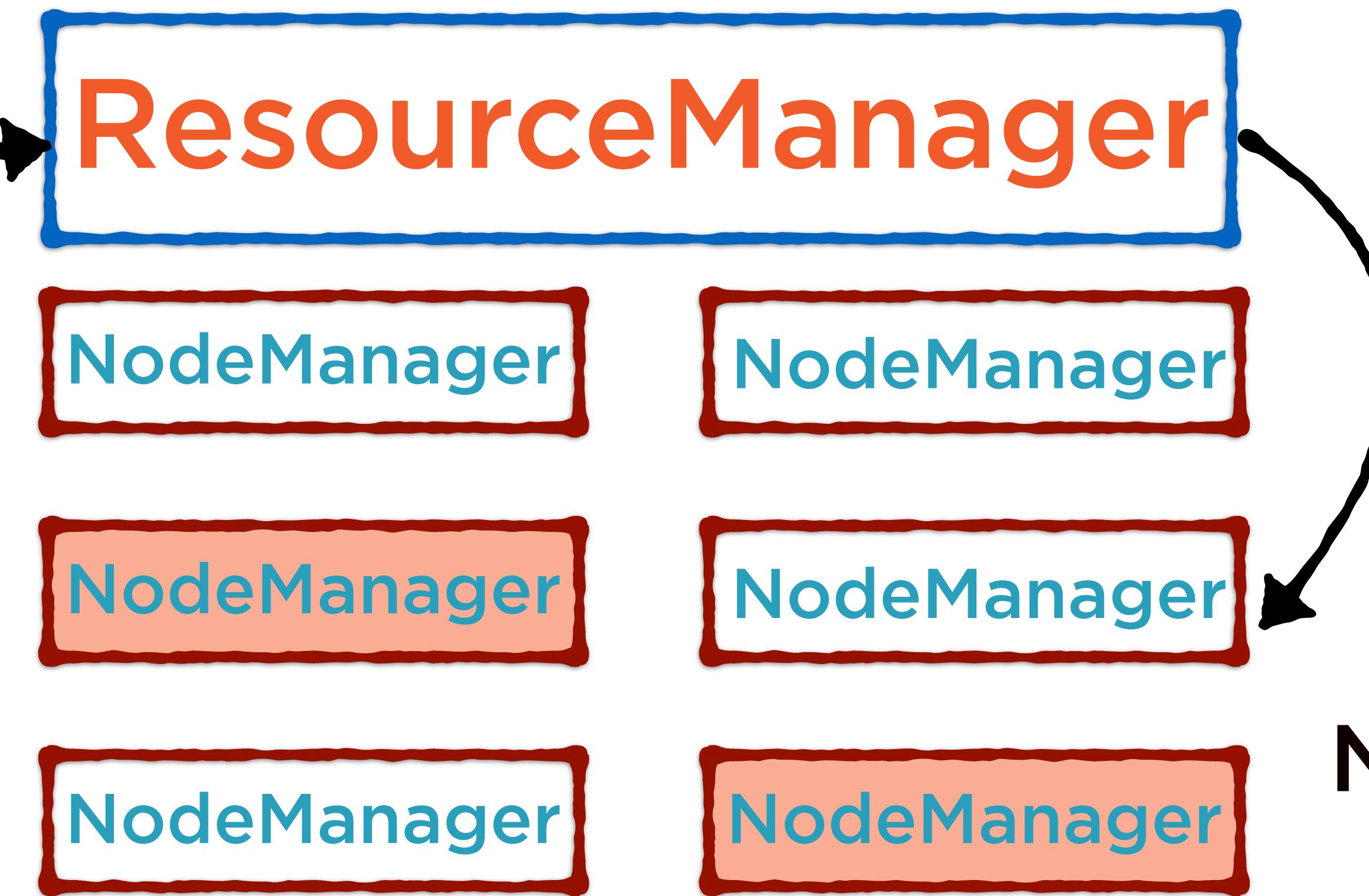
NodeManager

YARN



Job

Submitting a Job



Find a
NodeManager
with free
capacity

Application Master Process

All processes
on a node are
run within a
container

NodeManager

Container

Application Master Process

This is the logical unit for resources the process needs - memory, CPU etc

NodeManager

Container

Application Master Process

A container executes a specific application

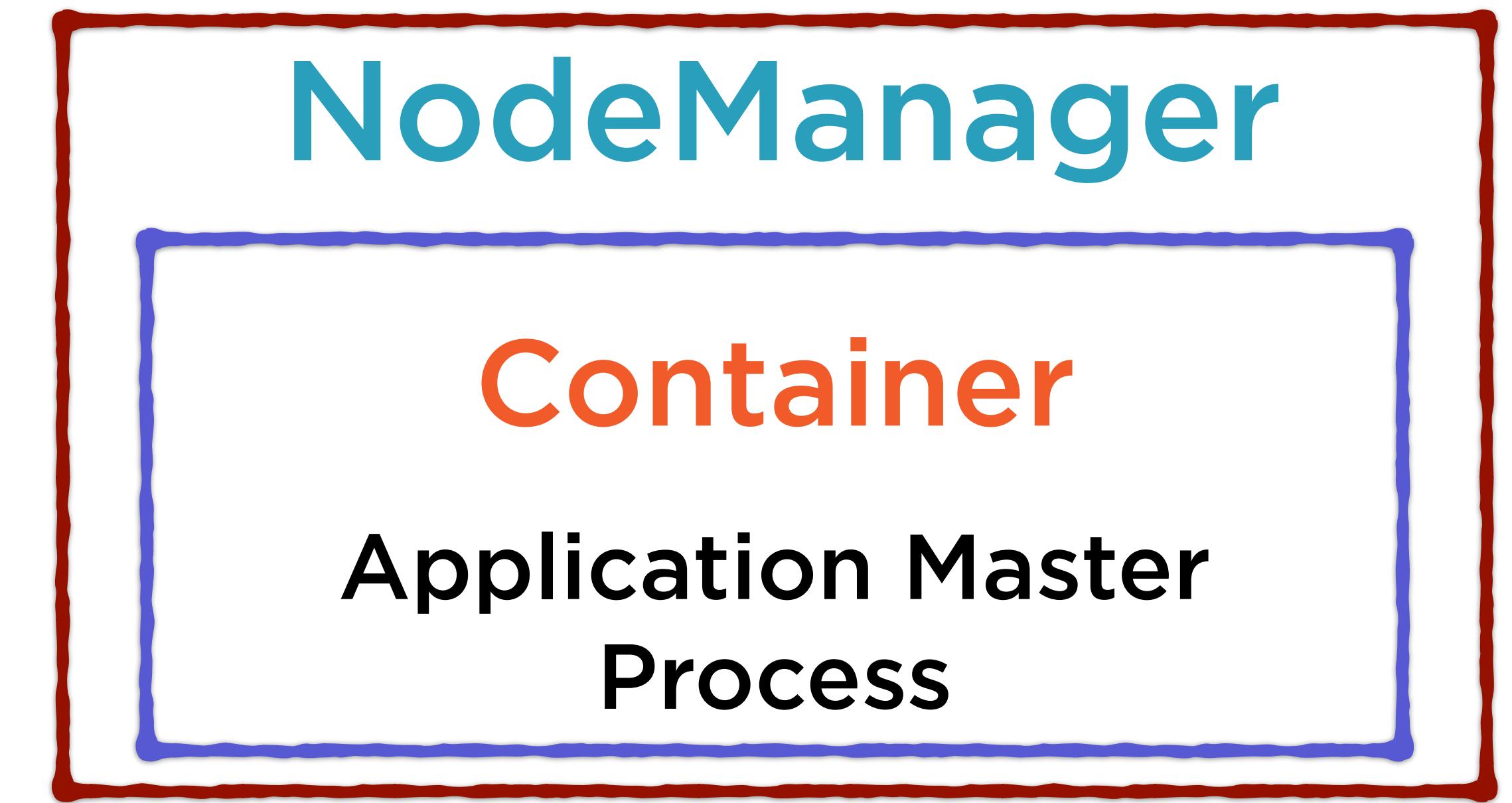
1 NodeManager can have multiple containers

NodeManager

Container

Application Master Process

The
ResourceManager
starts off the
Application Master
within the Container



Application Master Process

Performs the computation required for the task

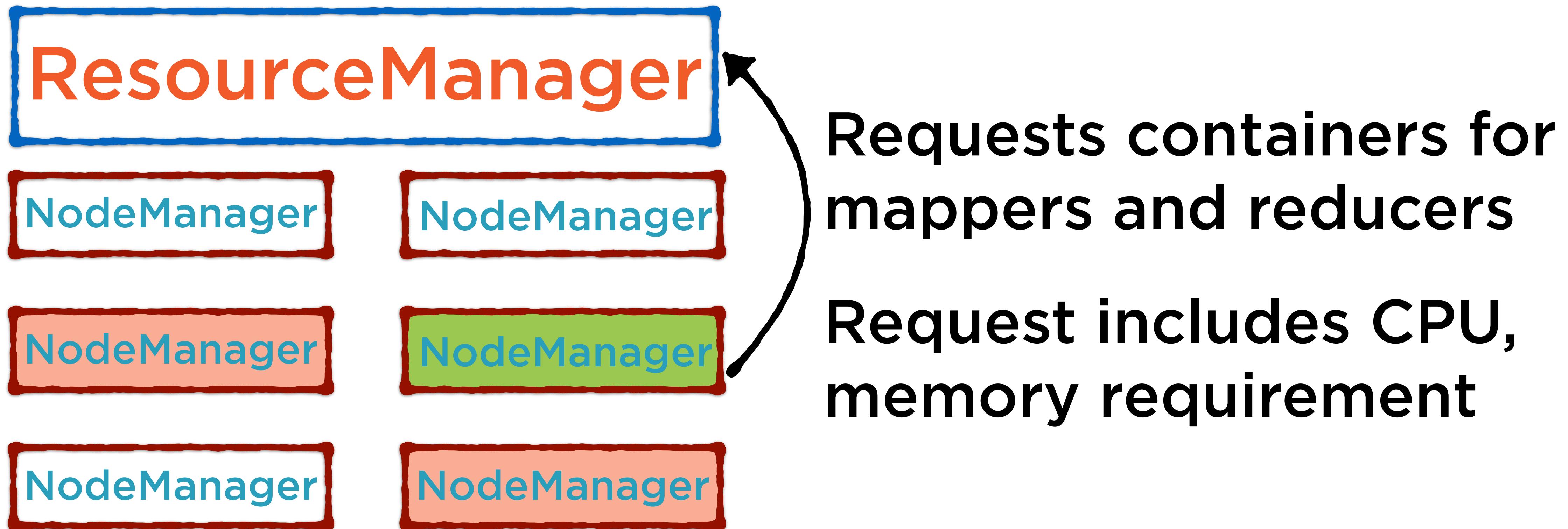
If additional resources are required, the Application Master makes the request

NodeManager

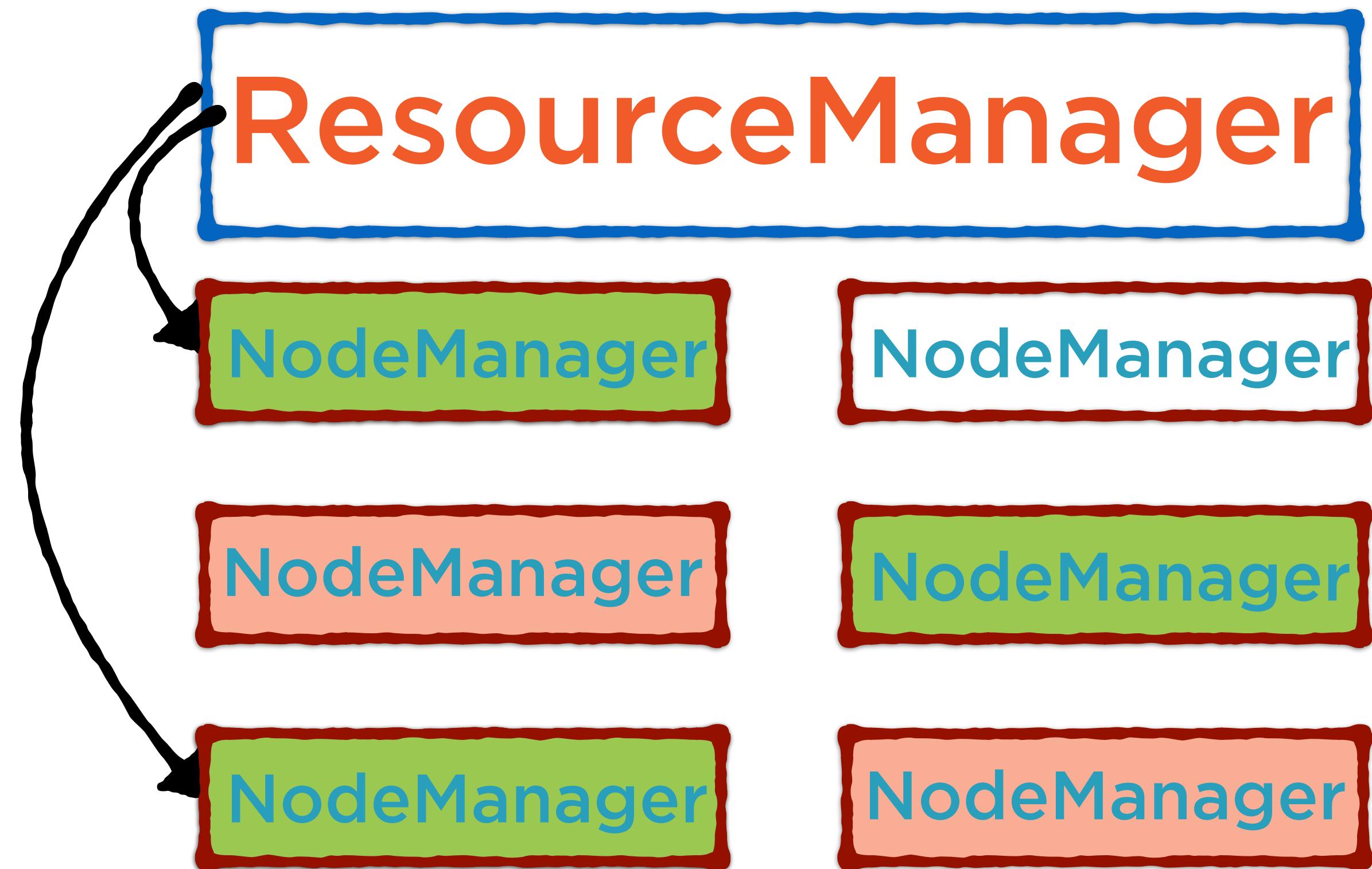
Container

**Application Master
Process**

Application Processes

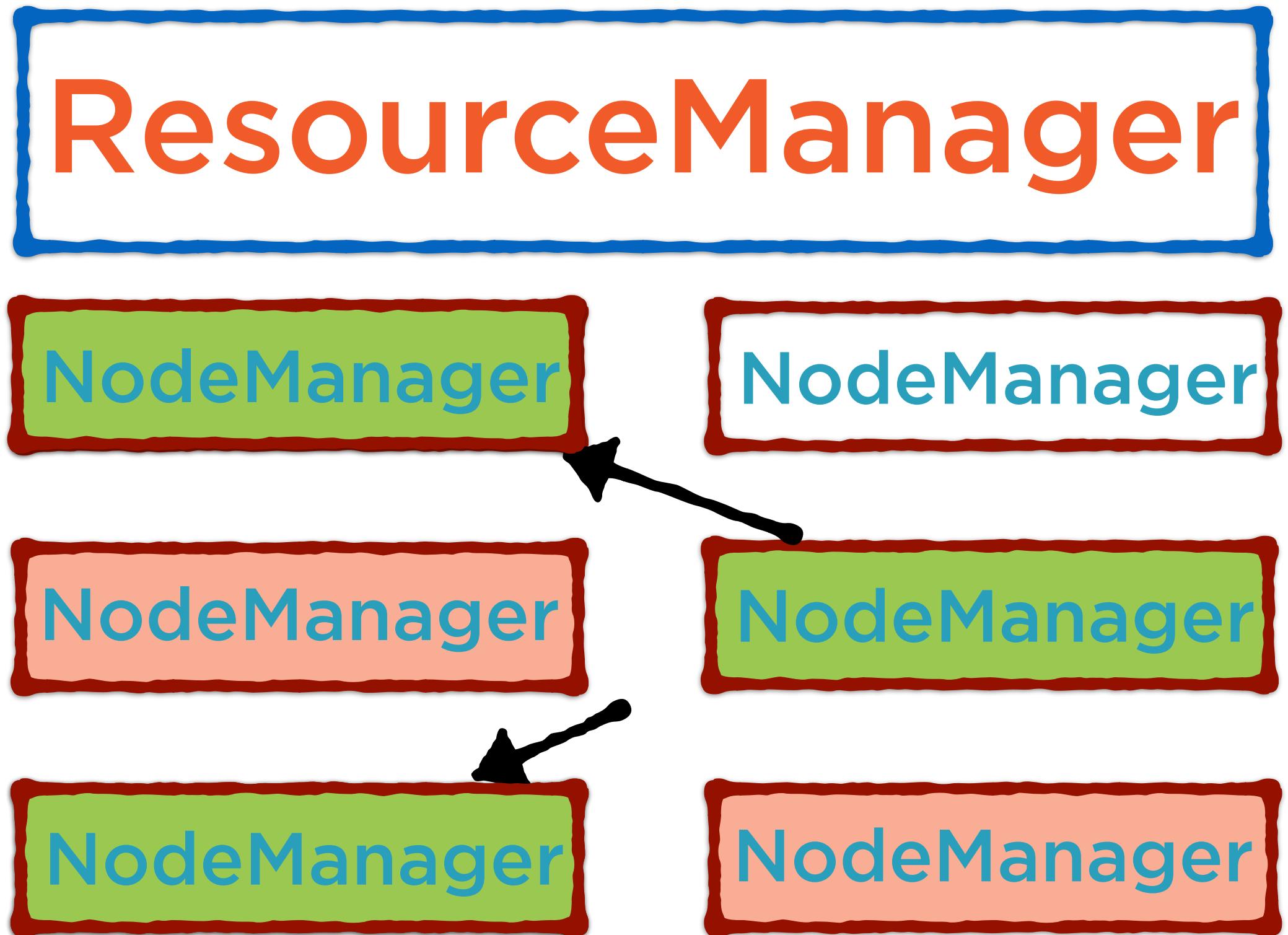


Application Processes



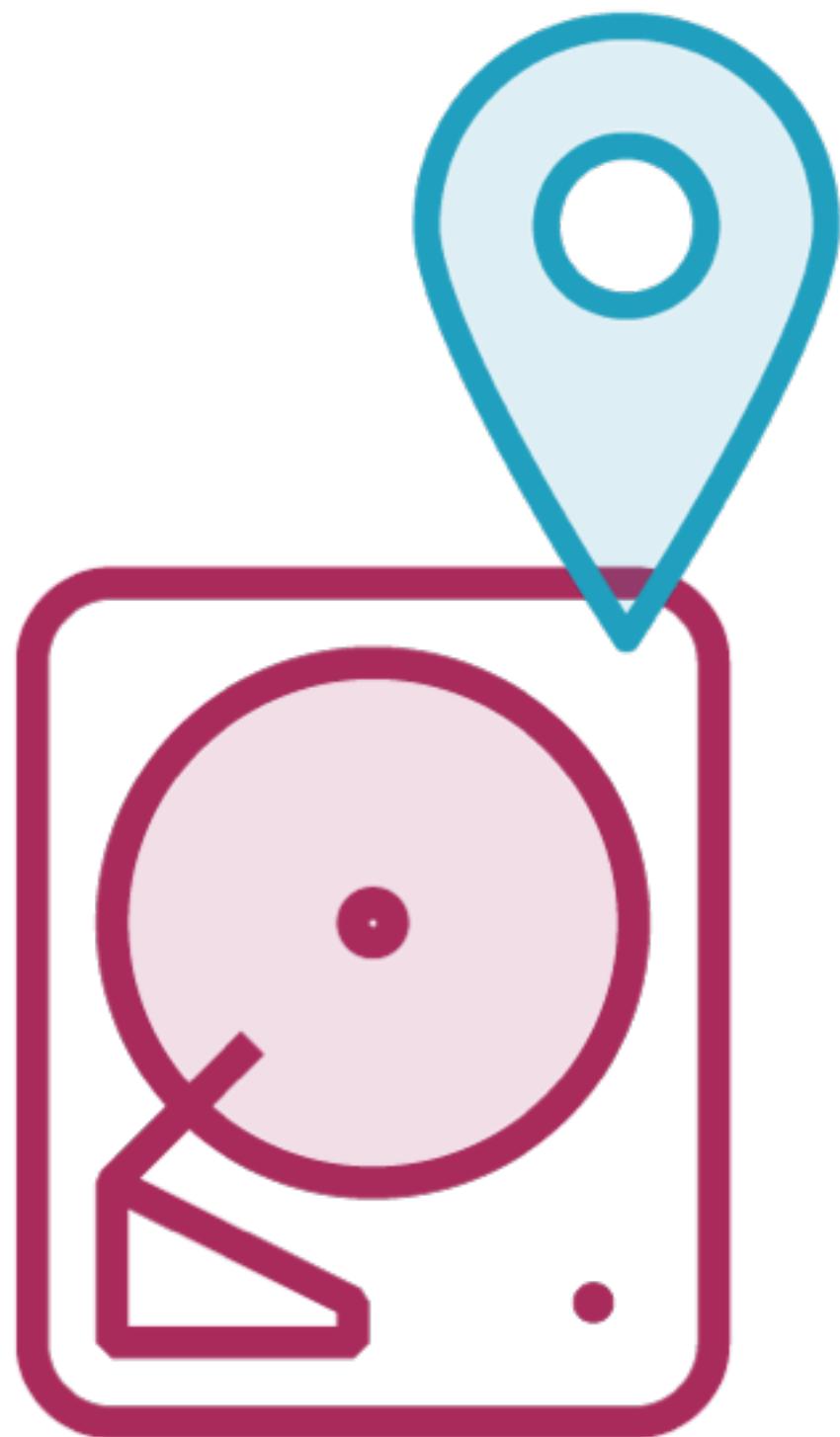
Assigns additional nodes
Notifies the original Application Master which made the request

Application Processes



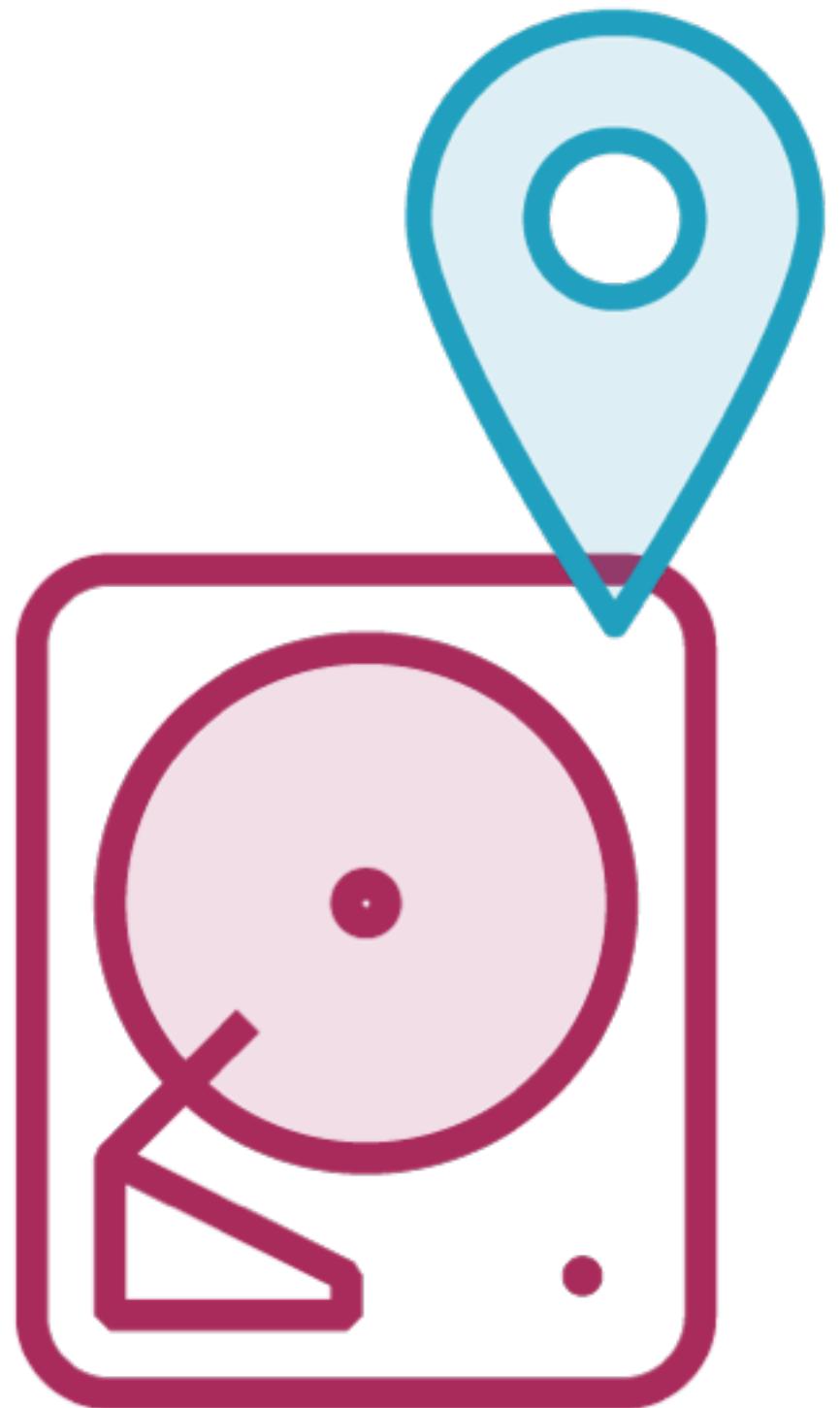
The Application Master
on the original node
starts off the Application
Masters on the newly
assigned nodes

The Location Constraint



**Try to minimize write bandwidth
i.e. assign a process to the same
node where the data to be
processed lives**

The Location Constraint



If CPU, memory are not available

wait!

Scheduling Policies

FIFO
Scheduler

Capacity
Scheduler

Fair
Scheduler

Introducing Hive

Hadoop

HDFS

MapReduce

YARN

**File system to
manage the storage
of data**

**Framework to
process data across
multiple servers**

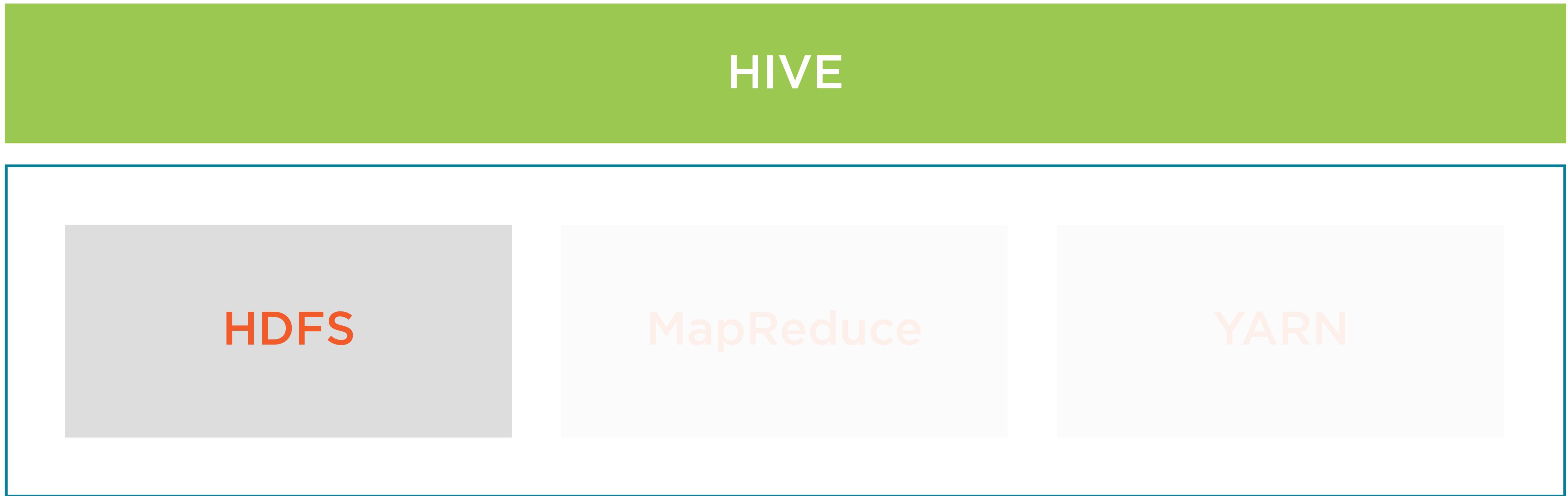
**Framework to run and
manage the data
processing tasks**

Hive on Hadoop



Hive runs **on top of the Hadoop distributed computing framework**

Hive on Hadoop



Hive stores its data in HDFS

HDFS

Hadoop Distributed File System

Data is stored as **files** - text files, binary files

Partitioned across machines in the cluster

Replicated for fault tolerance

Processing tasks **parallelized** across multiple machines

Hive on Hadoop



**Hive runs all processes in the form of
MapReduce jobs under the hood**

MapReduce

MapReduce

A **parallel programming model**

Defines the **logic** to process data on multiple machines

Batch processing operations on files in HDFS

Usually written in **Java** using the Hadoop MapReduce library

Hive on Hadoop

HIVE

HDFS

MapReduce

YARN

**Do we have to write MapReduce
code to work with Hive?**

No

HiveQL



Hive Query Language
**A SQL-like interface to the
underlying data**

HiveQL



Modeled on the Structured Query Language (SQL**)**

Familiar to analysts and engineers

Simple query constructs

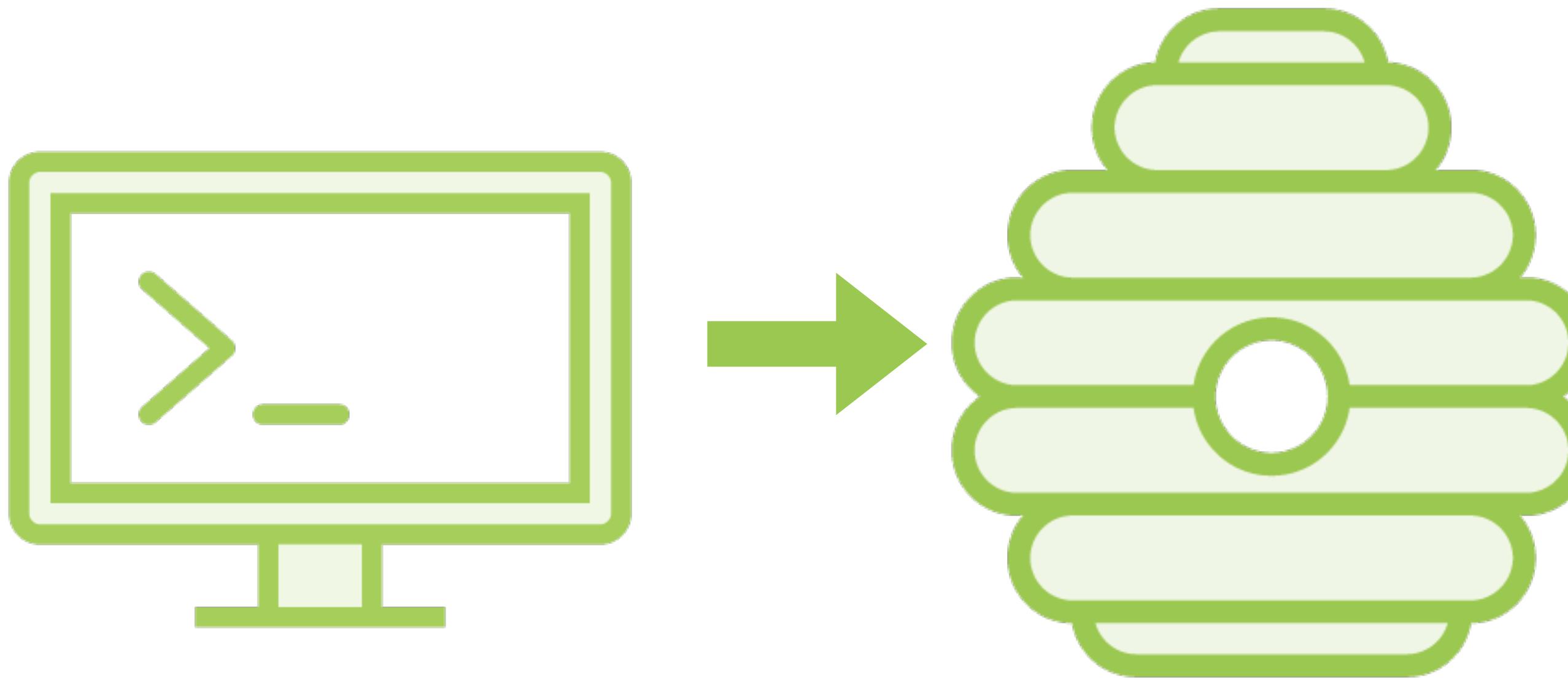
- **select**
- **group by**
- **join**

HiveQL



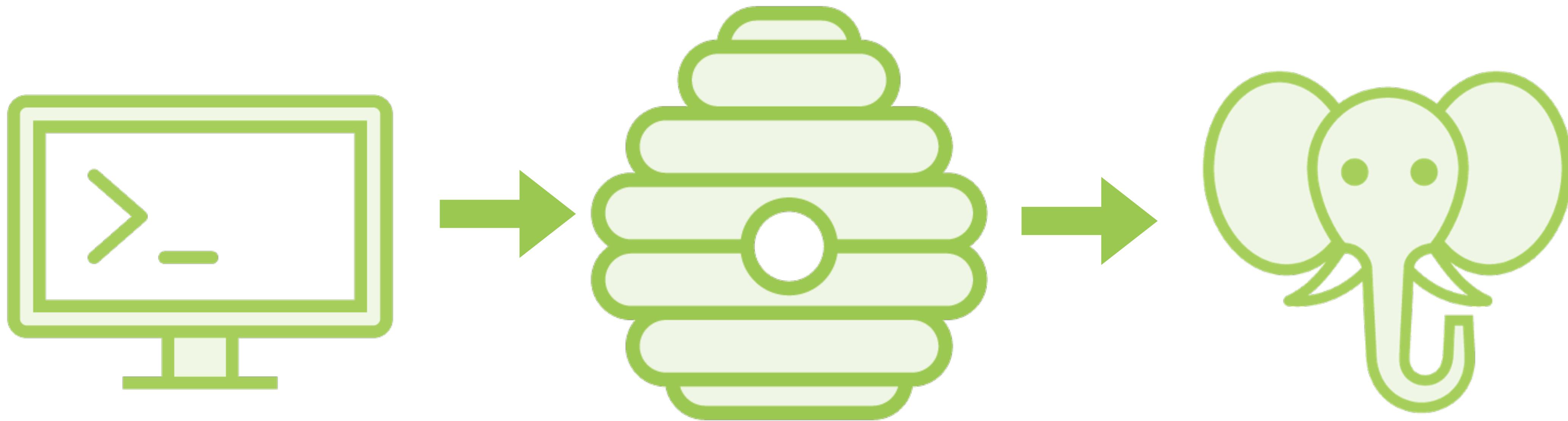
**Hive exposes files in HDFS in the form of
tables to the user**

HiveQL



Write SQL-like query in HiveQL and submit it to Hive

HiveQL



Hive will translate the query to MapReduce tasks and run them on Hadoop

HiveQL



MapReduce will process files on HDFS and return results to Hive

Hive **abstracts away** the details of
the underlying MapReduce jobs

Work with Hive **almost**
exactly like you would with a
traditional database

The Hive Metastore



**A Hive user sees data as if they were
stored in tables**

The Hive Metastore



**Exposes the file-based storage of HDFS in
the form of tables**

The Hive Metastore



Metastore

The **bridge** between data stored in files
and the tables exposed to users



The Hive Metastore

- Stores metadata for all the tables in Hive**
- Maps the files and directories in Hive to tables**
- Holds table definitions and the schema for each table**
- Has information on converting files to table representations**



The Hive Metastore



**Any database with a
JDBC driver can be used
as a metastore**



The Hive Metastore



**Development environments use
the built-in Derby database**

Embedded metastore



The Hive Metastore



Same Java process as Hive itself

One Hive session to connect to the database



The Hive Metastore

Production environments

Local metastore: Allows multiple sessions to connect to Hive

Remote metastore: Separate processes for Hive and the metastore

Hive vs. RDBMS

Data size

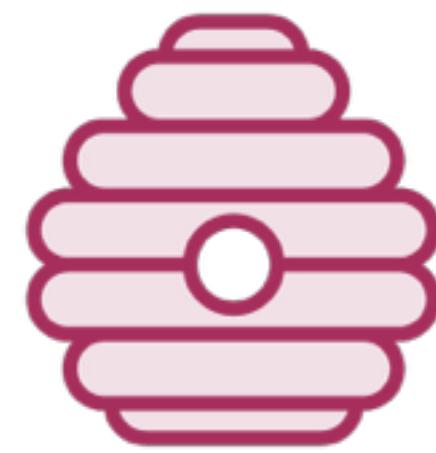
Computation

Latency

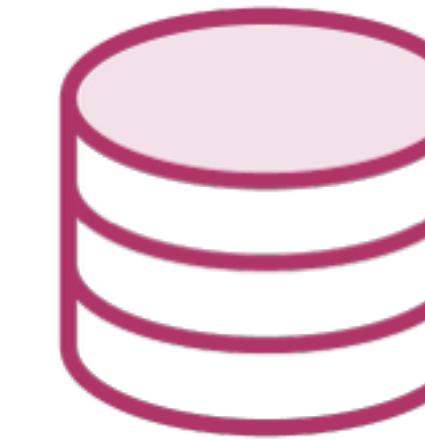
Operations

ACID compliance

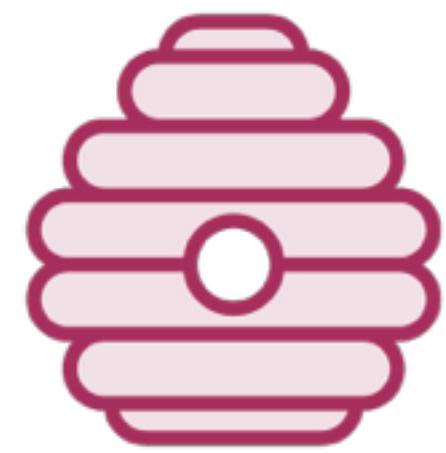
Query language



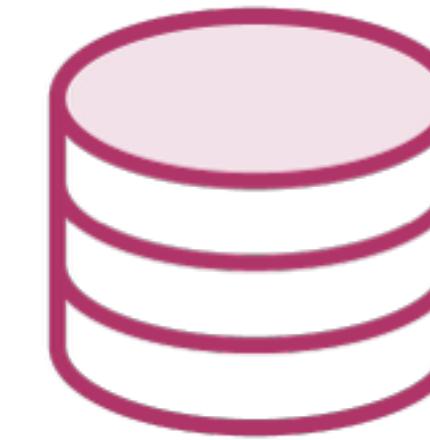
Hive vs. RDBMS



Hive	RDBMS
Large datasets	Small datasets
Parallel computations	Serial computations
High latency	Low latency
Read operations	Read/write operations
Not ACID compliant by default	ACID compliant
HiveQL	SQL



Hive vs. RDBMS

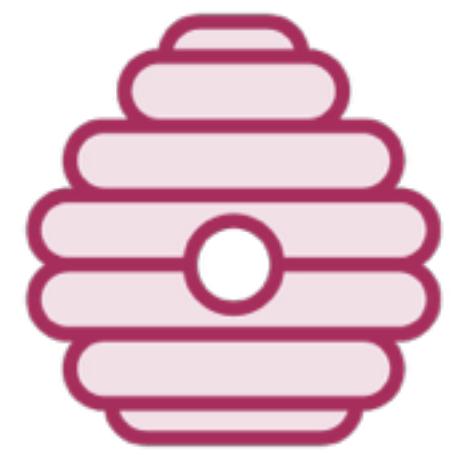


Hive

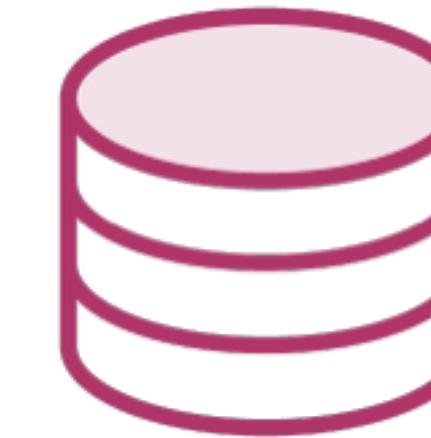
Large datasets
Parallel computations
High latency
Read operations
Not ACID compliant by default
HiveQL

RDBMS

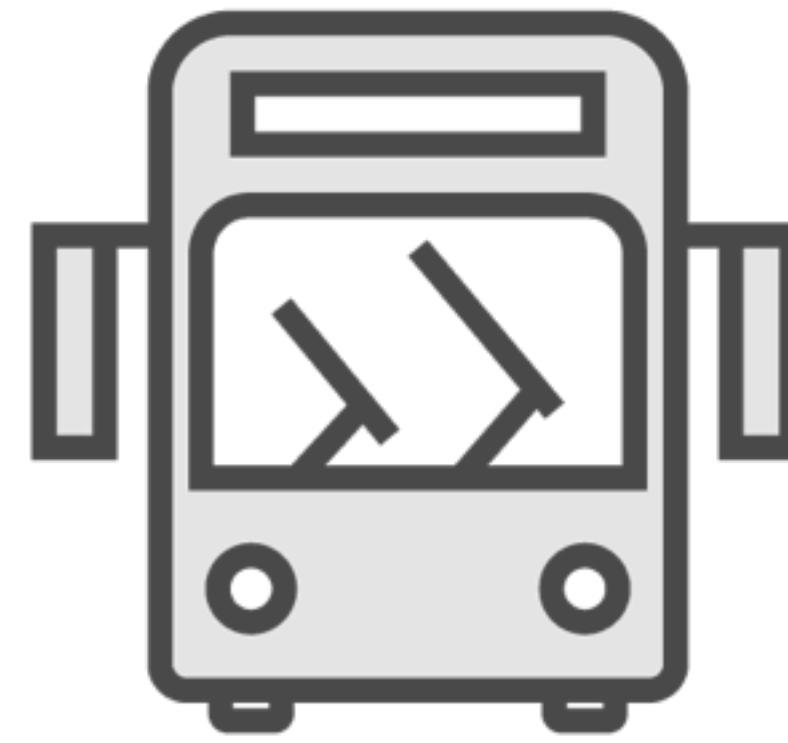
Small datasets
Serial computations
Low latency
Read/write operations
ACID compliant
SQL



Hive vs. RDBMS



Large Datasets

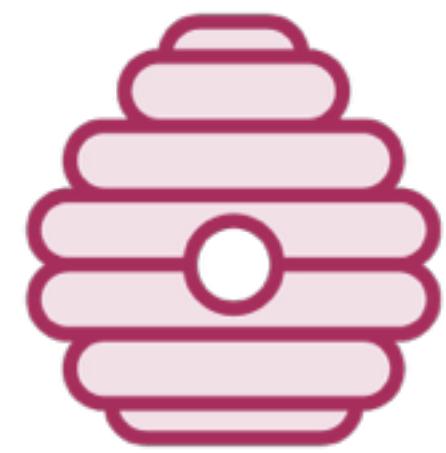


**Gigabytes or
petabytes**

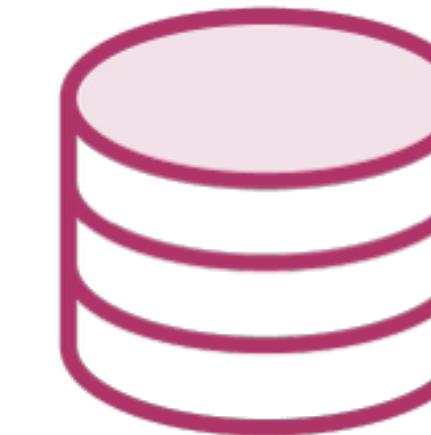
Small Datasets



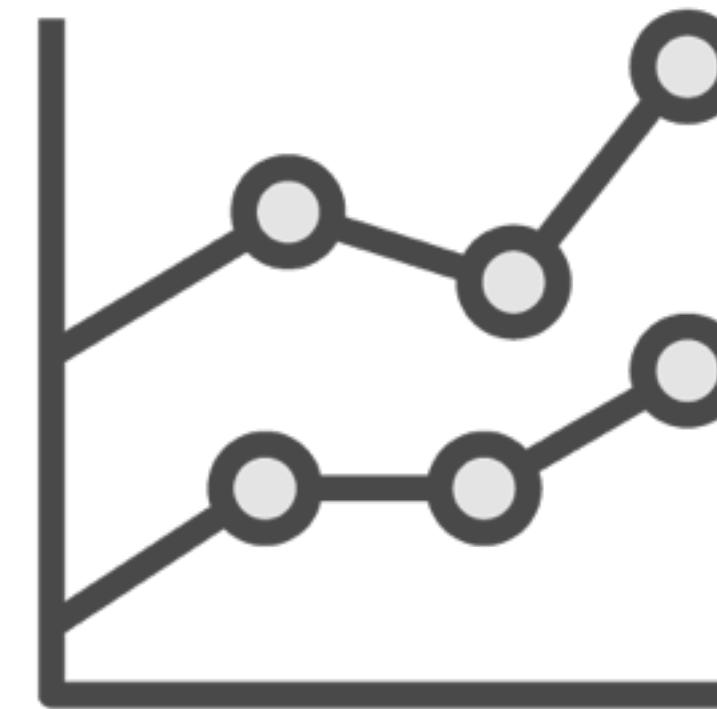
**Megabytes or
gigabytes**



Hive vs. RDBMS

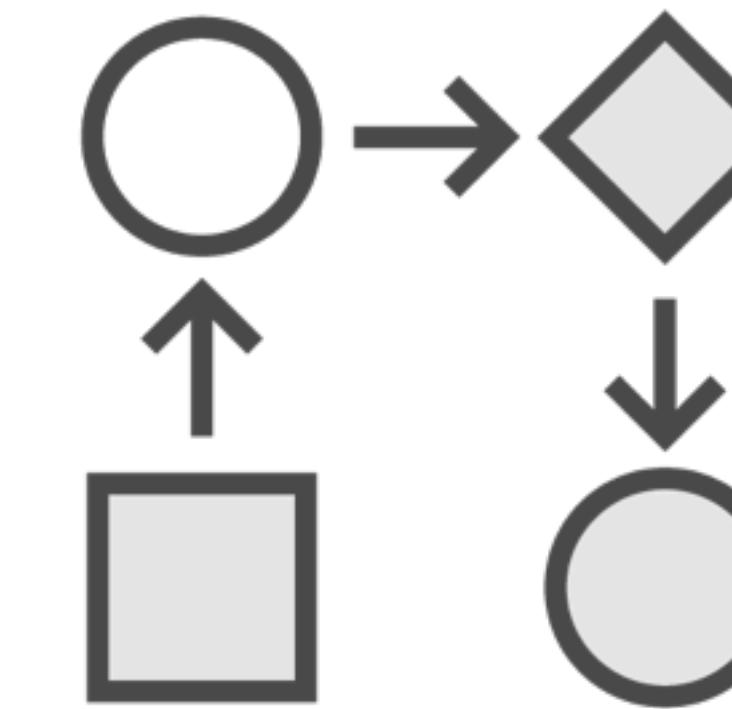


Large Datasets

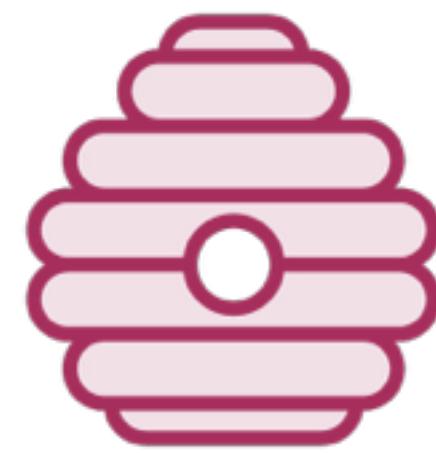


Calculating trends

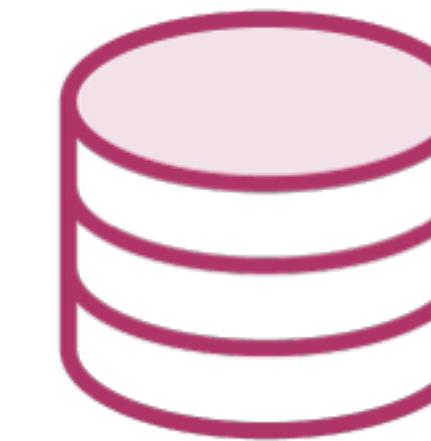
Small Datasets



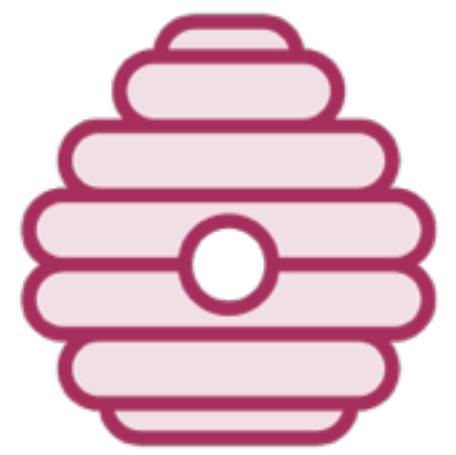
Accessing and updating
individual records



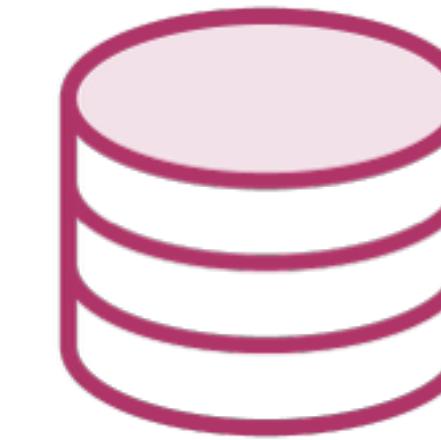
Hive vs. RDBMS



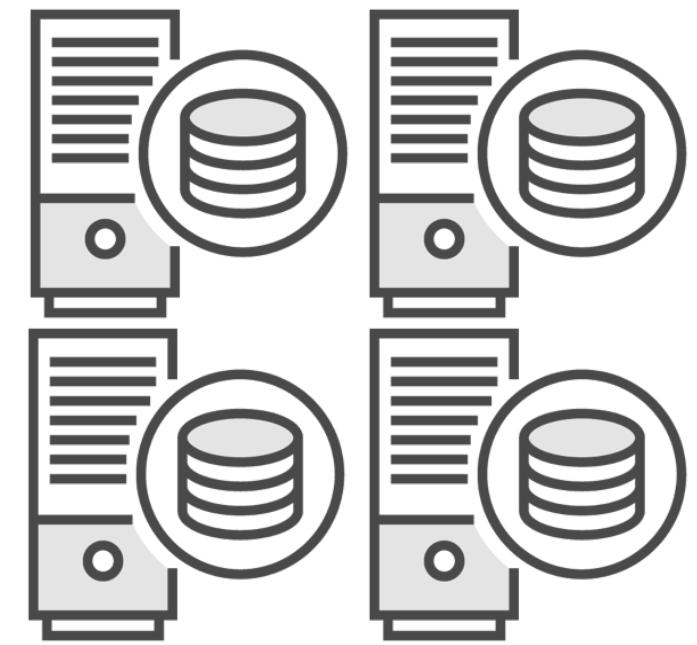
Hive	RDBMS
Large datasets	Small datasets
Parallel computations	Serial computations
High latency	Low latency
Read operations	Read/write operations
Not ACID compliant by default	ACID compliant
HiveQL	SQL



Hive vs. RDBMS



Parallel Computations

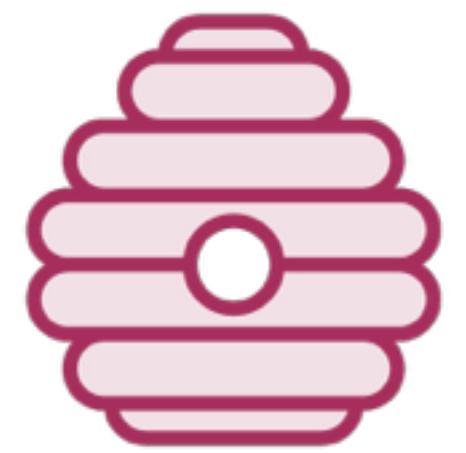


**Distributed system
with multiple
machines**

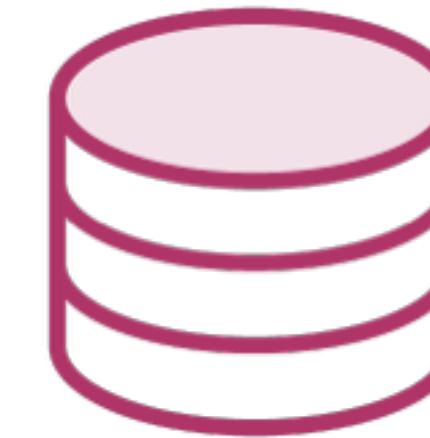
Serial Computations



**Single computer
with backup**



Hive vs. RDBMS

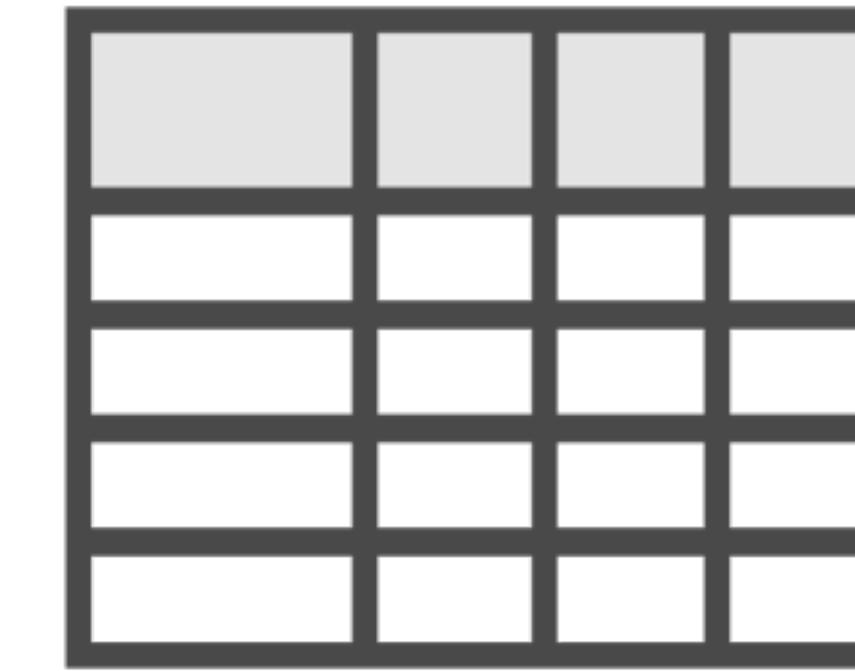


Parallel Computations

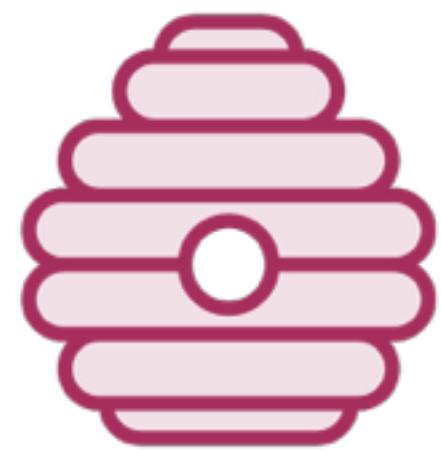


**Semi-structured data
files partitioned
across machines**

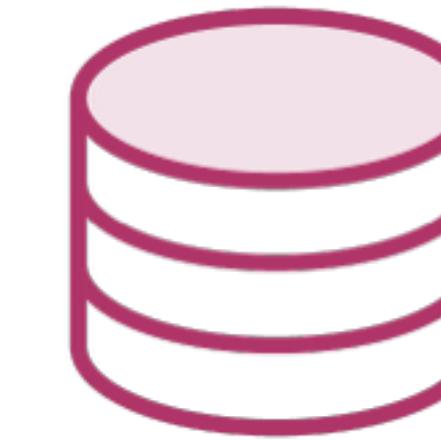
Serial Computations



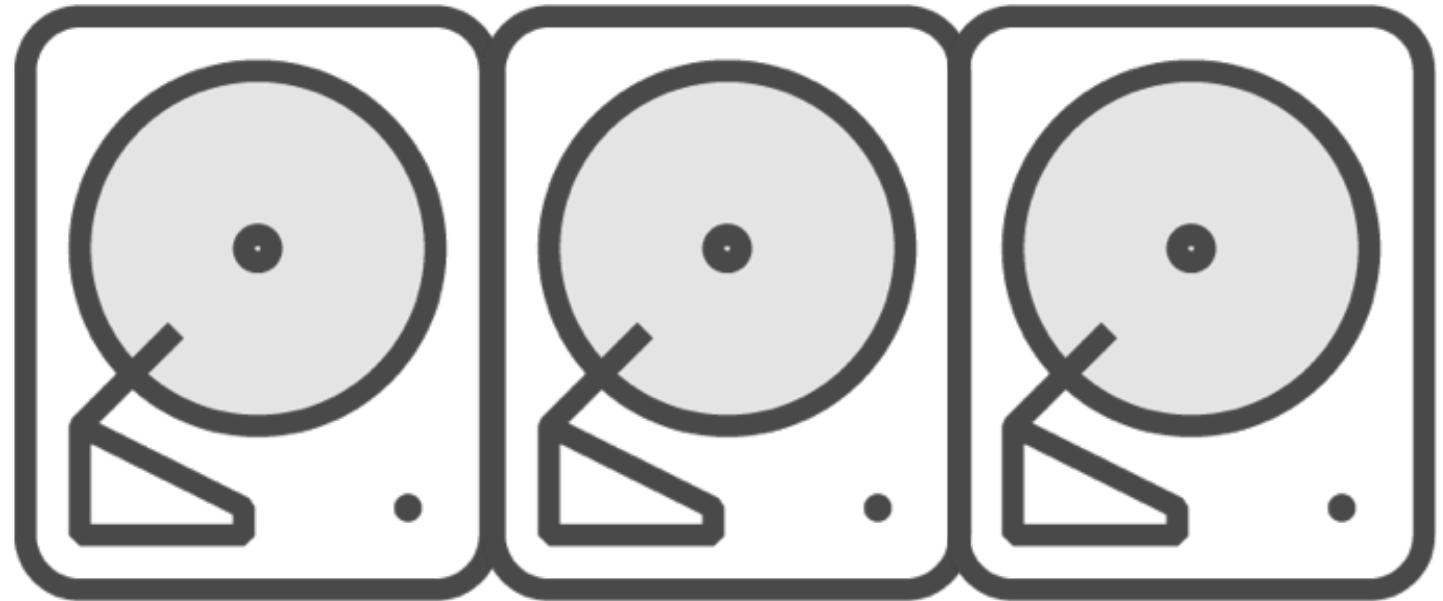
**Structured data in
tables on one
machine**



Hive vs. RDBMS

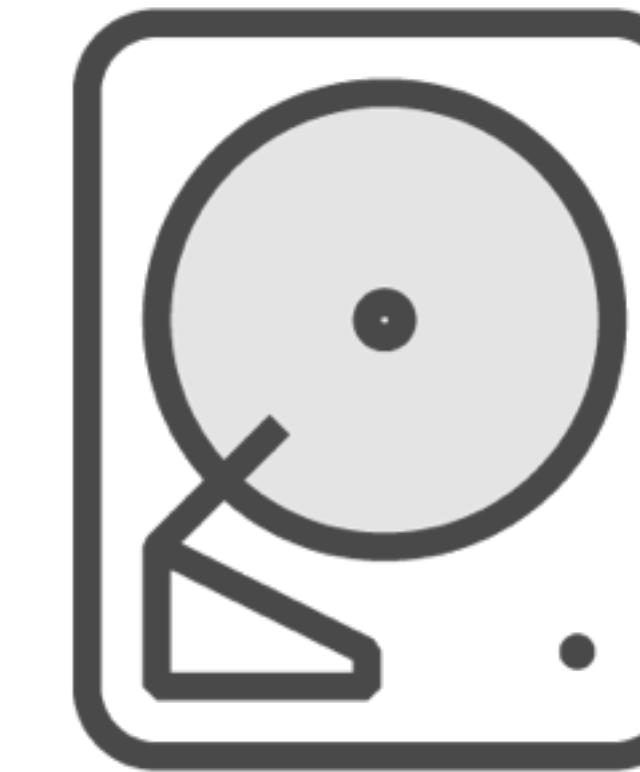


Parallel Computations

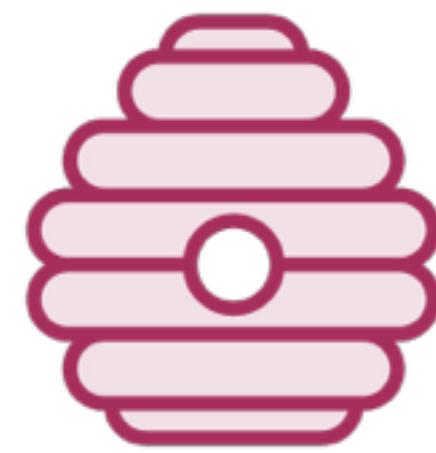


Disk space cheap,
can add space by
adding machines

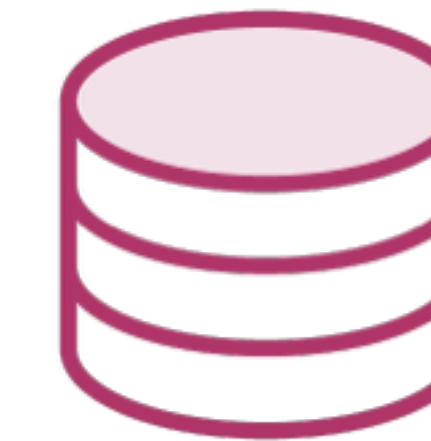
Serial Computations



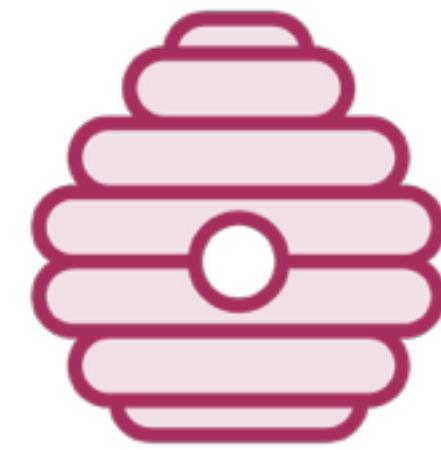
Disk space
expensive on a
single machine



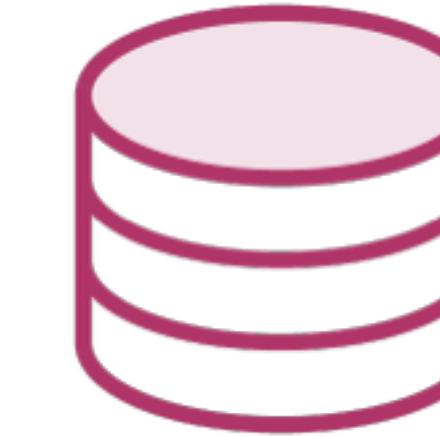
Hive vs. RDBMS



Hive	RDBMS
Large datasets	Small datasets
Parallel computations	Serial computations
High latency	Low latency
Read operations	Read/write operations
Not ACID compliant by default	ACID compliant
HiveQL	SQL



Hive vs. RDBMS

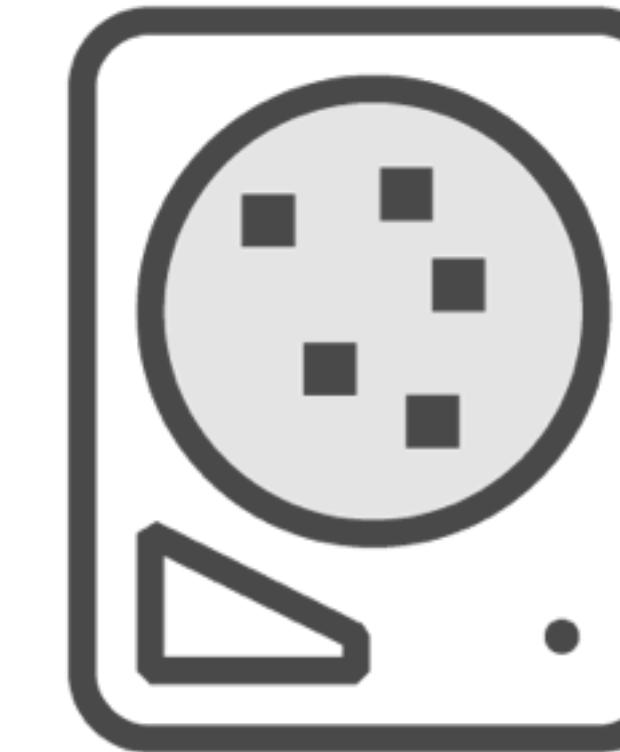


High Latency

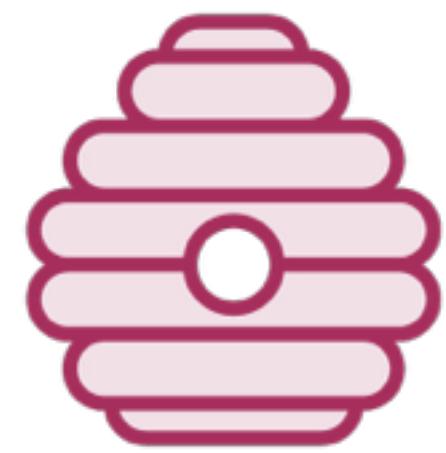


Records not indexed, cannot be accessed quickly

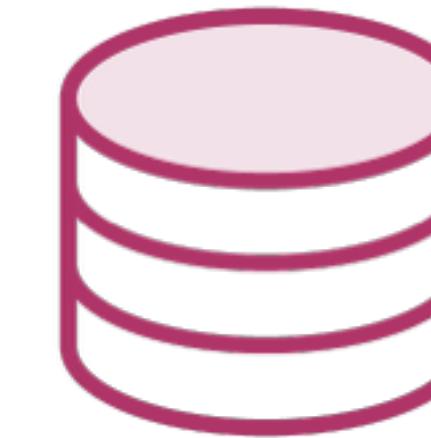
Low Latency



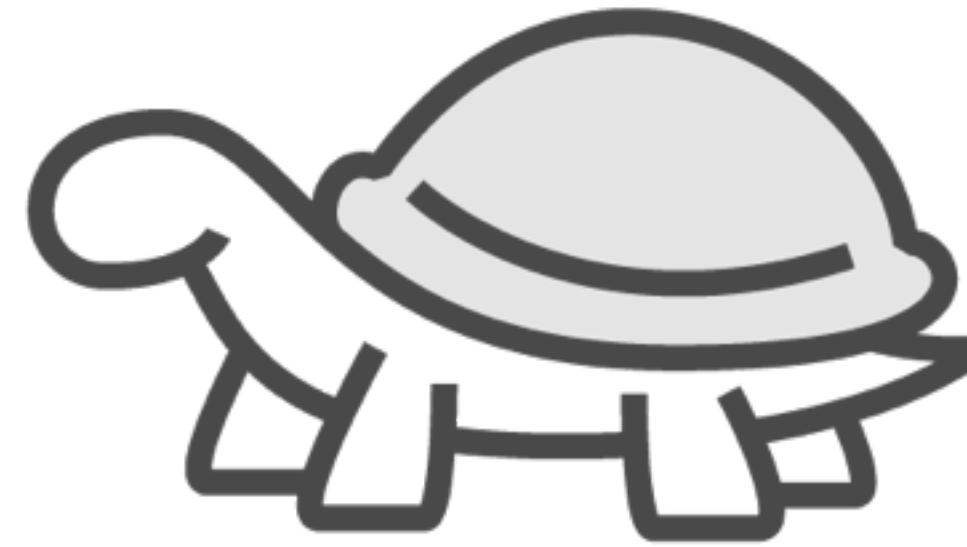
Records indexed, can be accessed and updated fast



Hive vs. RDBMS

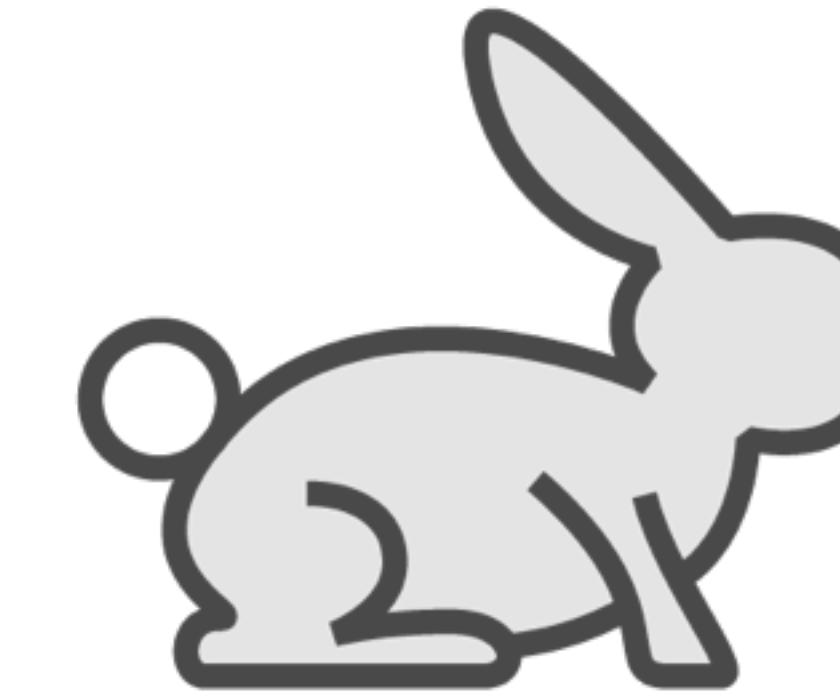


High Latency

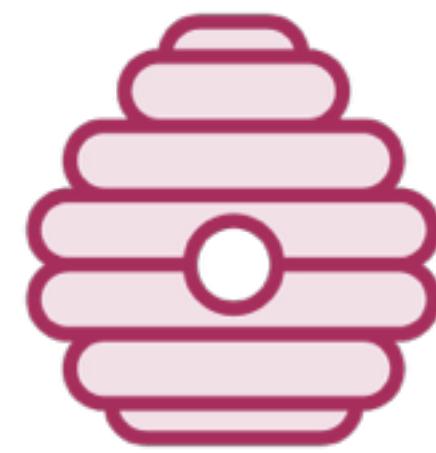


**Fetching a row will
run a MapReduce
that might take
minutes**

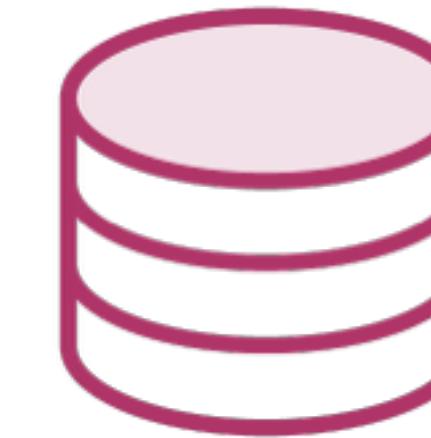
Low Latency



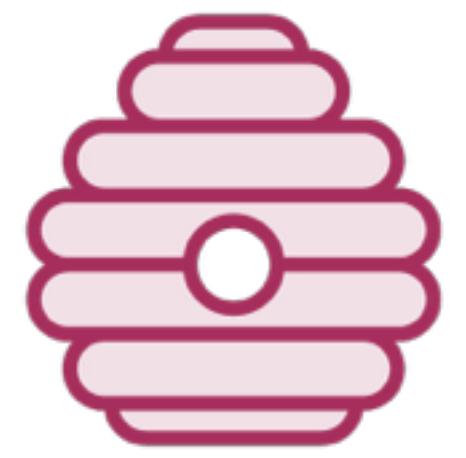
**Queries can be
answered in
milliseconds or
microseconds**



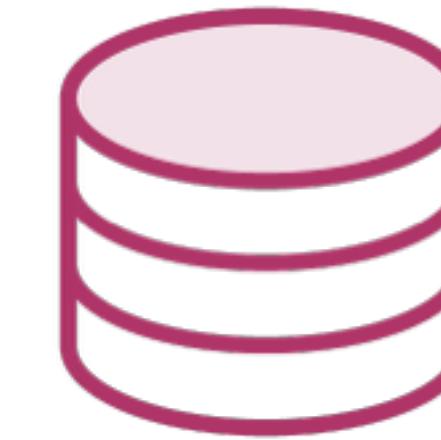
Hive vs. RDBMS



Hive	RDBMS
Large datasets	Small datasets
Parallel computations	Serial computations
High latency	Low latency
Read operations	Read/write operations
Not ACID compliant by default	ACID compliant
HiveQL	SQL



Hive vs. RDBMS

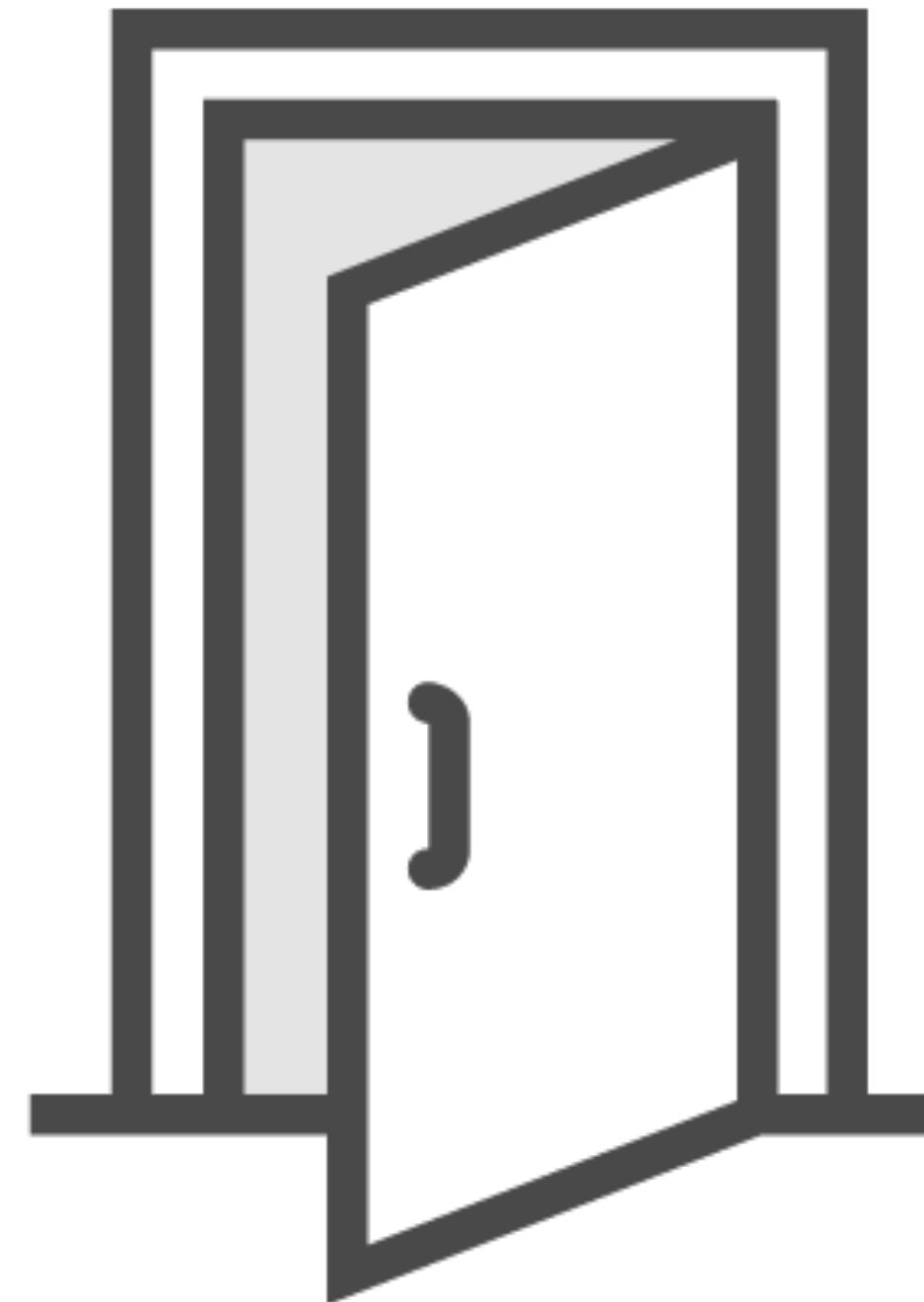


Read Operations



**Not the owner of
data**

Read/Write Operations



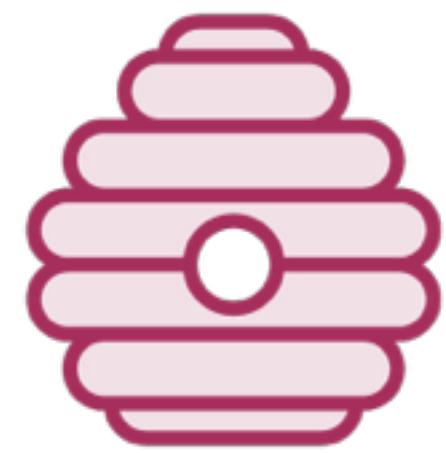
No Data Ownership

Hive stores files in HDFS

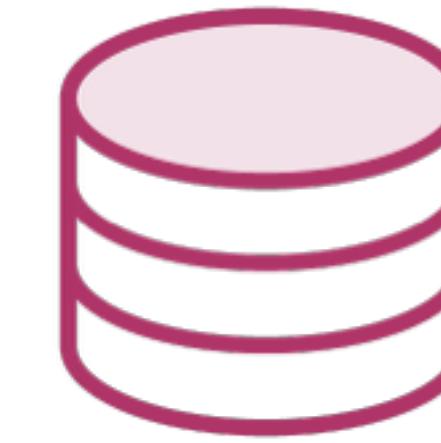
Hive files can be read and written by many technologies

- **Hadoop, Pig, Spark**

Hive database schema cannot be enforced on these files



Hive vs. RDBMS

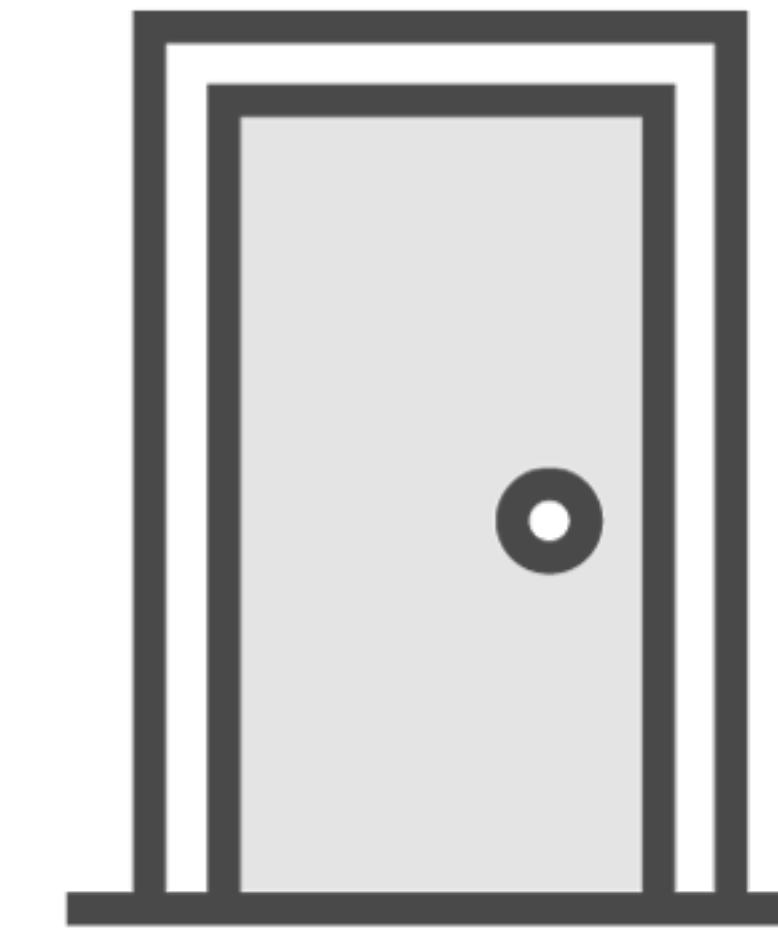


Read Operations

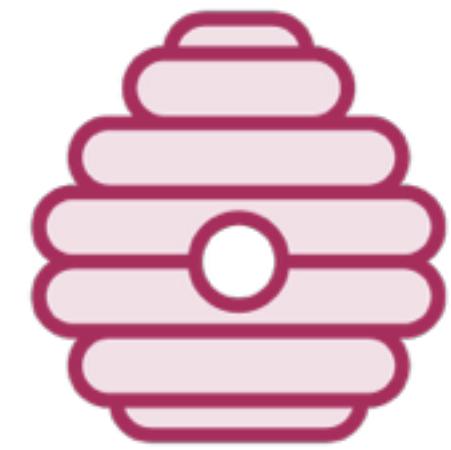


**Not the owner of
data**

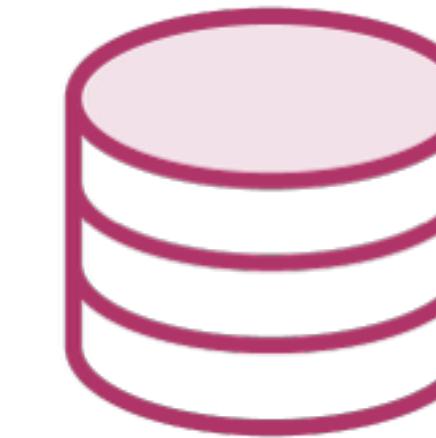
Read/Write Operations



**Sole gatekeeper for
data**



Hive vs. RDBMS



Read Operations



Schema-on-read

Read/Write Operations

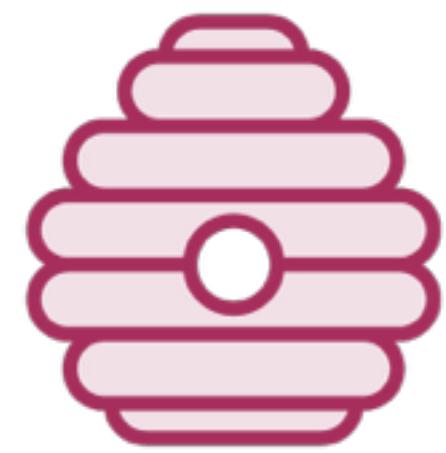


Schema-on-read

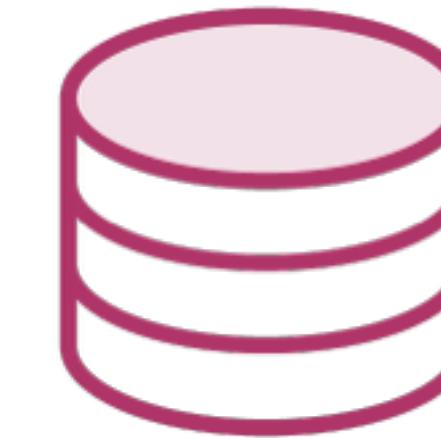
Number of columns, column types, constraints specified at table creation

Hive **tries to impose this schema when data is read**

It may not succeed, may pad data with nulls



Hive vs. RDBMS



Read Operations

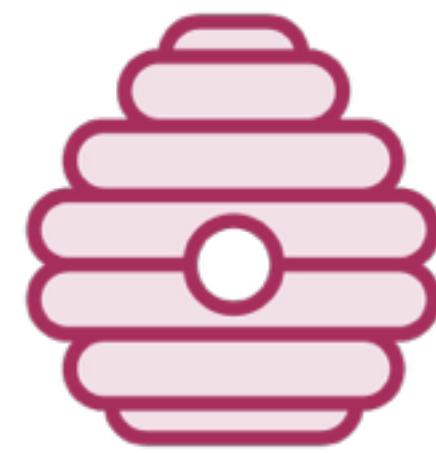


Schema-on-read

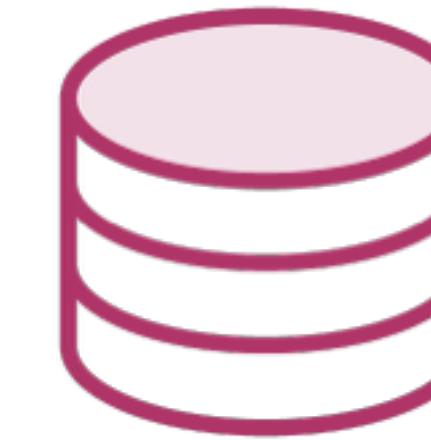
Read/Write Operations



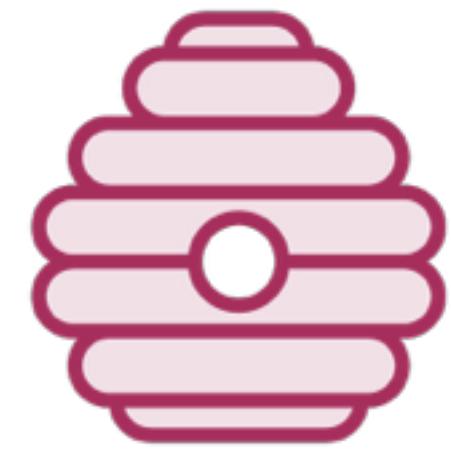
Schema-on-write



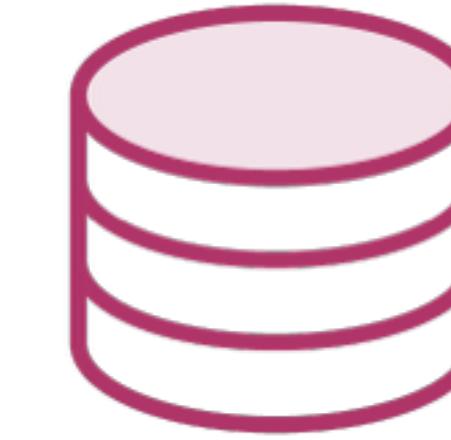
Hive vs. RDBMS



Hive	RDBMS
Large datasets	Small datasets
Parallel computations	Serial computations
High latency	Low latency
Read operations	Read/write operations
Not ACID compliant by default	ACID compliant
HiveQL	SQL



Hive vs. RDBMS



Not ACID Compliant

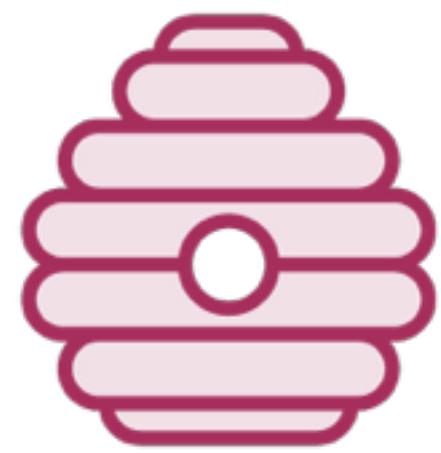


Data can be dumped into Hive tables from any source

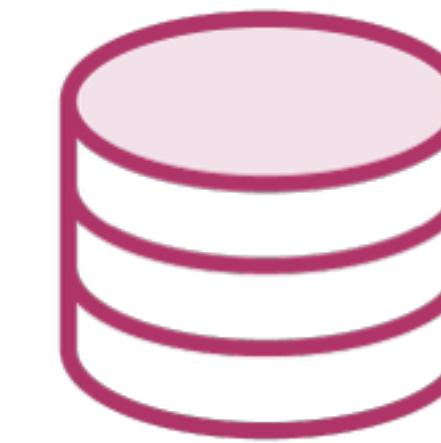
ACID Compliant



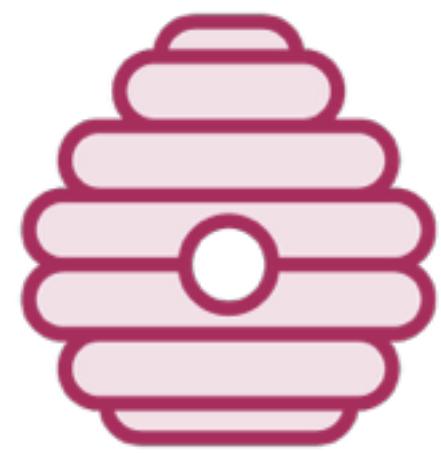
Only data which satisfies constraints are stored in the database



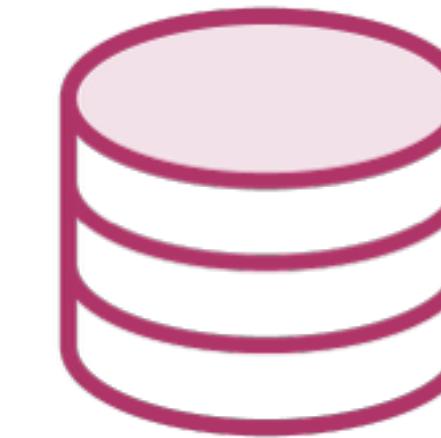
Hive vs. RDBMS



Hive	RDBMS
Large datasets	Small datasets
Parallel computations	Serial computations
High latency	Low latency
Read operations	Read/write operations
Not ACID compliant by default	ACID compliant
HiveQL	SQL



Hive vs. RDBMS



HiveQL

Schema on read, no constraints enforced

Minimal index support

Row level updates, deletes as a special case

Many more built-in functions

Only equi-joins allowed

Restricted subqueries

SQL

Schema on write keys, not null, unique all enforced

Indexes allowed

Row level operations allowed in general

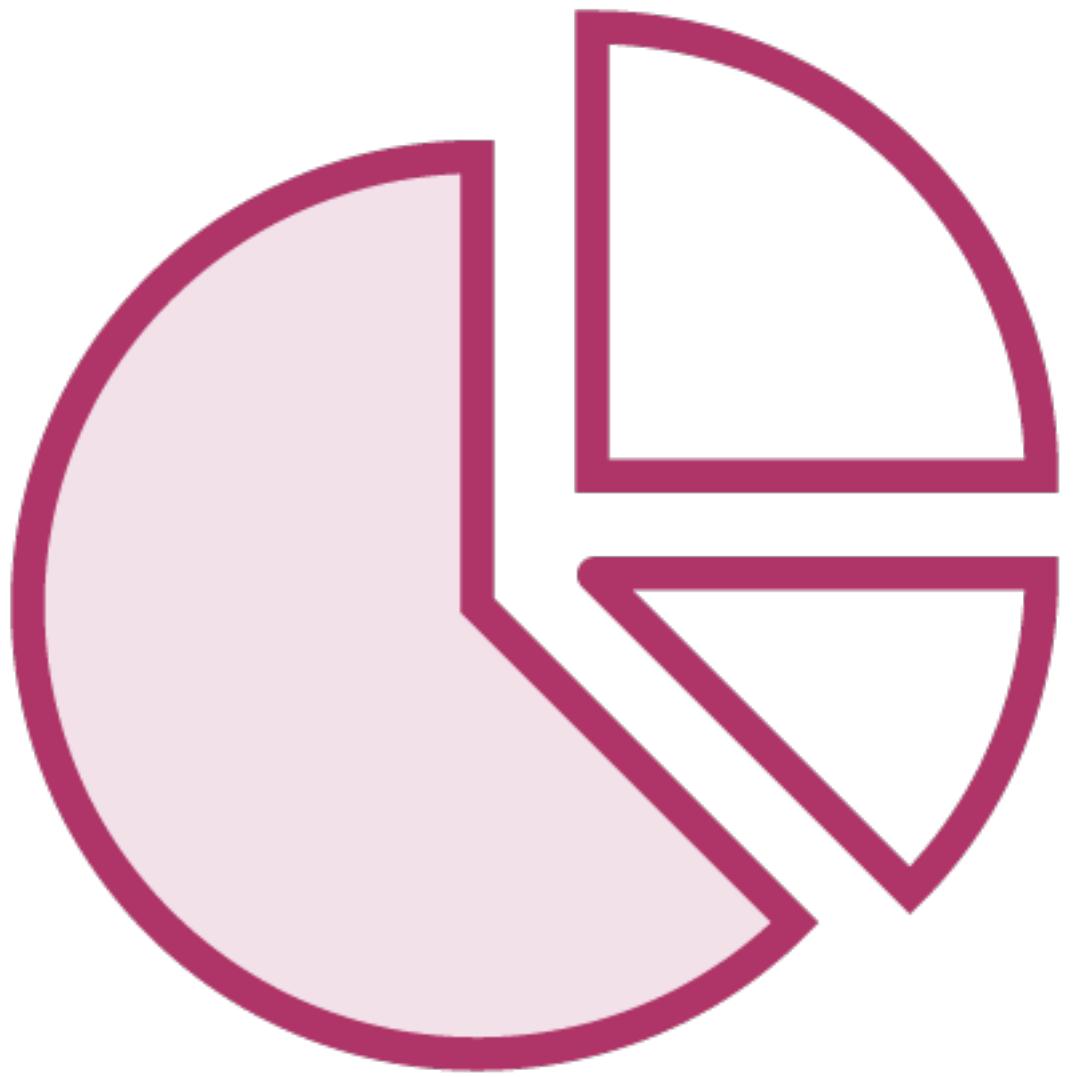
Basic built-in functions

No restriction on joins

Whole range of subqueries

OLAP Capabilities

Partitioning and Bucketing



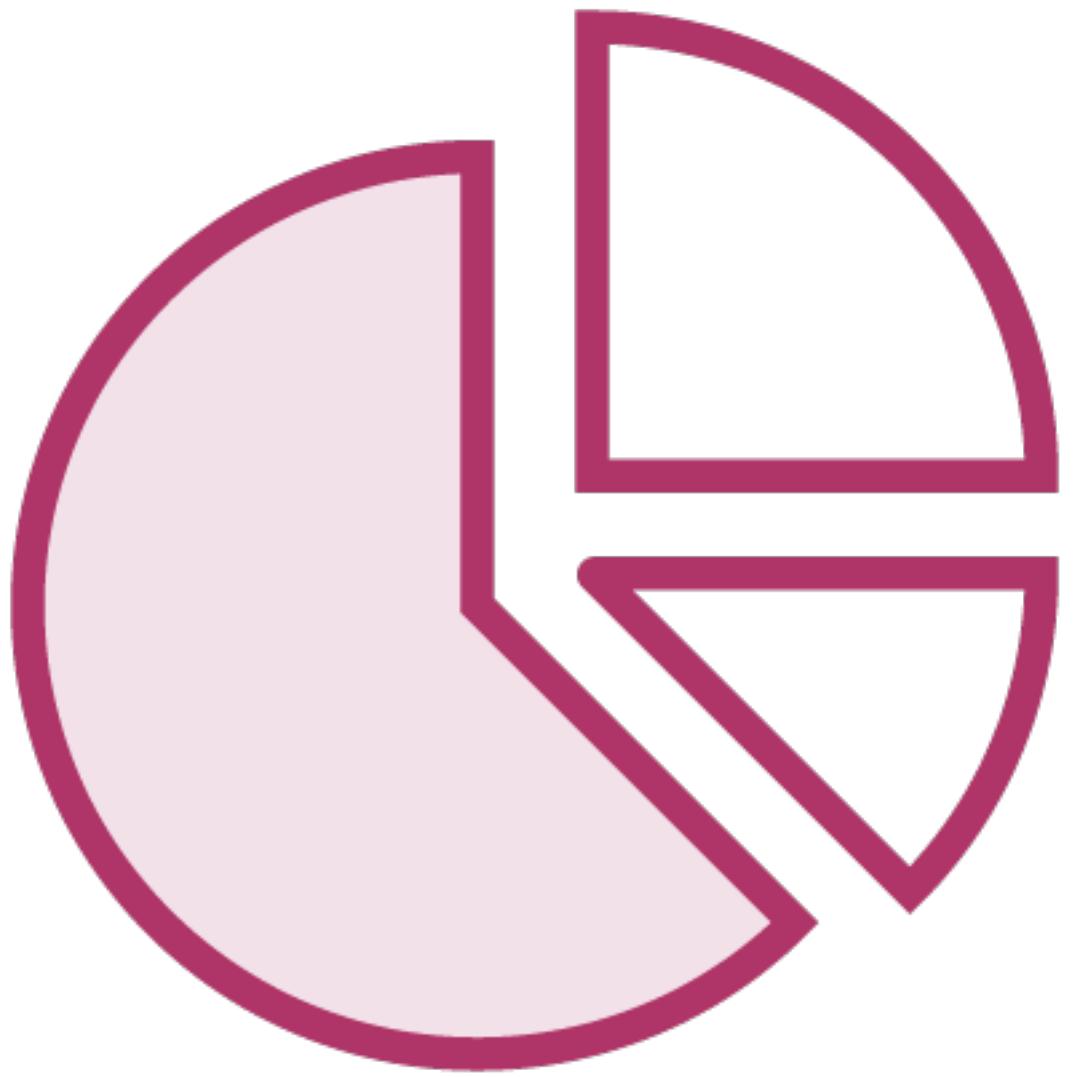
Partitioning



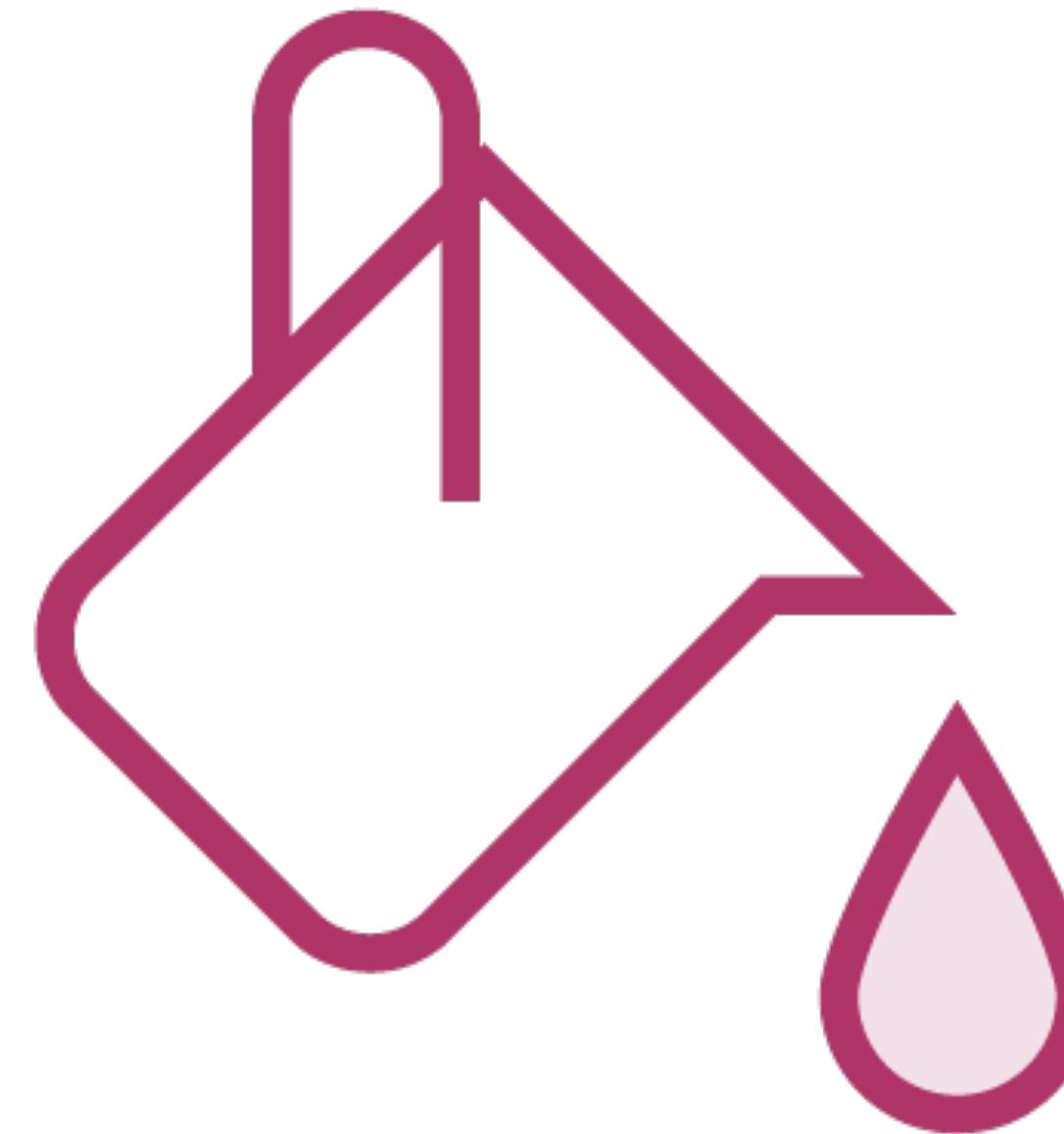
Bucketing

Splits data into smaller,
manageable parts

Partitioning and Bucketing



Partitioning



Bucketing

**Enables performance
optimizations**

Partitioning

Data may be naturally split into logical units



Customers in the US

Partitioning

Each of these units will be stored in a different directory

WA

CT

OR

NY

CA

GA

Partitioning

**State specific queries will run only on
data in **one** directory**

WA

CT

OR

NY

CA

GA

Partitioning

Splits may **not** be of the same size

WA

CT

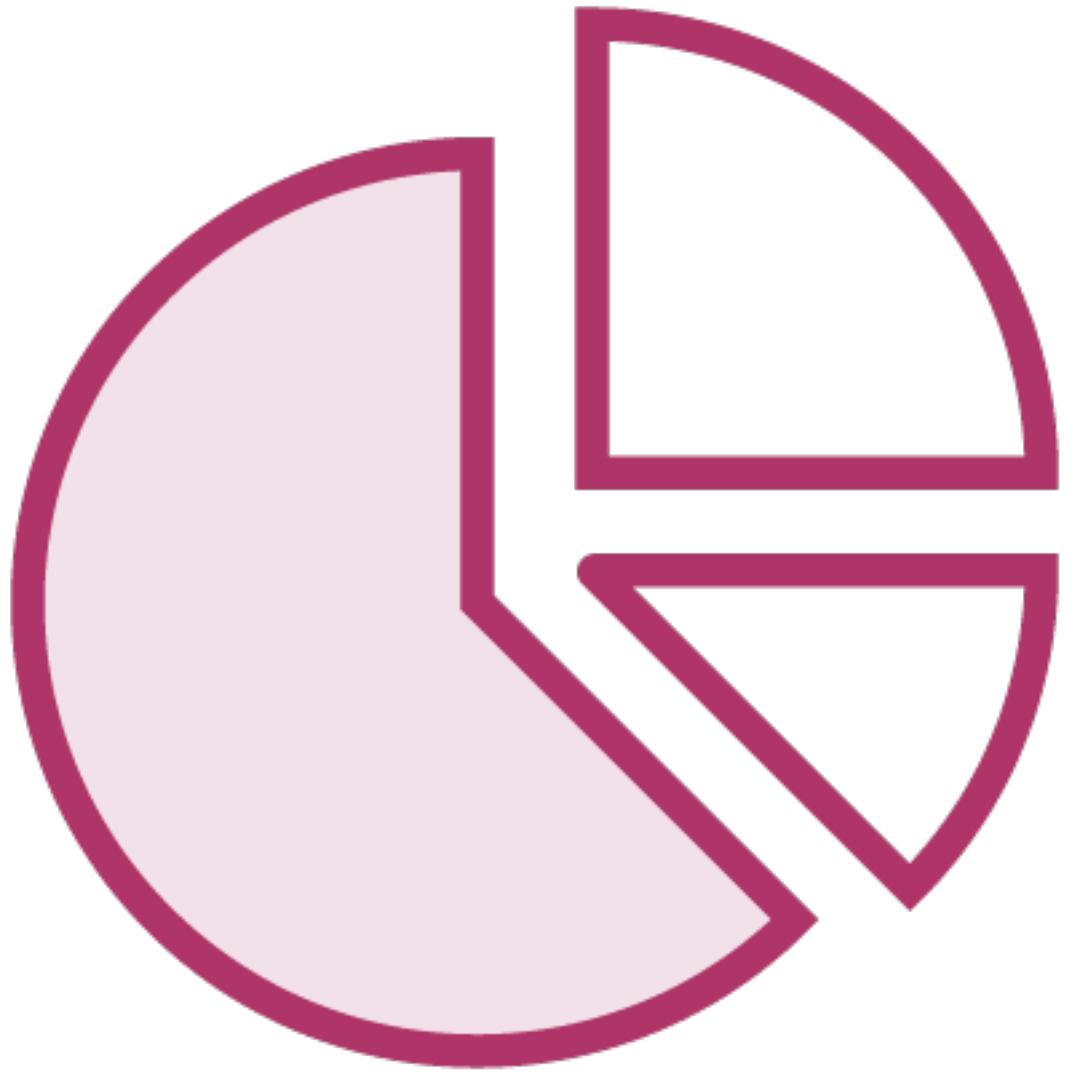
OR

NY

CA

GA

Partitioning and Bucketing



Partitioning



Bucketing

Bucketing

Size of each split should be the same



Customers in the US

Bucketing

Hash of a column value - address, name,



Customers in the US

Bucketing

Each bucket is a separate file

Bucket 1

Bucket 3

Bucket 2

Bucket 4

Bucketing

Makes sampling and joining
data more efficient

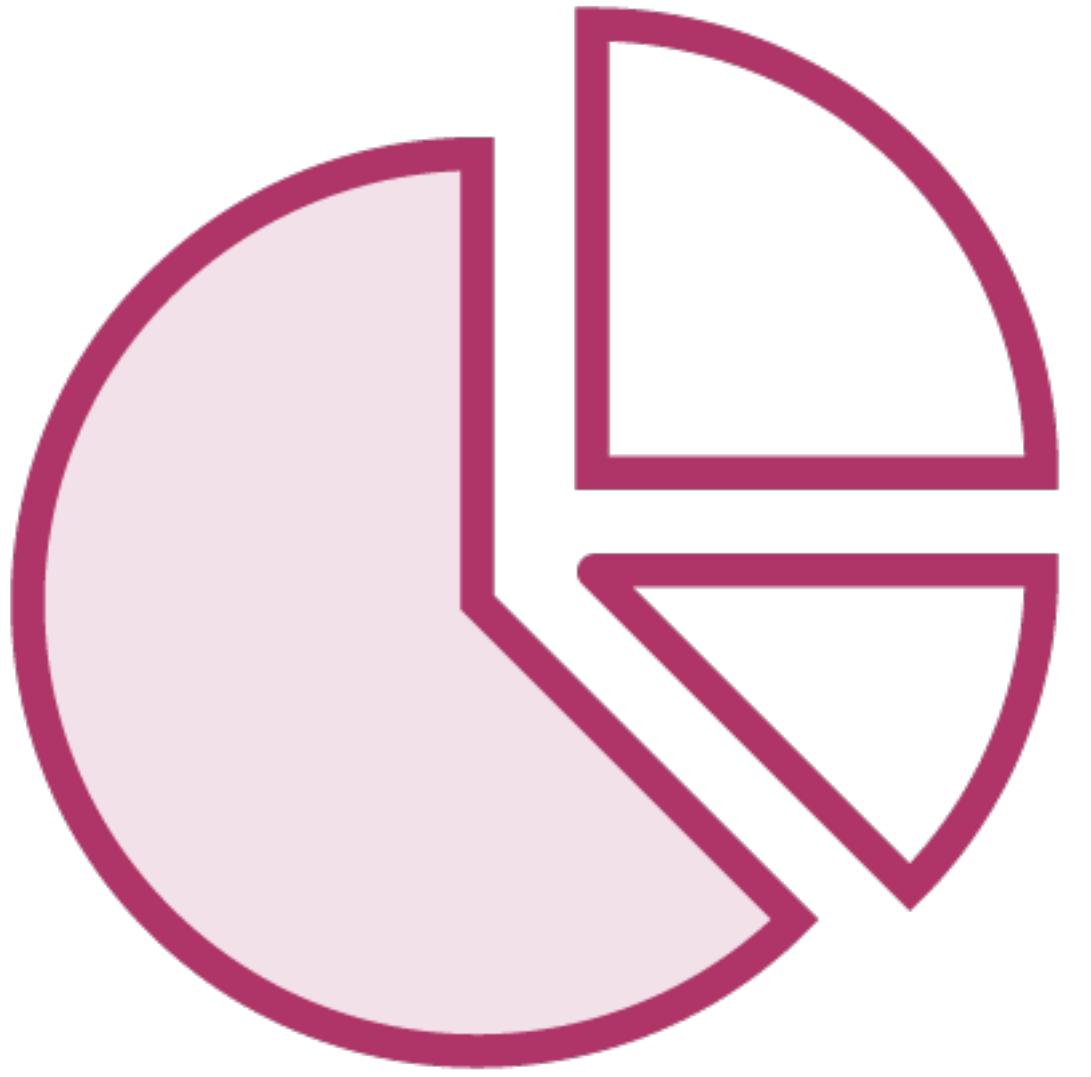
Bucket 1

Bucket 3

Bucket 2

Bucket 4

Partitioning and Bucketing

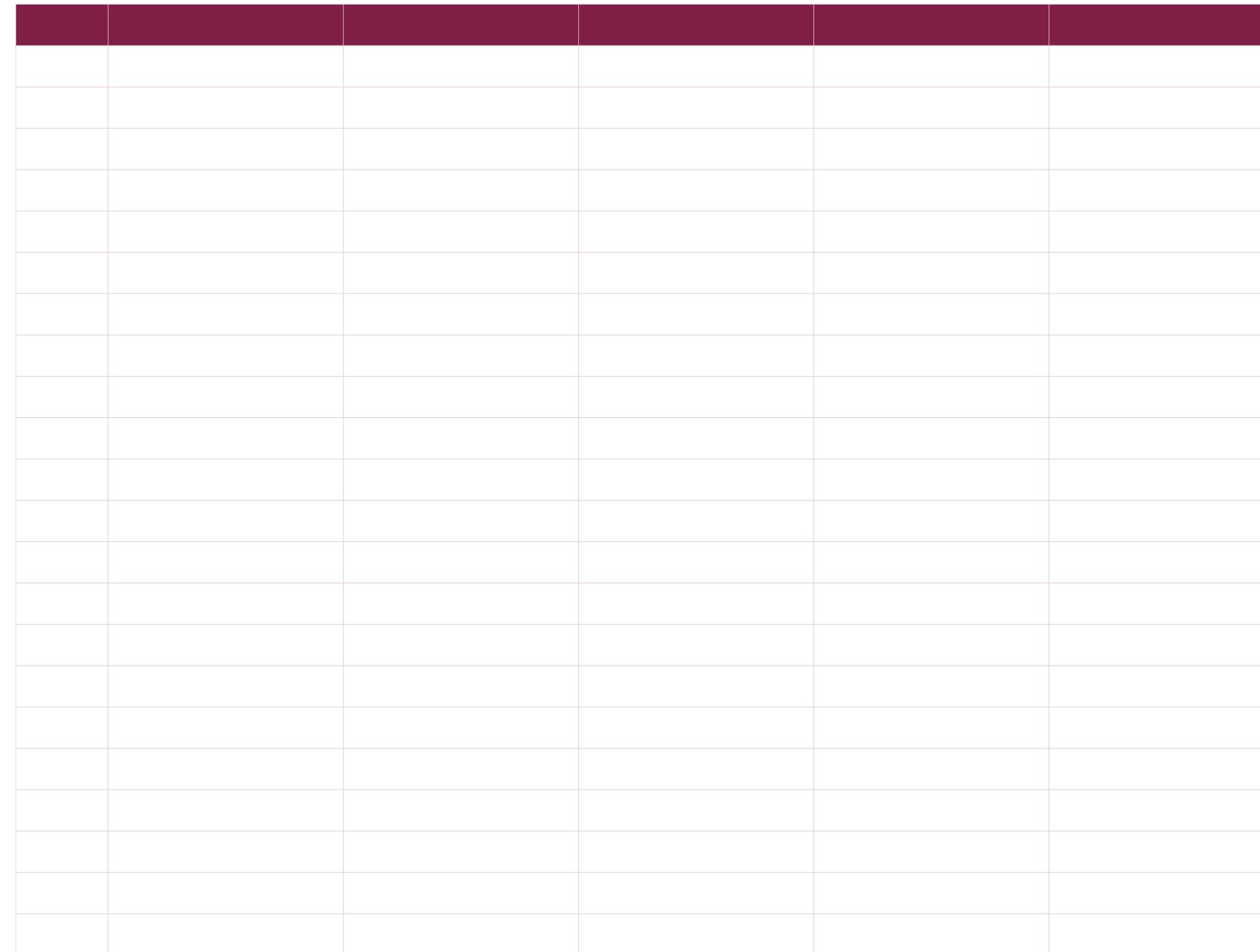


Partitioning



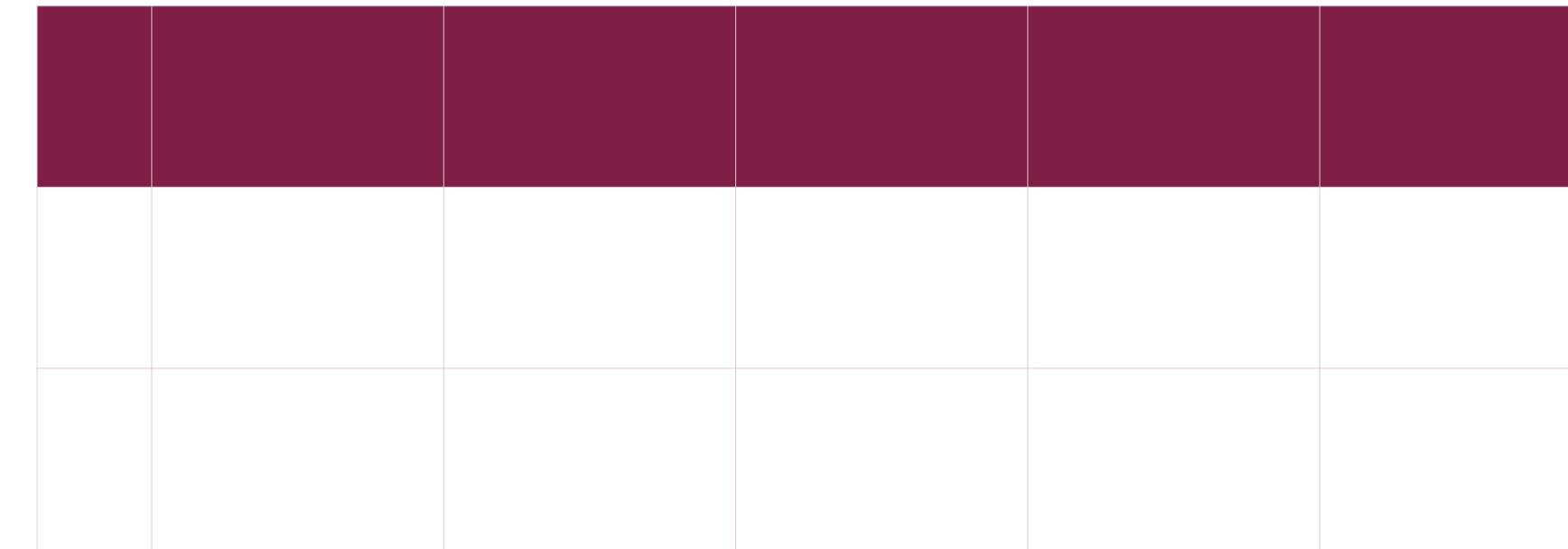
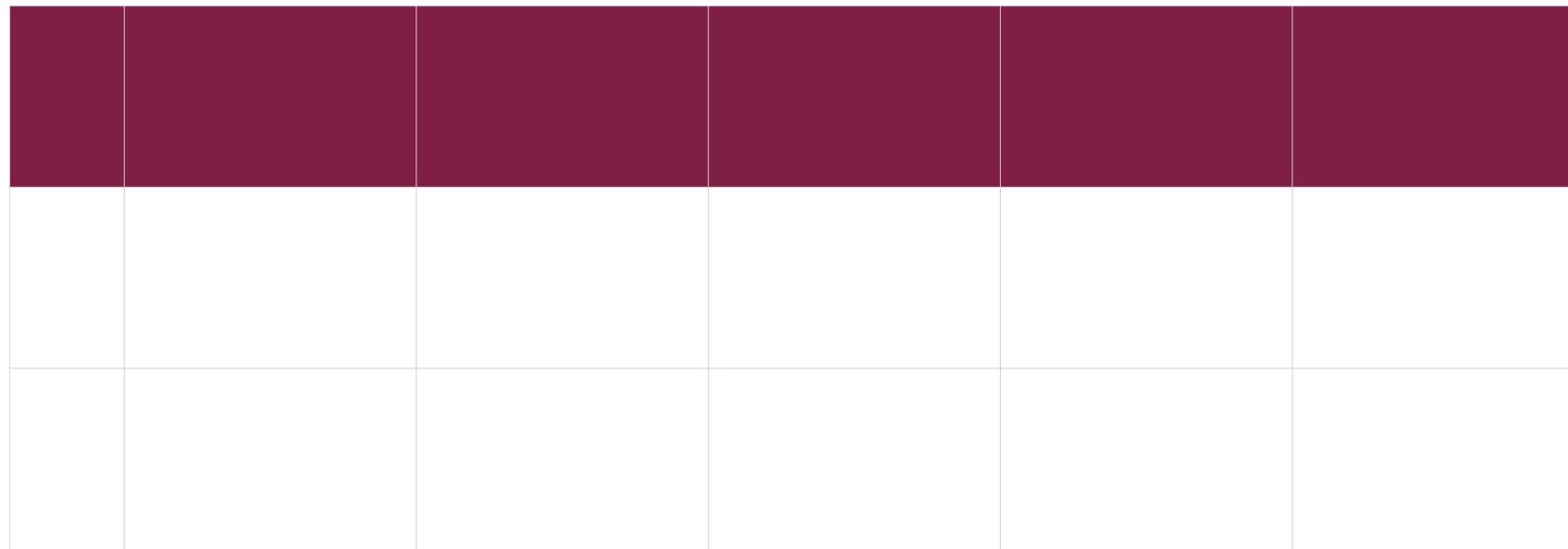
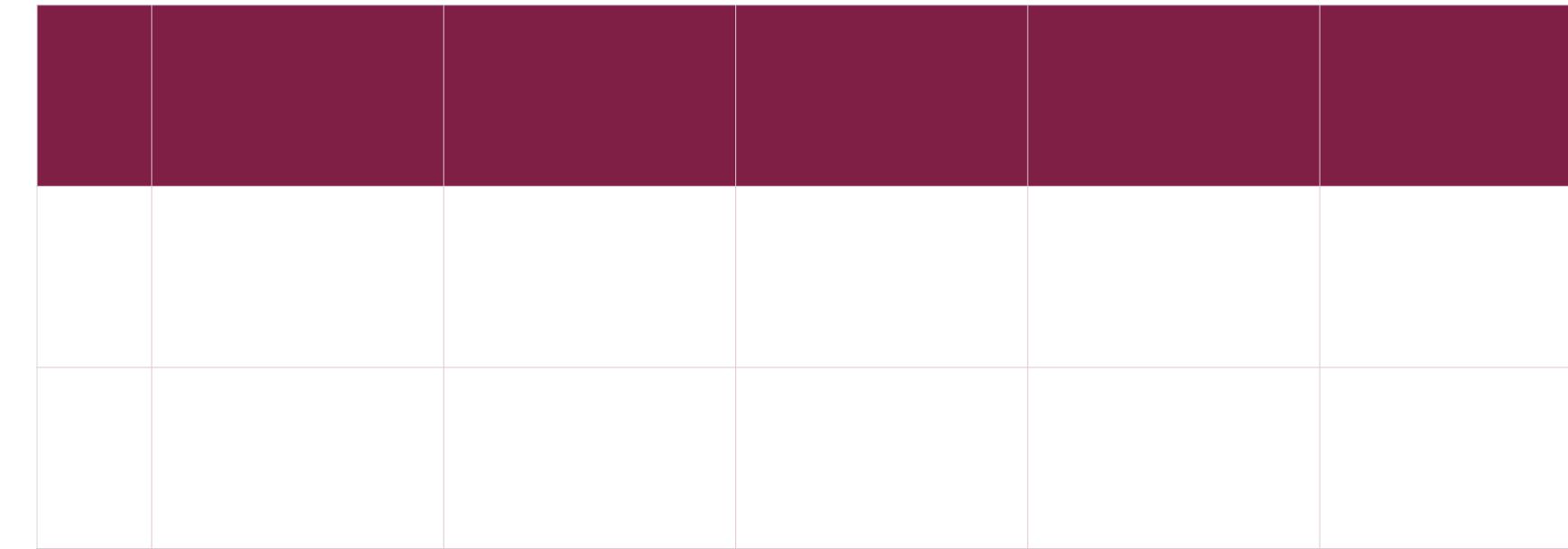
Bucketing

Partitioning and Bucketing



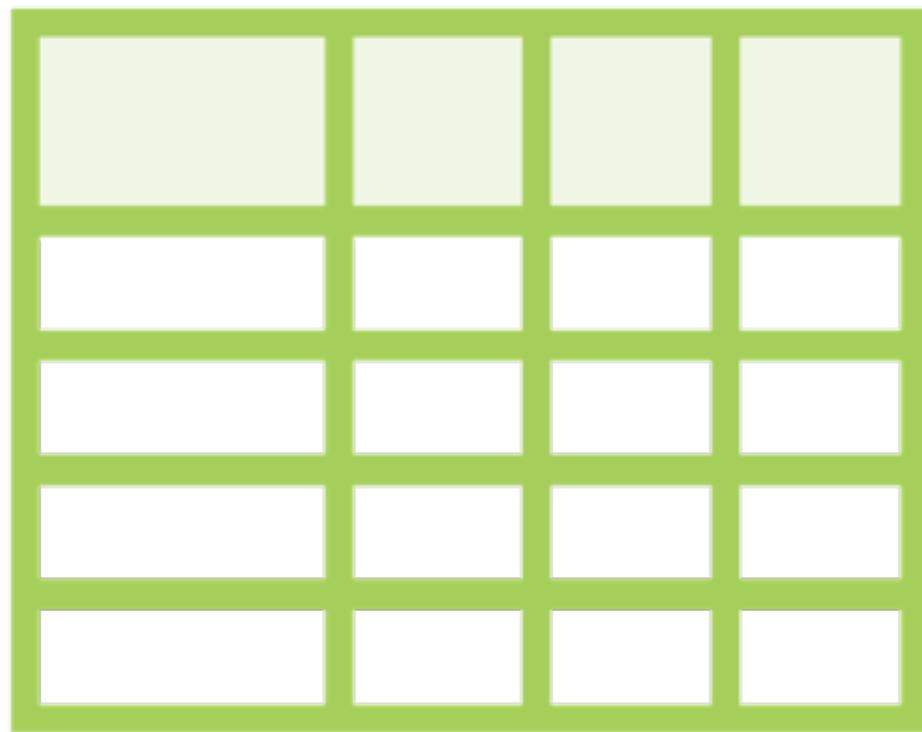
**Rather than running queries
on such tables**

Partitioning and Bucketing

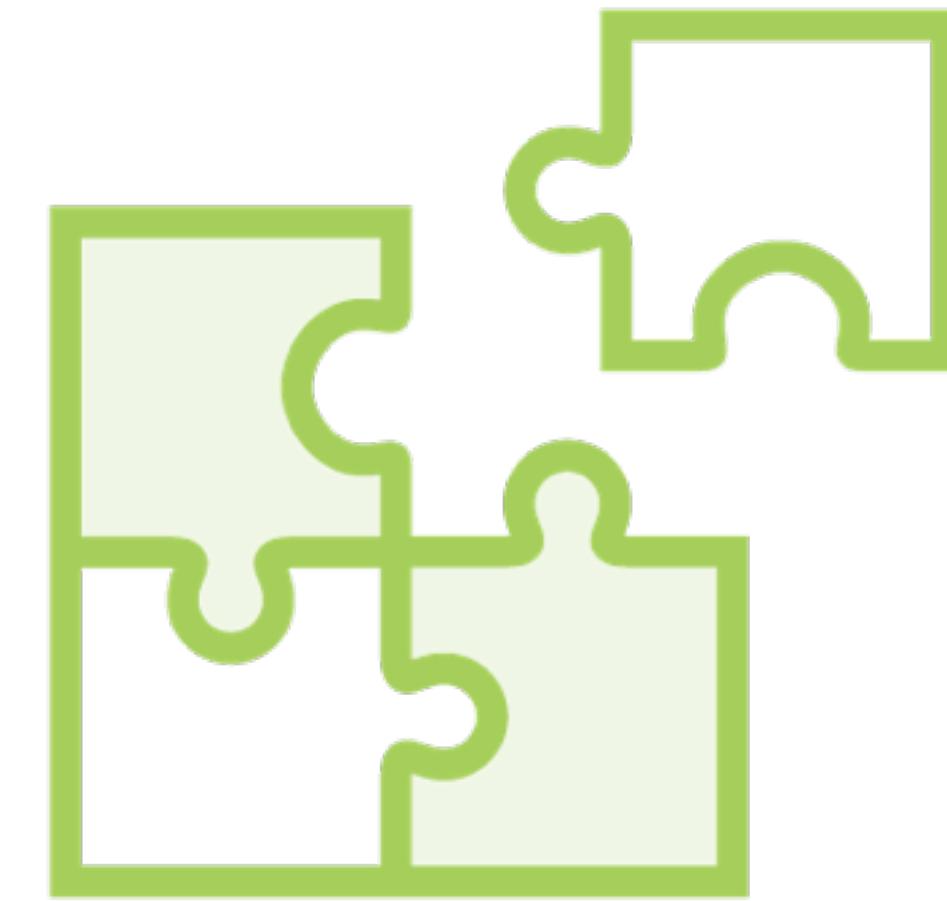


**Run queries on subsets that
are manageable**

Queries on Big Data



**Partitioning and
Bucketing of Tables**

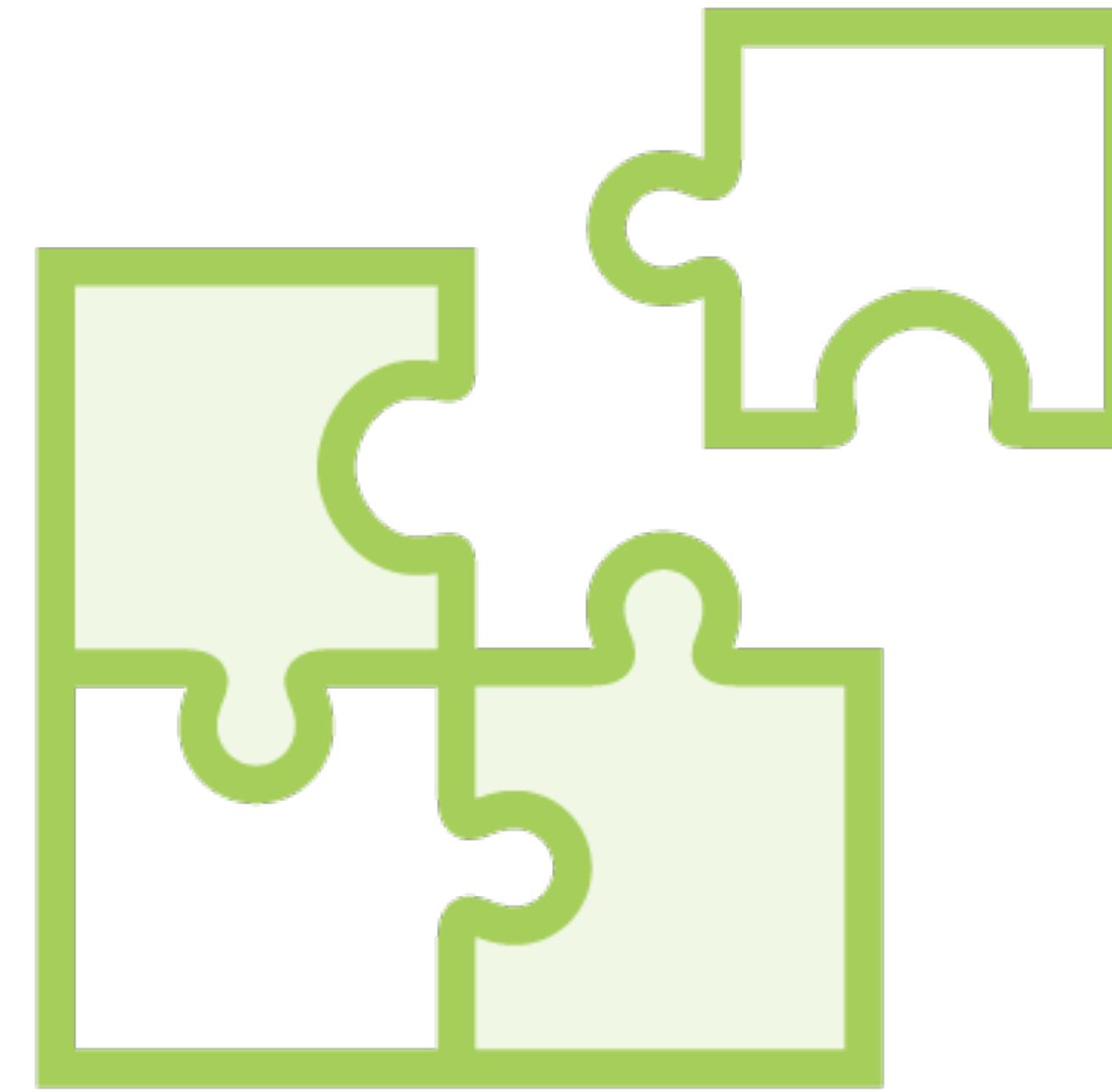


Join Optimizations



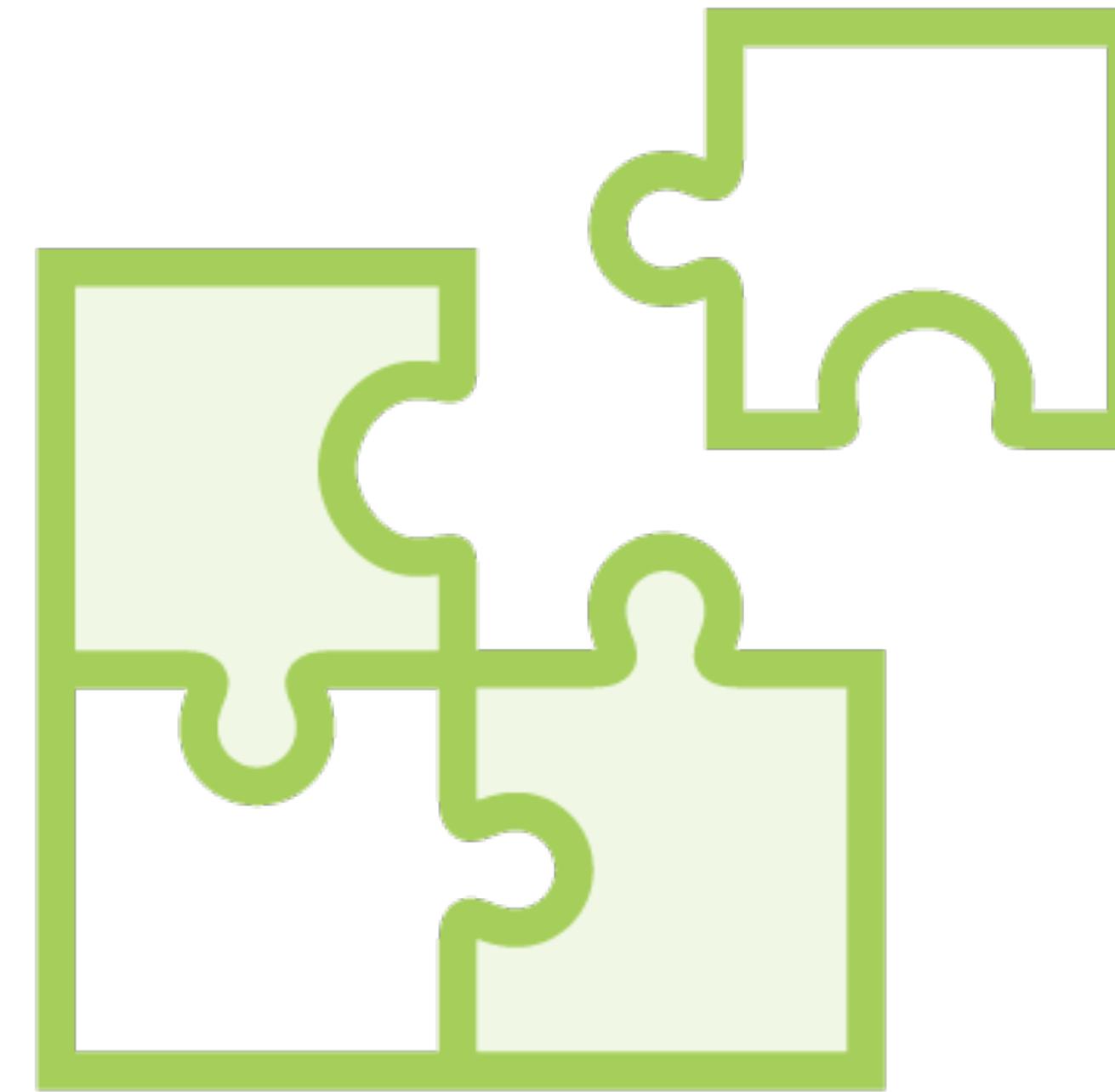
Window Functions

Join Optimizations



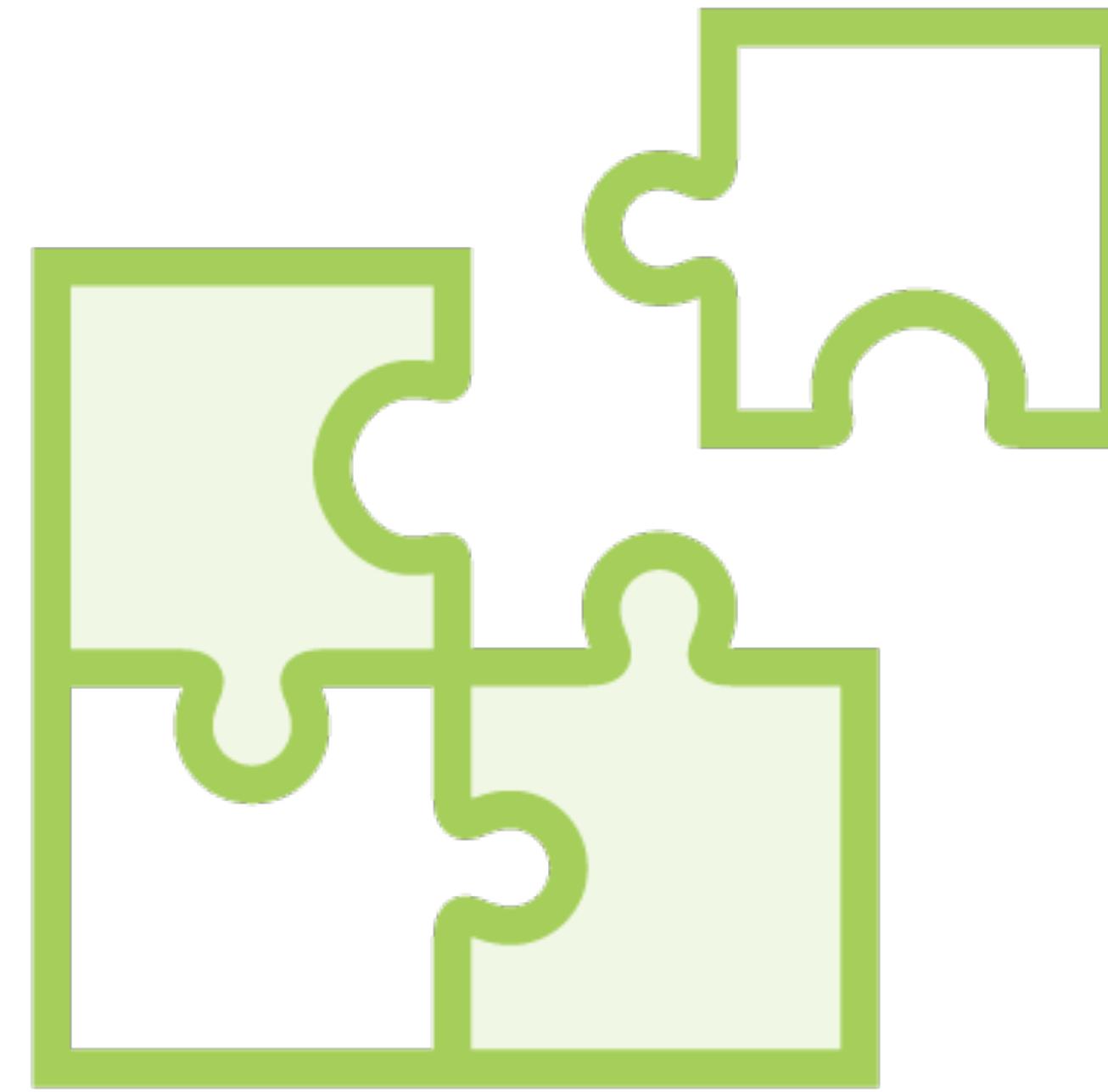
**Join operations are MapReduce
jobs under the hood**

Join Optimizations



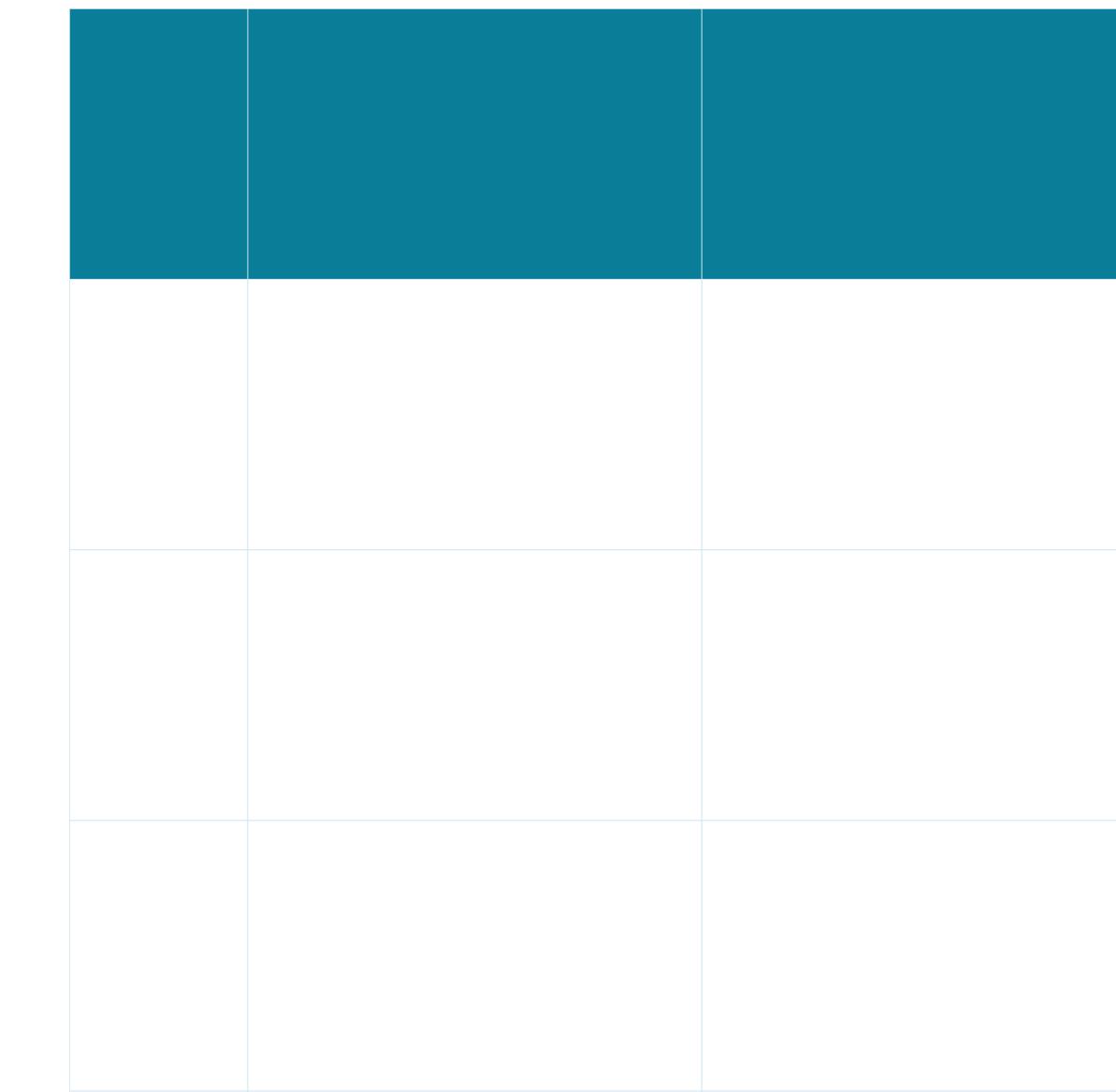
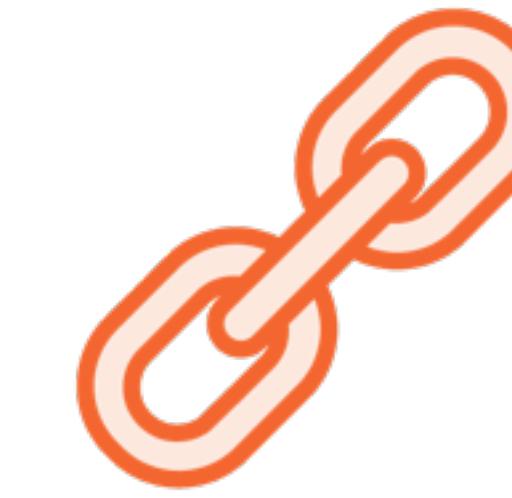
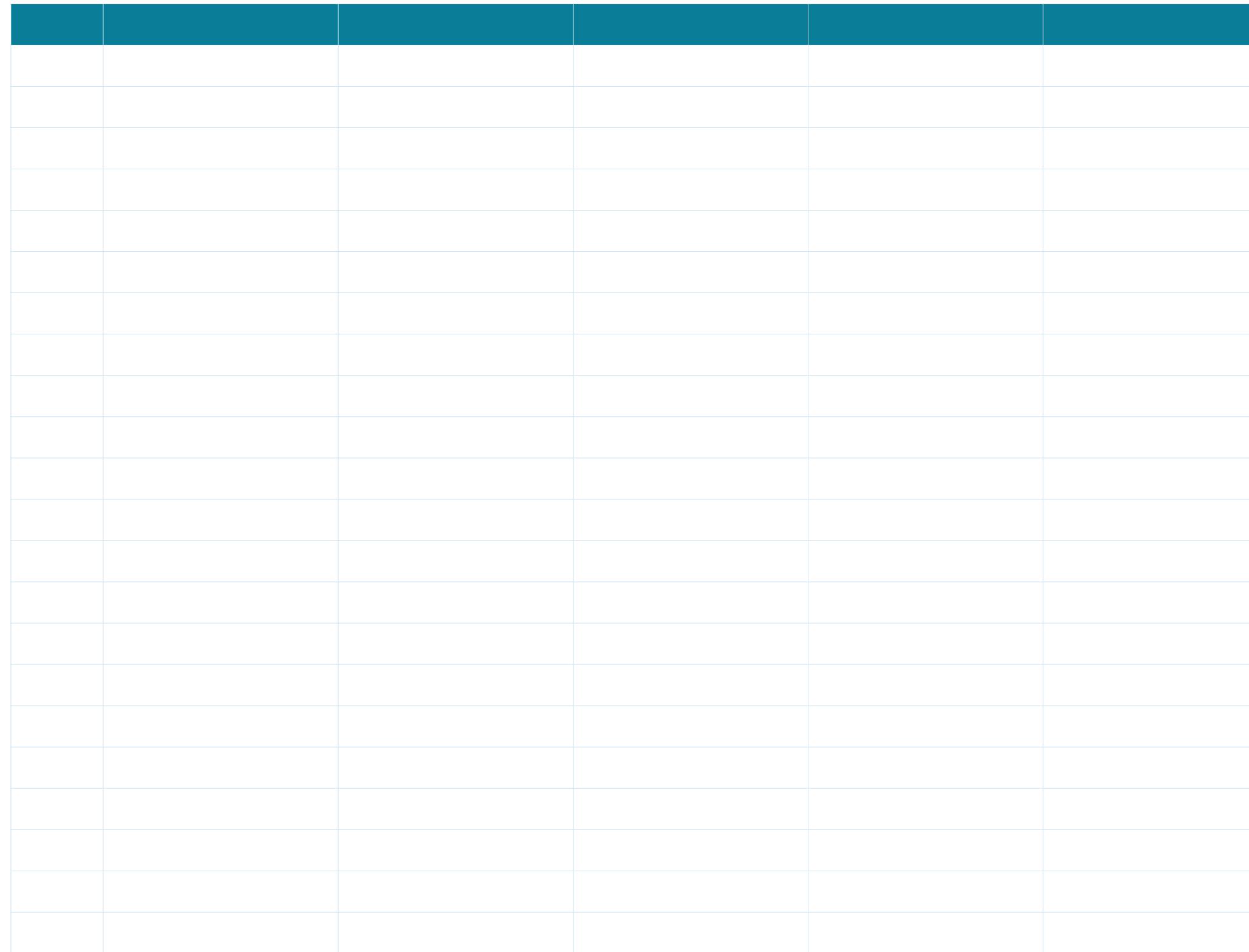
Optimize joins by **reducing the amount of data held in memory**

Join Optimizations



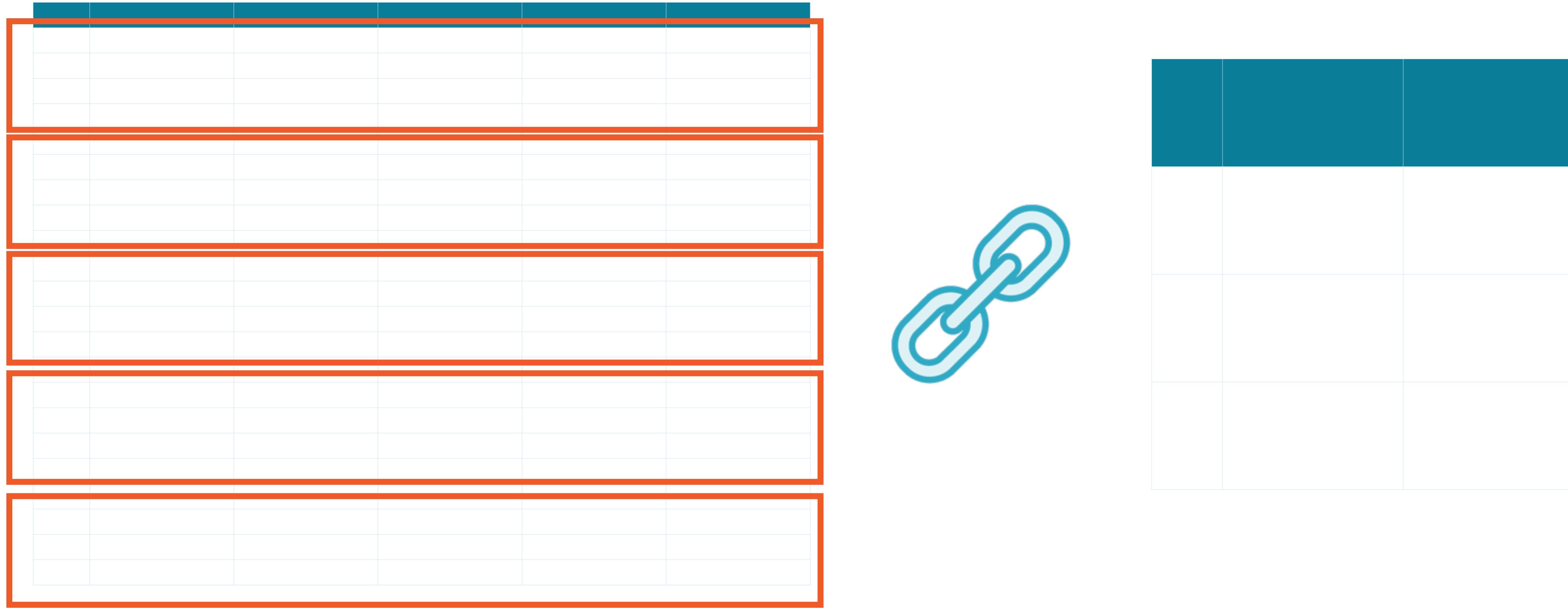
Or by structuring joins as a **map-only** operation

Reducing Data Held in Memory



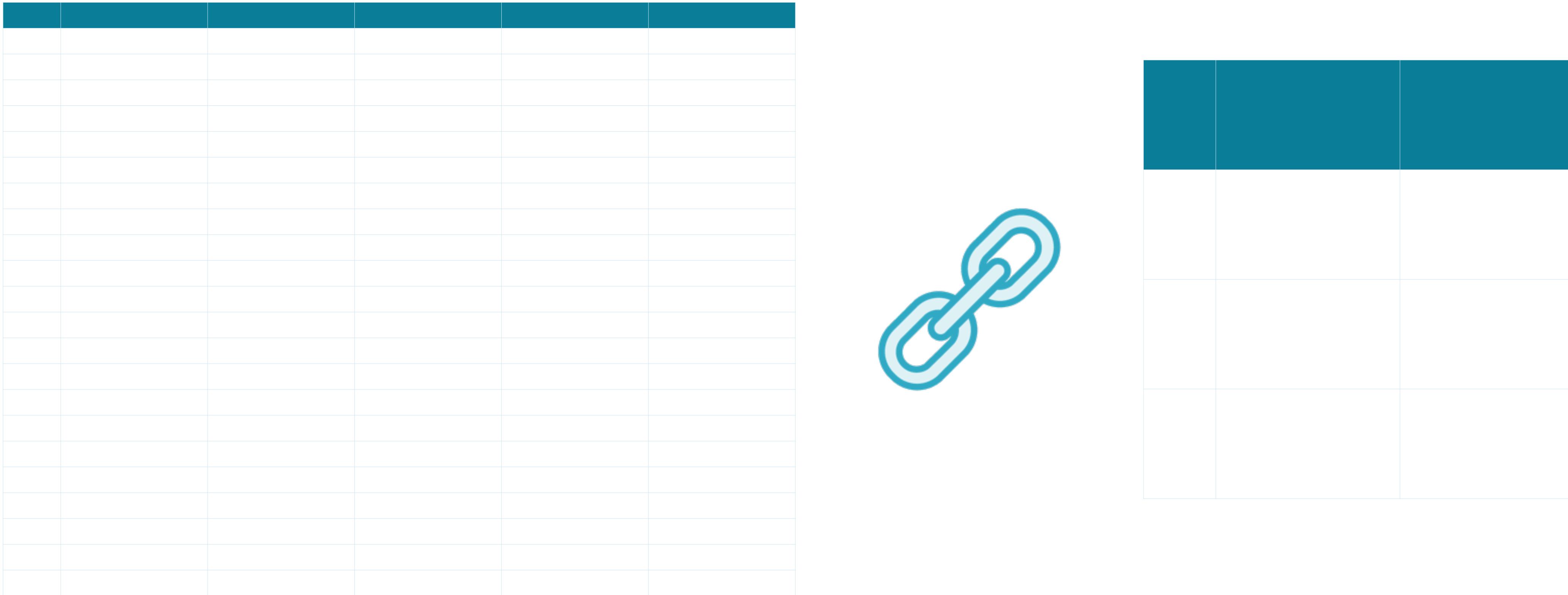
A 500GB table joined with a 5MB table

Reducing Data Held in Memory



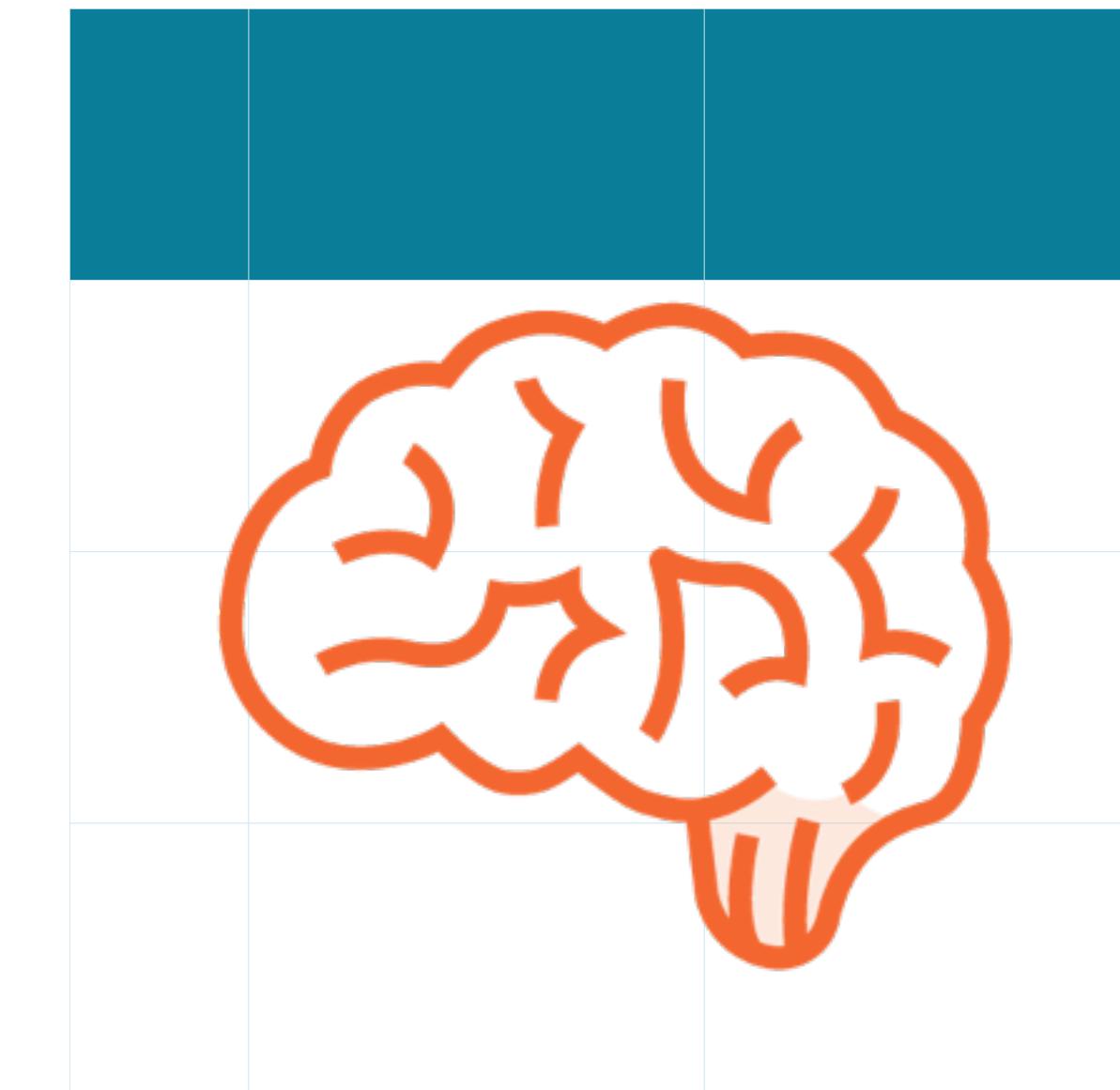
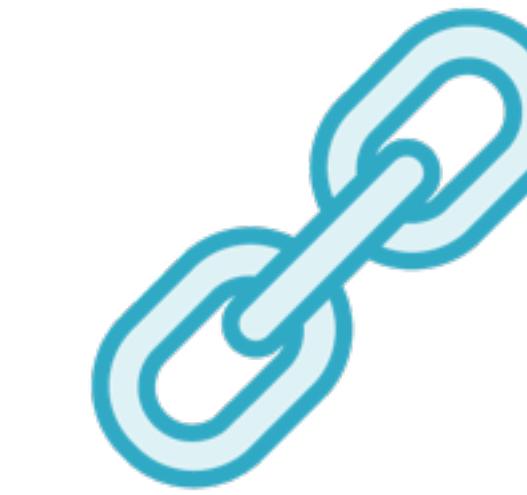
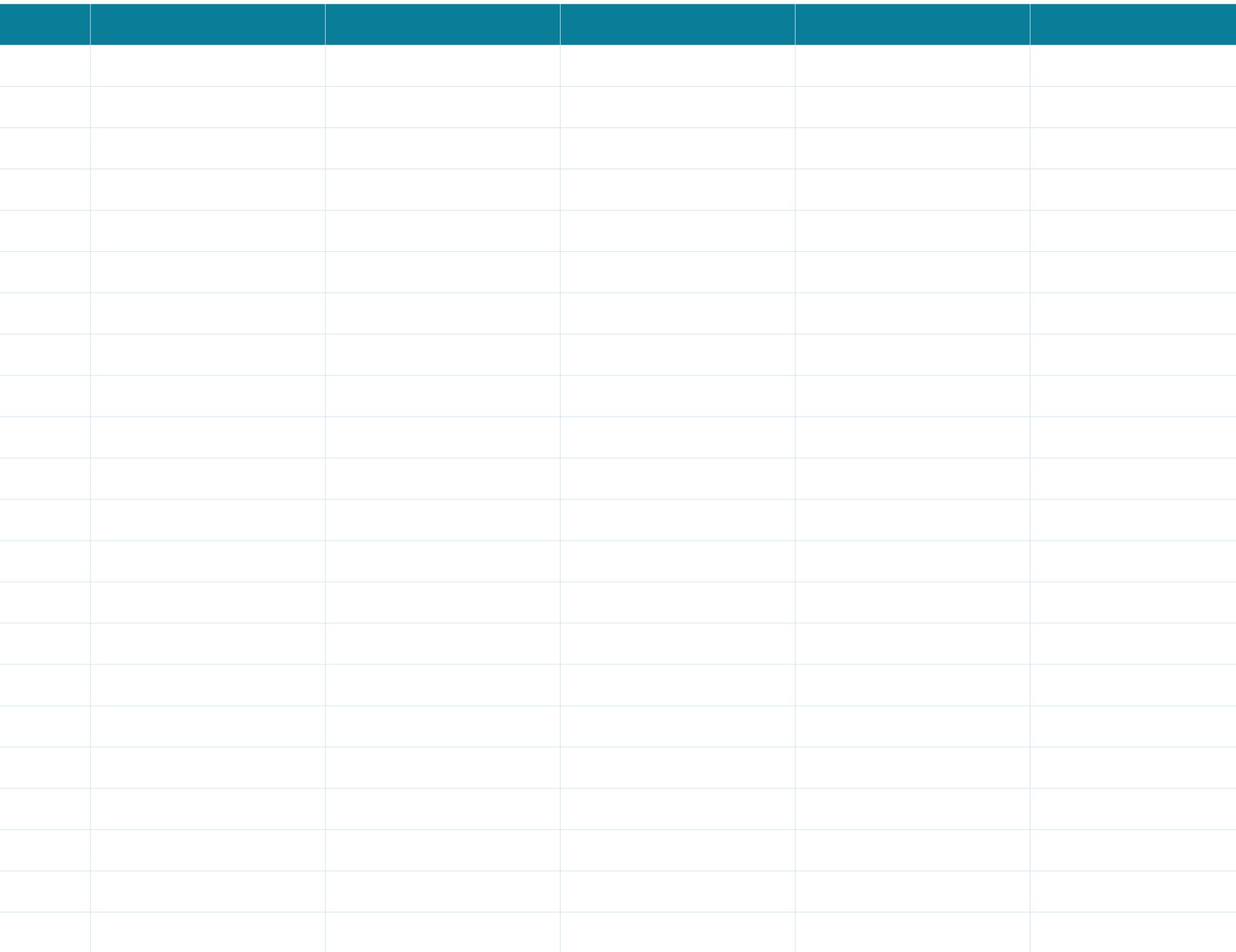
The large table will be **split** across
multiple machines in the cluster

Reducing Data Held in Memory



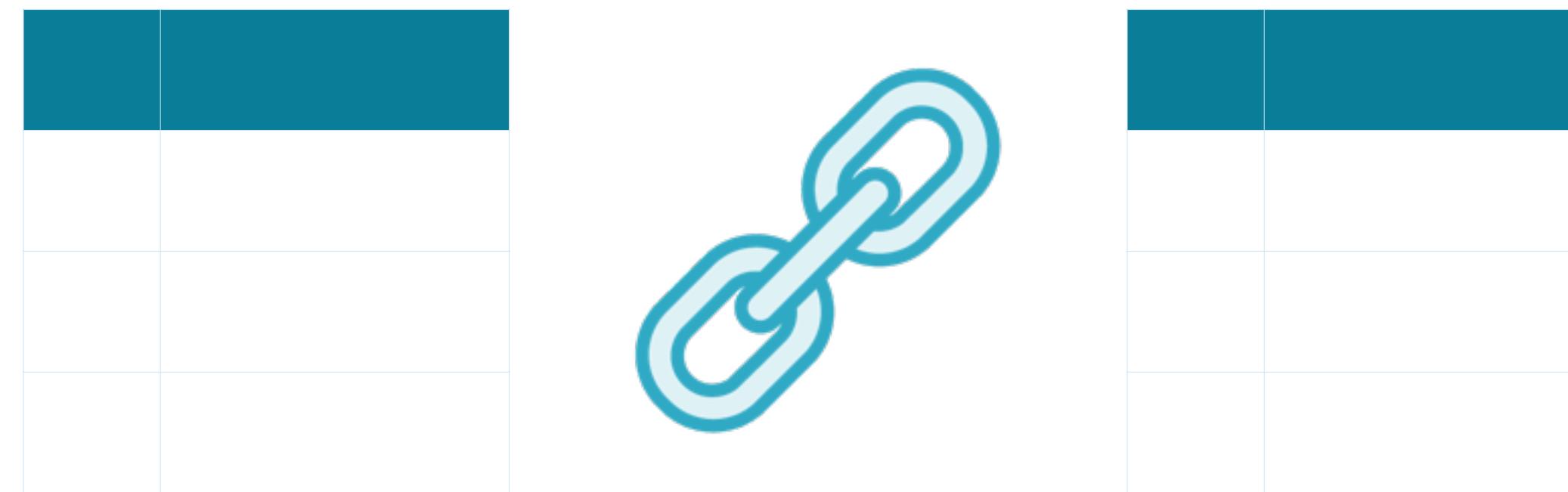
One table is held in memory while
the other is read from disk

Reducing Data Held in Memory



For better performance the **smaller** table should be held in **memory**

Joins as Map-only Operations



**MapReduce operations have
2 phases of processing**

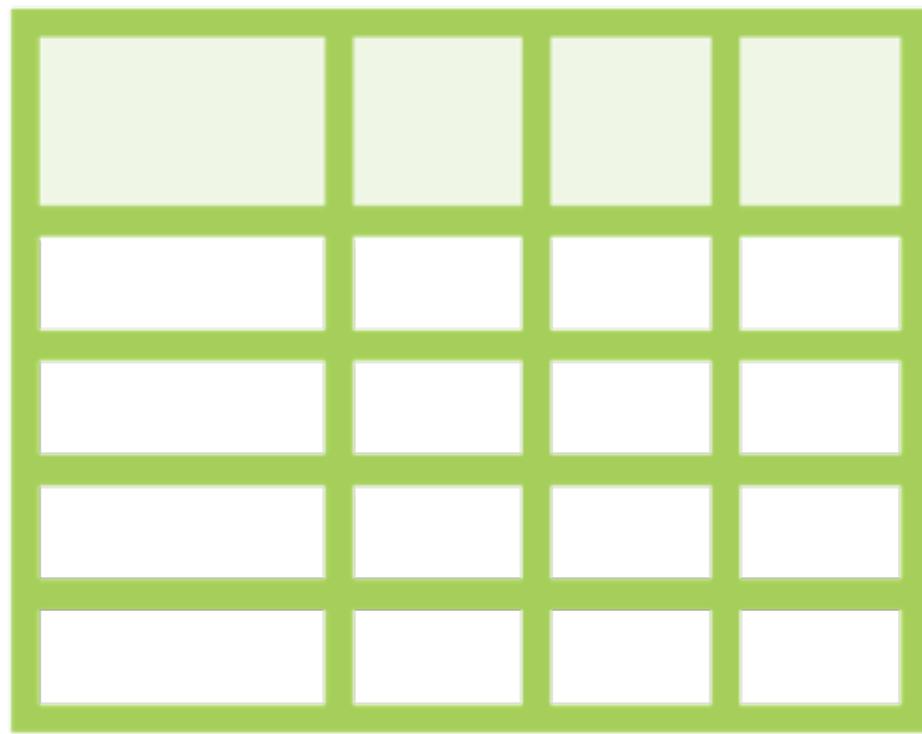


Joins as Map-only Operations

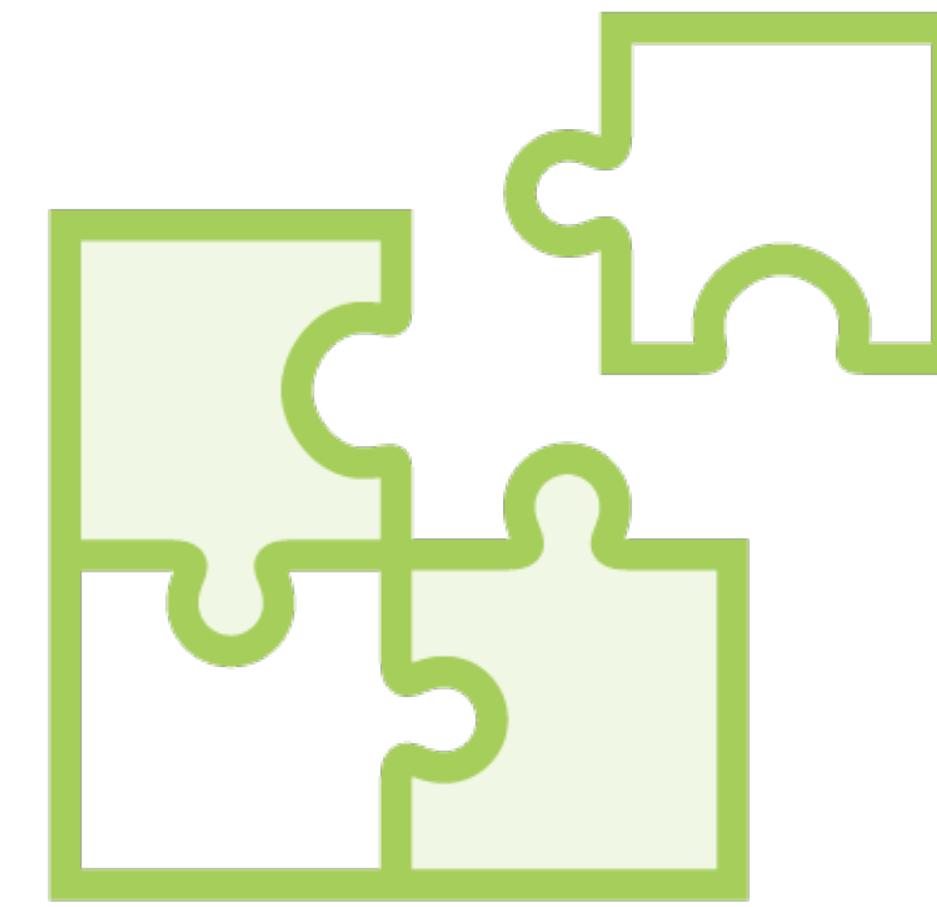


Certain queries can be
structured to have **no**
reduce phase

Queries on Big Data



**Partitioning and
Bucketing of Tables**



Join Optimizations



Window Functions

Window Functions



A suite of functions which are
syntactic sugar for complex queries

Window Functions



Make complex operations **simple** without
needing many **intermediate** calculations

Window Functions



**Reduces the need for intermediate
tables to store temporary data**



Window Functions

What were the top selling N products in last week's sale?

Window = one week

Operation = ranking product sales



Window Functions

What revenue percentile did this supplier fall into for this quarter?

Window = one quarter

Operation = percentiles on revenues

Window Functions



A suite of functions which are
syntactic sugar for complex queries

Window Functions

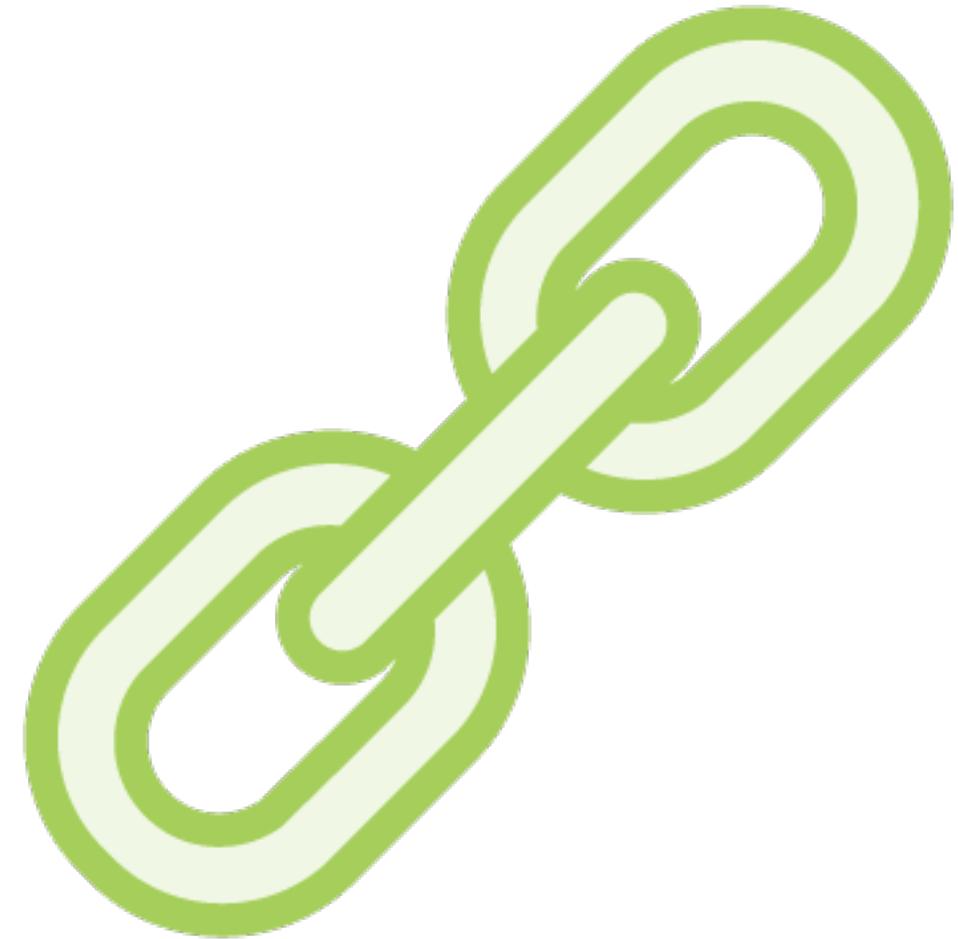


Make complex operations **simple** without
needing many **intermediate** calculations

Window Functions



**Reduces the need for intermediate
tables to store temporary data**



Window Functions

What were the top selling N products in last week's sale?

Window = one week

Operation = ranking product sales



Window Functions

What revenue percentile did this supplier fall into for this quarter?

Window = one quarter

Operation = percentiles on revenues



Window Functions

What were the top selling N products in last week's sale?

What revenue percentile did this supplier fall into for this quarter?

Can be expressed
in a single query

Running Total of Revenue

ID	Store	Product	Date	Revenue
o1	Seattle	Bananas	2017-01-01	7
o2	Kent	Apples	2017-01-02	20
o3	Bellevue	Flowers	2017-01-02	10
o4	Redmond	Meat	2017-01-03	40
o5	Seattle	Potatoes	2017-01-04	9
o6	Bellevue	Bread	2017-01-04	5
o7	Redmond	Bread	2017-01-05	5
o8	Issaquah	Onion	2017-01-05	4

Orders in a grocery store

Running Total of Revenue

ID	Store	Product	Date	Revenue
o1	Seattle	Bananas	2017-01-01	7
o2	Kent	Apples	2017-01-02	20
o3	Bellevue	Flowers	2017-01-02	10
o4	Redmond	Meat	2017-01-03	40
o5	Seattle	Potatoes	2017-01-04	9
o6	Bellevue	Bread	2017-01-04	5
o7	Redmond	Bread	2017-01-05	5
o8	Issaquah	Onion	2017-01-05	4

Sorted by order ID

Running Total of Revenue

ID	Store	Product	Date	Revenue	Running Total
o1	Seattle	Bananas	2017-01-01	7	7
o2	Kent	Apples	2017-01-02	20	27
o3	Bellevue	Flowers	2017-01-02	10	37
o4	Redmond	Meat	2017-01-03	40	77
o5	Seattle	Potatoes	2017-01-04	9	86
o6	Bellevue	Bread	2017-01-04	5	91
o7	Redmond	Bread	2017-01-05	5	96
o8	Issaquah	Onion	2017-01-05	4	10

Running Total of Revenue

ID	Store	Product	Date	Revenue	Running Total
o1	Seattle	Bananas	2017-01-01	7	7
o2	Kent	Apples	2017-01-02	20	27
o3	Bellevue	Flowers	2017-01-02	10	37
o4	Redmond	Meat	2017-01-03	40	77
o5	Seattle	Potatoes	2017-01-04	9	86
o6	Bellevue	Bread	2017-01-04	5	91
o7	Redmond	Bread	2017-01-05	5	96
o8	Issaquah	Onion	2017-01-05	4	100

Running Total of Revenue

ID	Store	Product	Date	Revenue	Running Total
o1	Seattle	Bananas	2017-01-01	7	7
o2	Kent	Apples	2017-01-02	20	27
o3	Bellevue	Flowers	2017-01-02	10	37
o4	Redmond	Meat	2017-01-03	40	77
o5	Seattle	Potatoes	2017-01-04	9	86
o6	Bellevue	Bread	2017-01-04	5	91
o7	Redmond	Bread	2017-01-05	5	96
o8	Issaquah	Onion	2017-01-05	4	100

Running Total of Revenue

ID	Store	Product	Date	Revenue	Running Total
o1	Seattle	Bananas	2017-01-01	7	7
o2	Kent	Apples	2017-01-02	20	27
o3	Bellevue	Flowers	2017-01-02	10	37
o4	Redmond	Meat	2017-01-03	40	77
o5	Seattle	Potatoes	2017-01-04	9	86
o6	Bellevue	Bread	2017-01-04	5	91
o7	Redmond	Bread	2017-01-05	5	96
o8	Issaquah	Onion	2017-01-05	4	10

Running Total of Revenue

ID	Store	Product	Date	Revenue	Running Total
o1	Seattle	Bananas	2017-01-01	7	7
o2	Kent	Apples	2017-01-02	20	27
o3	Bellevue	Flowers	2017-01-02	10	37
o4	Redmond	Meat	2017-01-03	40	77
o5	Seattle	Potatoes	2017-01-04	9	86
o6	Bellevue	Bread	2017-01-04	5	91
o7	Redmond	Bread	2017-01-05	5	96
o8	Issaquah	Onion	2017-01-05	4	10

Running Total of Revenue

ID	Store	Product	Date	Revenue	Running Total
o1	Seattle	Bananas	2017-01-01	7	7
o2	Kent	Apples	2017-01-02	20	27
o3	Bellevue	Flowers	2017-01-02	10	37
o4	Redmond	Meat	2017-01-03	40	77
o5	Seattle	Potatoes	2017-01-04	9	86
o6	Bellevue	Bread	2017-01-04	5	91
o7	Redmond	Bread	2017-01-05	5	96
o8	Issaquah	Onion	2017-01-05	4	10

For each row:
Calculate sum over all rows till the current row

Running Total of Revenue

ID	Store	Product	Date	Revenue	Running Total
o1	Seattle	Bananas	2017-01-01	7	7
o2	Kent	Apples	2017-01-02	20	27
o3	Bellevue	Flowers	2017-01-02	10	37
o4	Redmond	Meat	2017-01-03	40	77
o5	Seattle	Potatoes	2017-01-04	9	86
o6	Bellevue	Bread	2017-01-04	5	91
o7	Redmond	Bread	2017-01-05	5	96
o8	Issaquah	Onion	2017-01-05	4	100

Operation = sum

Running Total of Revenue

ID	Store	Product	Date	Revenue	Running Total
o1	Seattle	Bananas	2017-01-01	7	7
o2	Kent	Apples	2017-01-02	20	27
o3	Bellevue	Flowers	2017-01-02	10	37
o4	Redmond	Meat	2017-01-03	40	77
o5	Seattle	Potatoes	2017-01-04	9	86
o6	Bellevue	Bread	2017-01-04	5	91
o7	Redmond	Bread	2017-01-05	5	96
o8	Issaquah	Onion	2017-01-05	4	10

**Window is all rows from the top
to the current row**

Per Day Revenue Totals

ID	Store	Product	Date	Revenue
o1	Seattle	Bananas	2017-01-11	7
o2	Kent	Apples	2017-01-11	20
o3	Bellevue	Flowers	2017-01-11	10
o4	Redmond	Meat	2017-01-12	40
o5	Seattle	Potatoes	2017-01-12	9
o6	Bellevue	Bread	2017-01-12	5
o7	Redmond	Bread	2017-01-13	5
o8	Issaquah	Onion	2017-01-13	4

A revenue running total for each day

Per Day Revenue Totals

ID	Store	Product	Date	Revenue
o1	Seattle	Bananas	2017-01-11	7
o2	Kent	Apples	2017-01-11	20
o3	Bellevue	Flowers	2017-01-11	10
o4	Redmond	Meat	2017-01-12	40
o5	Seattle	Potatoes	2017-01-12	9
o6	Bellevue	Bread	2017-01-12	5
o7	Redmond	Bread	2017-01-13	5
o8	Issaquah	Onion	2017-01-13	4

Reset to 0 when a new day begins

Per Day Revenue Totals

ID	Store	Product	Date	Revenue
o1	Seattle	Bananas	2017-01-11	7
o2	Kent	Apples	2017-01-11	20
o3	Bellevue	Flowers	2017-01-11	10
o4	Redmond	Meat	2017-01-12	40
o5	Seattle	Potatoes	2017-01-12	9
o6	Bellevue	Bread	2017-01-12	5
o7	Redmond	Bread	2017-01-13	5
o8	Issaquah	Onion	2017-01-13	4

Operate on blocks of one day

Moving Averages

ID	Store	Product	Date	Revenue
o1	Seattle	Bananas	2017-01-11	7
o2	Kent	Apples	2017-01-11	20
o3	Bellevue	Flowers	2017-01-11	10
o4	Redmond	Meat	2017-01-12	40
o5	Seattle	Potatoes	2017-01-12	9
o6	Bellevue	Bread	2017-01-12	5
o7	Redmond	Bread	2017-01-13	5
o8	Issaquah	Onion	2017-01-13	4

Calculate the average for the last 3 items sold

Moving Averages

ID	Store	Product	Date	Revenue
o1	Seattle	Bananas	2017-01-11	7
o2	Kent	Apples	2017-01-11	20
o3	Bellevue	Flowers	2017-01-11	10
o4	Redmond	Meat	2017-01-12	40
o5	Seattle	Potatoes	2017-01-12	9
o6	Bellevue	Bread	2017-01-12	5
o7	Redmond	Bread	2017-01-13	5
o8	Issaquah	Onion	2017-01-13	4

3 previous rows from the current row

Moving Averages

ID	Store	Product	Date	Revenue
o1	Seattle	Bananas	2017-01-11	7
o2	Kent	Apples	2017-01-11	20
o3	Bellevue	Flowers	2017-01-11	10
o4	Redmond	Meat	2017-01-12	40
o5	Seattle	Potatoes	2017-01-12	9
o6	Bellevue	Bread	2017-01-12	5
o7	Redmond	Bread	2017-01-13	5
o8	Issaquah	Onion	2017-01-13	4

3 previous rows from the current row

Moving Averages

ID	Store	Product	Date	Revenue
o1	Seattle	Bananas	2017-01-11	7
o2	Kent	Apples	2017-01-11	20
o3	Bellevue	Flowers	2017-01-11	10
o4	Redmond	Meat	2017-01-12	40
o5	Seattle	Potatoes	2017-01-12	9
o6	Bellevue	Bread	2017-01-12	5
o7	Redmond	Bread	2017-01-13	5
o8	Issaquah	Onion	2017-01-13	4

3 previous rows from the current row

Percentage of the Total per Day

ID	Store	Product	Date	Revenue
o1	Seattle	Bananas	2017-01-11	7
o2	Kent	Apples	2017-01-11	20
o3	Bellevue	Flowers	2017-01-11	10
o4	Redmond	Meat	2017-01-12	40
o5	Seattle	Potatoes	2017-01-12	9
o6	Bellevue	Bread	2017-01-12	5
o7	Redmond	Bread	2017-01-13	5
o8	Issaquah	Onion	2017-01-13	4

Orders for one day, total revenue = 37

Percentage of the Total per Day

ID	Store	Product	Date	Revenue
o1	Seattle	Bananas	2017-01-11	7
o2	Kent	Apples	2017-01-11	20
o3	Bellevue	Flowers	2017-01-11	10
o4	Redmond	Meat	2017-01-12	40
o5	Seattle	Potatoes	2017-01-12	9
o6	Bellevue	Bread	2017-01-12	5
o7	Redmond	Bread	2017-01-13	5
o8	Issaquah	Onion	2017-01-13	4

% contribution of o1 = $7/37 * 100 = 18.9\%$

Percentage of the Total per Day

ID	Store	Product	Date	Revenue
o1	Seattle	Bananas	2017-01-11	7
o2	Kent	Apples	2017-01-11	20
o3	Bellevue	Flowers	2017-01-11	10
o4	Redmond	Meat	2017-01-12	40
o5	Seattle	Potatoes	2017-01-12	9
o6	Bellevue	Bread	2017-01-12	5
o7	Redmond	Bread	2017-01-13	5
o8	Issaquah	Onion	2017-01-13	4

% contribution of o2 = $20 / 37 * 100 = 54\%$

Percentage of the Total per Day

ID	Store	Product	Date	Revenue
o1	Seattle	Bananas	2017-01-11	7
o2	Kent	Apples	2017-01-11	20
o3	Bellevue	Flowers	2017-01-11	10
o4	Redmond	Meat	2017-01-12	40
o5	Seattle	Potatoes	2017-01-12	9
o6	Bellevue	Bread	2017-01-12	5
o7	Redmond	Bread	2017-01-13	5
o8	Issaquah	Onion	2017-01-13	4

% contribution of o3 = $10 / 37 * 100 = 27\%$

Percentage of the Total per Day

$$\% \text{ contribution} = \frac{\text{Revenue} \times 100}{\text{Total Revenue}}$$

Percentage of the Total per Day

$$\% \text{ contribution} = \frac{\text{Revenue} \times 100}{\text{sum(revenue) over (partition by day)}}$$

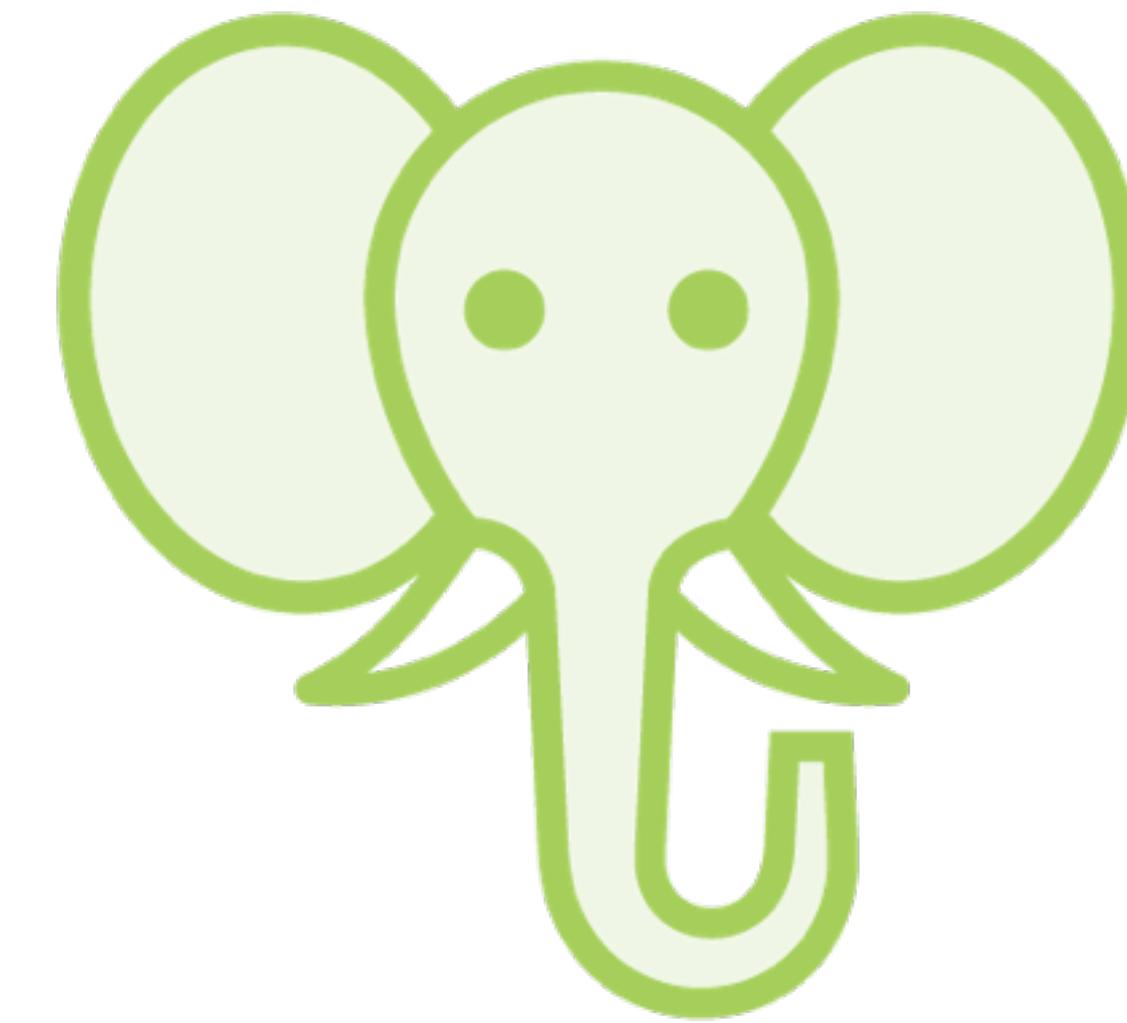
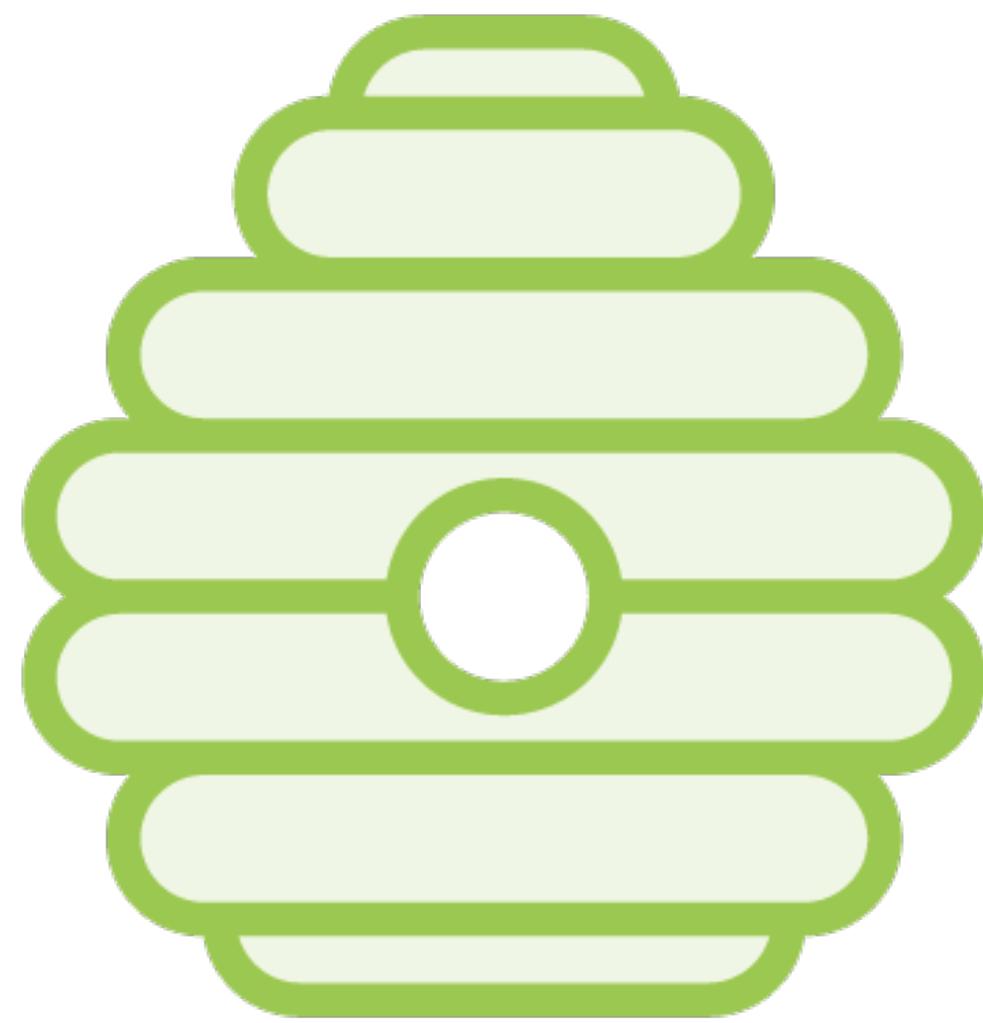
Introducing Pig

Hive on Hadoop



**Hive runs all processes in the form of
MapReduce jobs under the hood**

Hive on Hadoop



Allows processing of huge datasets

Errr... okay....

Where Does Data Come From?



Where Does Data Come From?



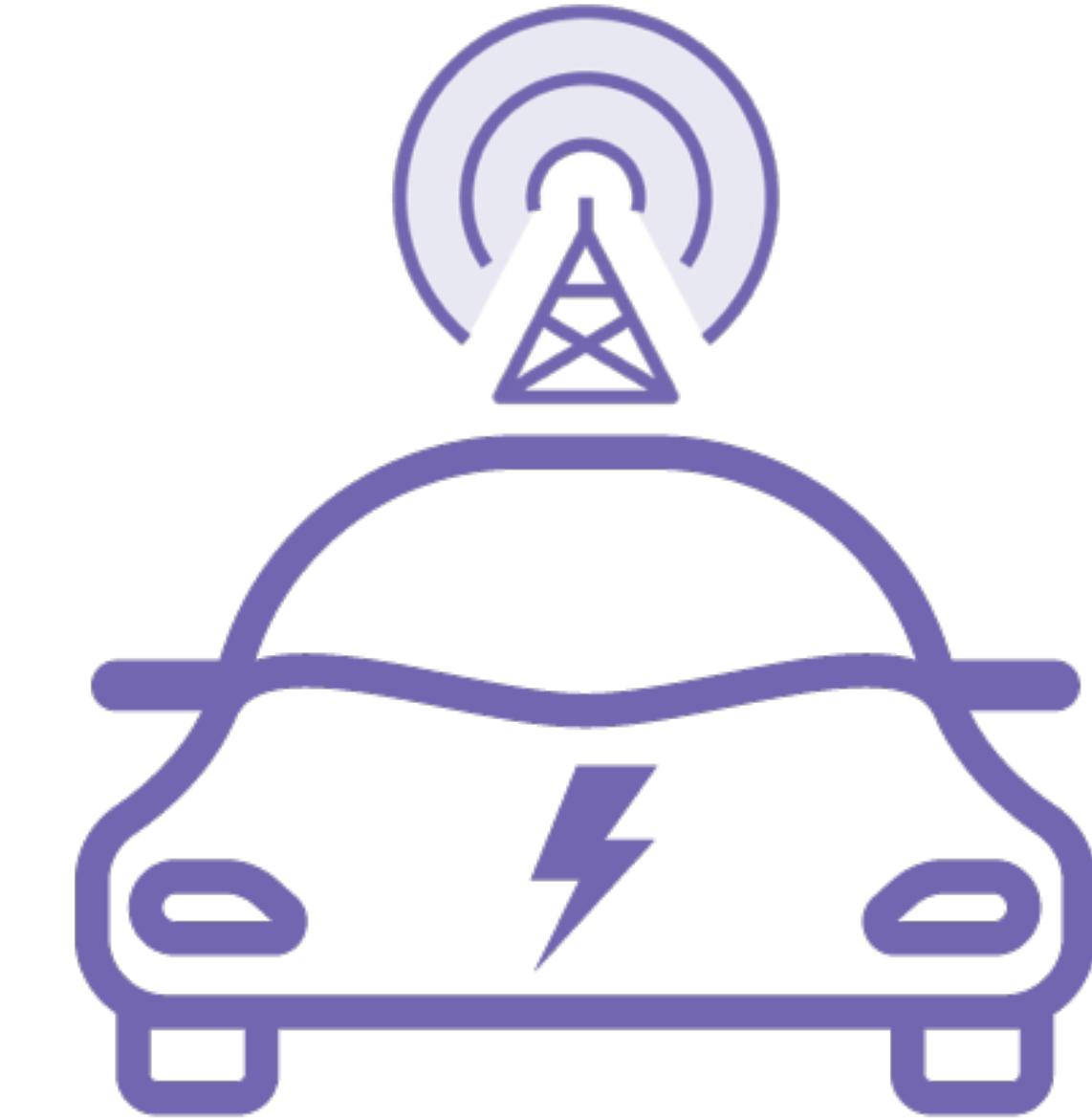
Where Does Data Come From?



Mobile phones



GPS location coordinates



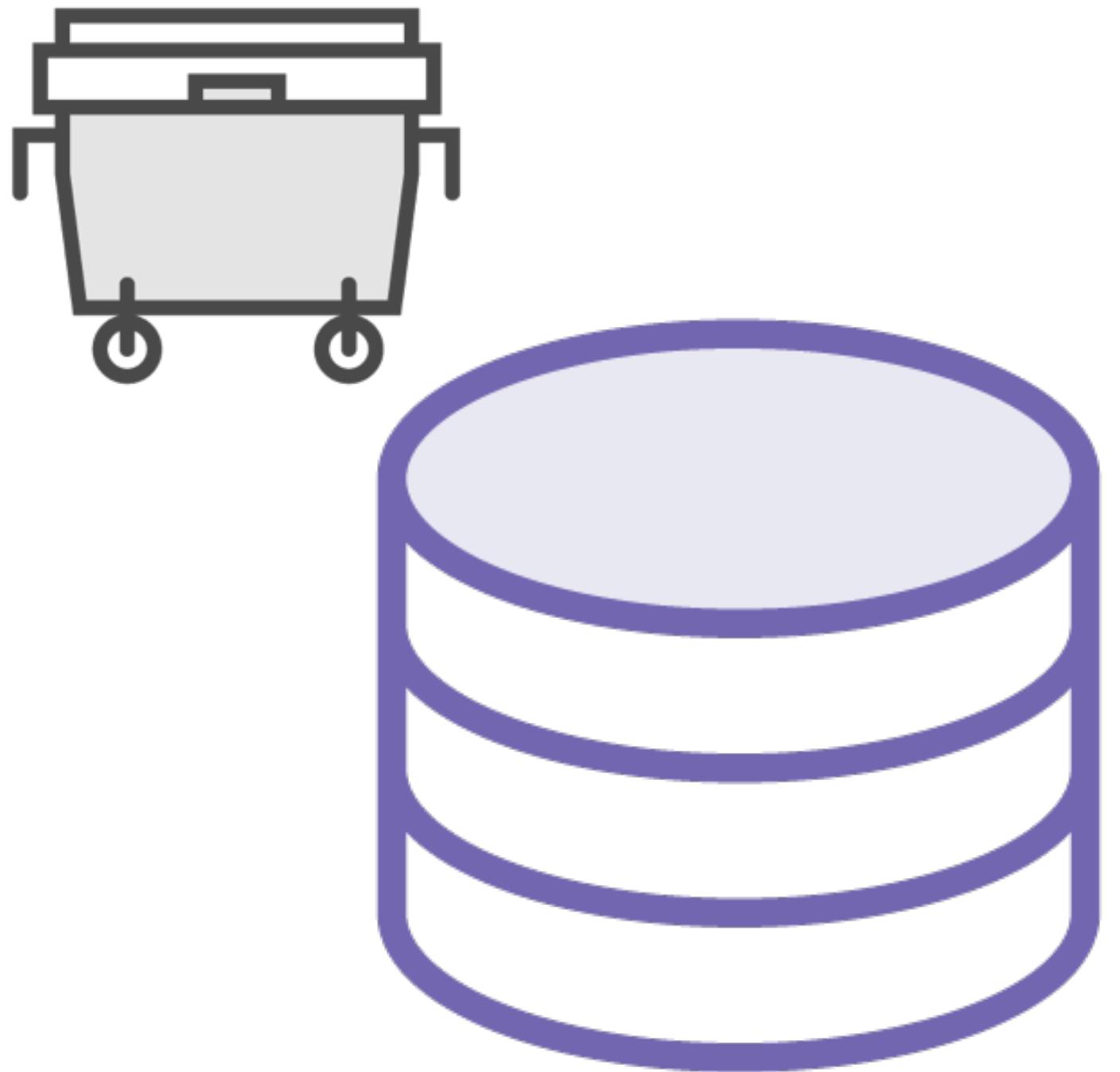
Self driving car sensors

Where Is This Data Stored?



**Files on some storage
system**

Characteristics of Data

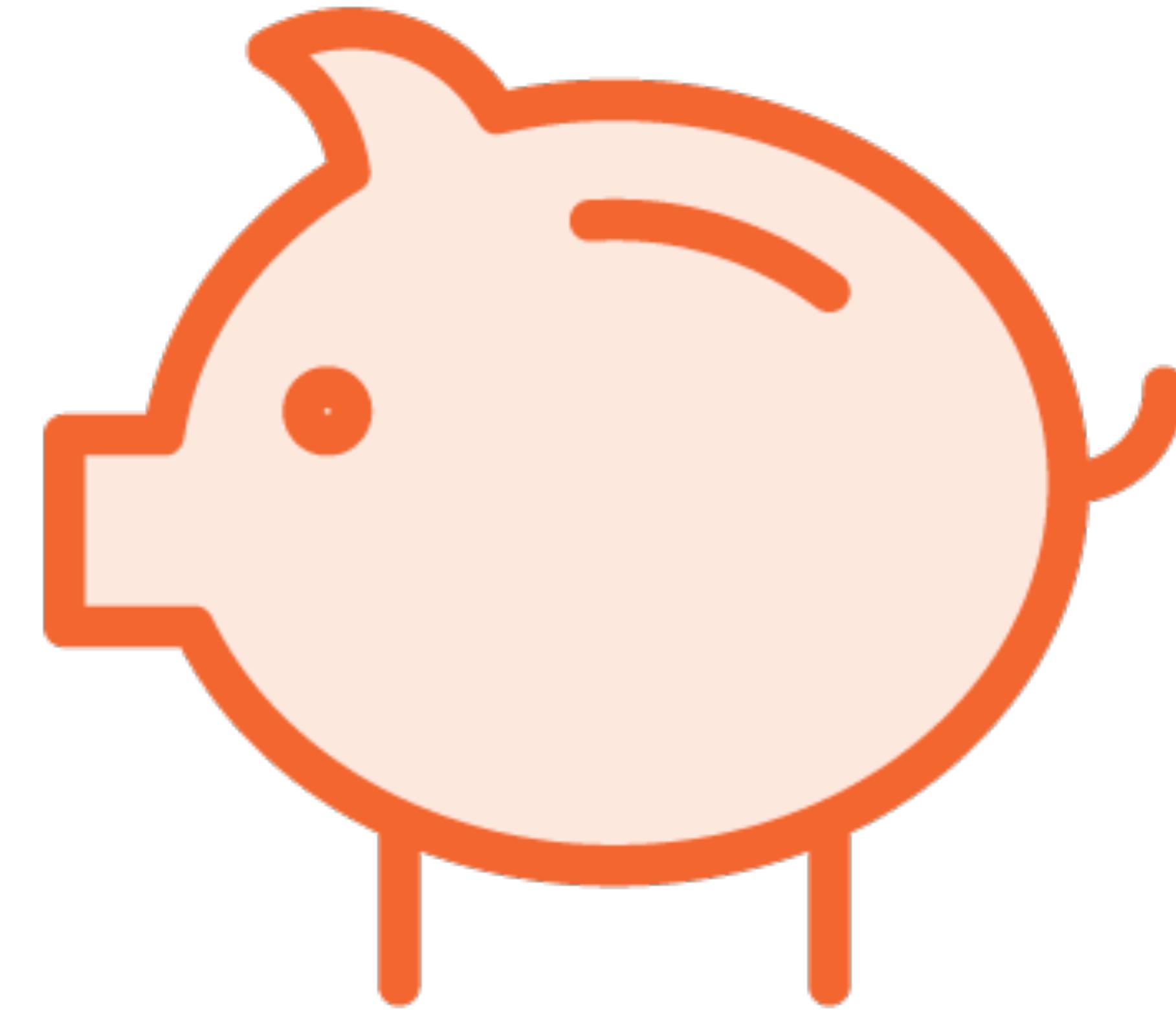


Unknown schema

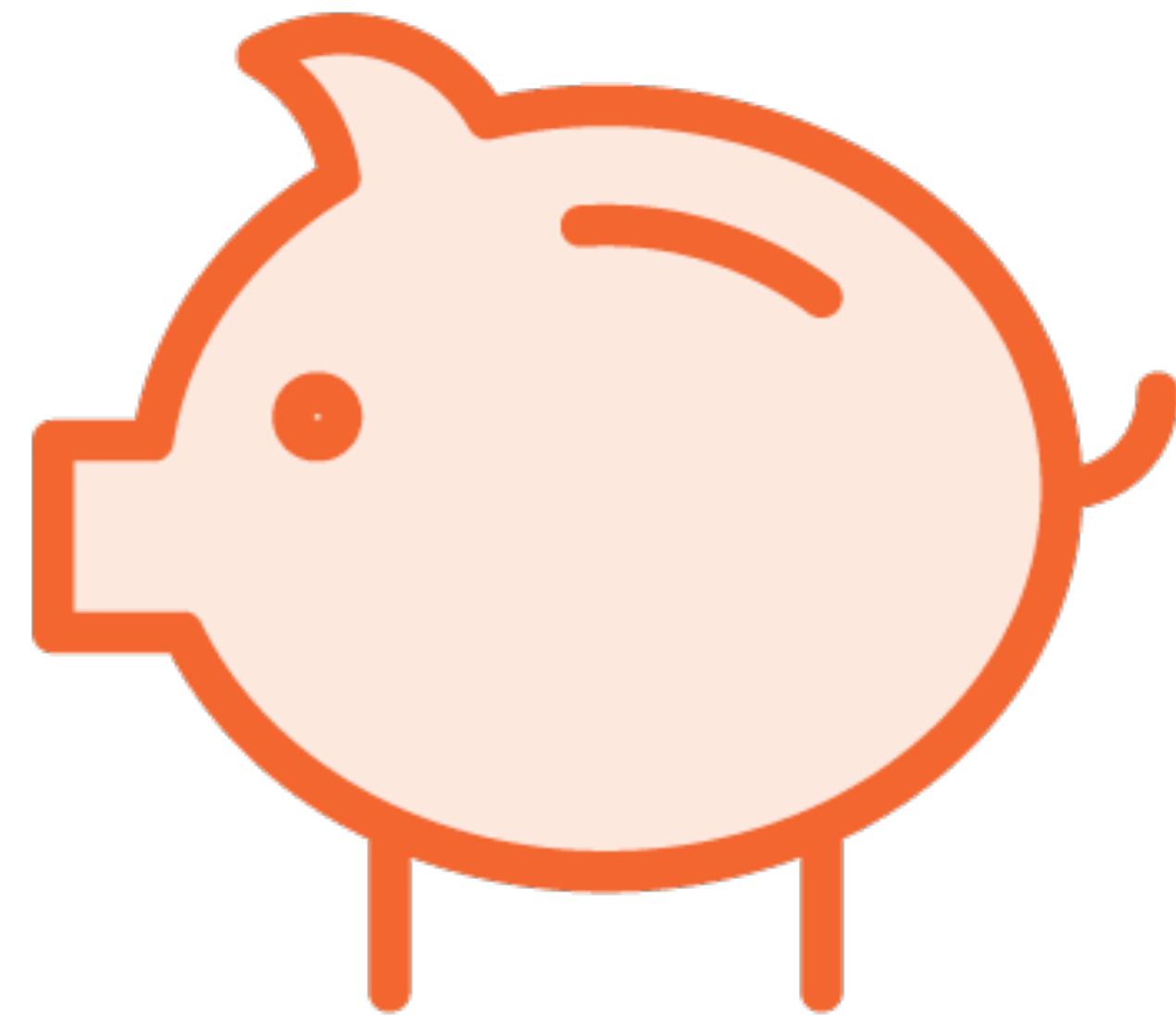
Incomplete data

Inconsistent records

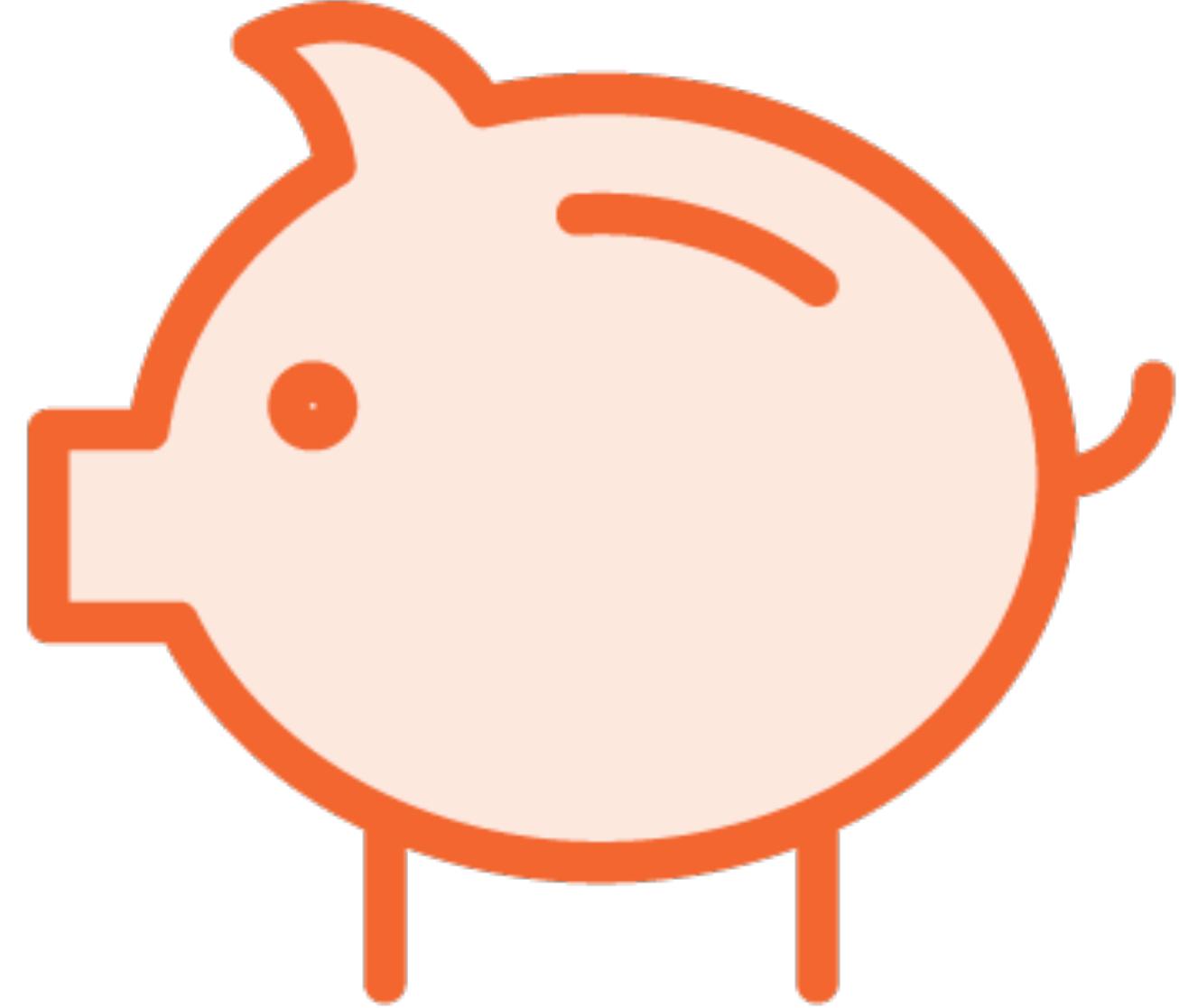
Apache Pig to the Rescue



Apache Pig to the Rescue



A high level scripting language to
work with data with **unknown** or
inconsistent schema



Pig

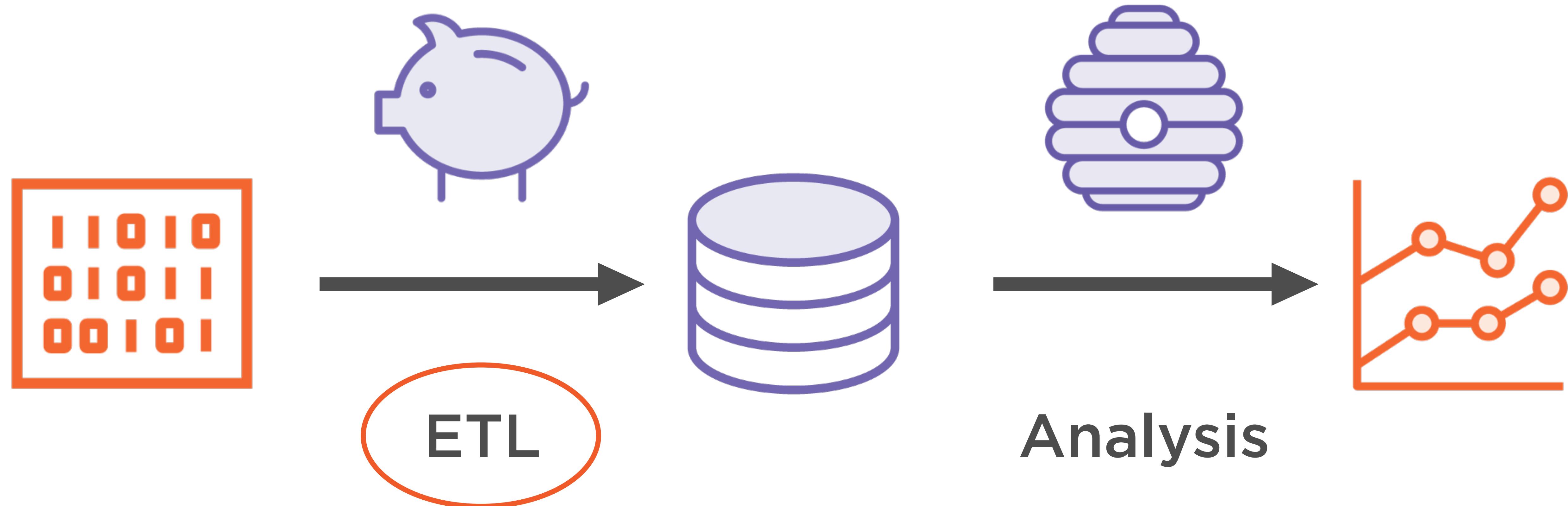
Part of the Hadoop eco-system

Works well with unstructured,
incomplete data

Can work directly on files in HDFS

Used to get data **into** a
data warehouse

Pig Complements Hive



Extract, Transform, Load



Pull unstructured, inconsistent data from source, clean it and place it in another database where it can be analyzed

Extract, Transform, Load



Pull unstructured, inconsistent data from source, clean it and place it in another database where it can be analyzed

Extract, Transform, Load



Pull unstructured, inconsistent data from source, **clean it** and place it in another database where it can be analyzed

Extract, Transform, Load



Pull unstructured, inconsistent data from source, clean it and place it in another database where it can be analyzed

Apache Pig

```
64.242.88.10 - - [07/Mar/2004:16:05:49 -0800] "GET /twiki/bin/edit/Main/Double_bounce_sender?topicparent>Main.ConfigurationVariables HTTP/1.1" 401 12846
64.242.88.10 - - [07/Mar/2004:16:06:51 -0800] "GET /twiki/bin/rdiff/TWiki/NewUserTemplate?rev1=1.3&rev2=1.2 HTTP/1.1" 200 4523
64.242.88.10 - - [07/Mar/2004:16:10:02 -0800] "GET /mailman/listinfo/hsdivision HTTP/1.1" 200 6291
64.242.88.10 - - [07/Mar/2004:16:11:58 -0800] "GET /twiki/bin/view/TWiki/WikiSyntax HTTP/1.1" 200 7352
64.242.88.10 - - [07/Mar/2004:16:20:55 -0800] "GET /twiki/bin/view/Main/DCCAndPostFix HTTP/1.1" 200 5253
64.242.88.10 - - [07/Mar/2004:16:23:12 -0800] "GET /twiki/bin/oops/TWiki/AppendixFileSystem?template=oopsmore¶m1=1.12¶m2=1.12 HTTP/1.1" 200 11382
64.242.88.10 - - [07/Mar/2004:16:24:16 -0800] "GET /twiki/bin/view/Main/PeterThoeny HTTP/1.1" 200 4924
64.242.88.10 - - [07/Mar/2004:16:29:16 -0800] "GET /twiki/bin/edit/Main/Header_checks?topicparent>Main.ConfigurationVariables HTTP/1.1" 401 12851
64.242.88.10 - - [07/Mar/2004:16:30:29 -0800] "GET /twiki/bin/attach/Main/OfficeLocations HTTP/1.1" 401 12851
64.242.88.10 - - [07/Mar/2004:16:31:48 -0800] "GET /twiki/bin/view/TWiki/WebTopicEditTemplate HTTP/1.1" 200 3732
64.242.88.10 - - [07/Mar/2004:16:32:50 -0800] "GET /twiki/bin/view/Main/WebChanges HTTP/1.1" 200 40520
64.242.88.10 - - [07/Mar/2004:16:33:53 -0800] "GET /twiki/bin/edit/Main/Smtpd_etrn_restrictions?topicparent>Main.ConfigurationVariables HTTP/1.1" 401 12851
64.242.88.10 - - [07/Mar/2004:16:35:19 -0800] "GET /mailman/listinfo/business HTTP/1.1" 200 6379
64.242.88.10 - - [07/Mar/2004:16:36:22 -0800] "GET /twiki/bin/rdiff/Main/WebIndex?rev1=1.2&rev2=1.1 HTTP/1.1" 200 46373
64.242.88.10 - - [07/Mar/2004:16:37:27 -0800] "GET /twiki/bin/view/TWiki/DontNotify HTTP/1.1" 200 4140
64.242.88.10 - - [07/Mar/2004:16:39:24 -0800] "GET /twiki/bin/view/Main/TokyoOffice HTTP/1.1" 200 3853
64.242.88.10 - - [07/Mar/2004:16:43:54 -0800] "GET /twiki/bin/view/Main/MikeMannix HTTP/1.1" 200 3686
64.242.88.10 - - [07/Mar/2004:16:45:56 -0800] "GET /twiki/bin/attach/Main/PostfixCommands HTTP/1.1" 401 12846
64.242.88.10 - - [07/Mar/2004:16:47:12 -0800] "GET /robots.txt HTTP/1.1" 200 68
64.242.88.10 - - [07/Mar/2004:16:47:46 -0800] "GET /twiki/bin/rdiff/Know/ReadmeFirst?rev1=1.5&rev2=1.4 HTTP/1.1" 200 5724
64.242.88.10 - - [07/Mar/2004:16:49:04 -0800] "GET /twiki/bin/view/Main/TWikiGroups?rev=1.2 HTTP/1.1" 200 5162
64.242.88.10 - - [07/Mar/2004:16:50:54 -0800] "GET /twiki/bin/rdiff/Main/ConfigurationVariables HTTP/1.1" 200 59679
64.242.88.10 - - [07/Mar/2004:16:52:35 -0800] "GET /twiki/bin/edit/Main/Flush_service_name?topicparent>Main.ConfigurationVariables HTTP/1.1" 401 12851
```

Apache Pig

```
64.242.88.10 - - [07/Mar/2004:16:05:49 -0800] "GET /twiki/bin/  
edit/Main/Double_bounce_sender?  
topicparent>Main.ConfigurationVariables HTTP/1.1" 401 12846
```

Server IP Address

Apache Pig

```
64.242.88.10 - - [07/Mar/2004:16:05:49 -0800] "GET /twiki/bin/  
edit/Main/Double_bounce_sender?  
topicparent>Main.ConfigurationVariables HTTP/1.1" 401 12846
```

Date and Time

Apache Pig

```
64.242.88.10 - - [07/Mar/2004:16:05:49 -0800] "GET /twiki/bin/  
edit/Main/Double_bounce_sender?  
topicparent>Main.ConfigurationVariables HTTP/1.1" 401 12846
```

Request Type

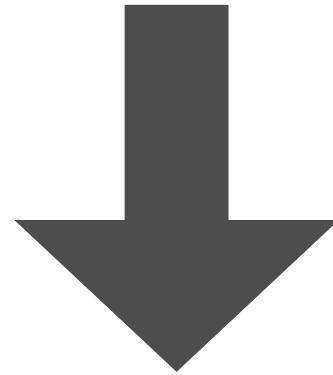
Apache Pig

```
64 242 88 10 - - [07/Mar/2004:16:05:49 -0800] "GET /twiki/bin/  
edit/Main/Double_bounce_sender?  
topicparent>Main.ConfigurationVariables HTTP/1.1" 401 12846
```

URL

Apache Pig

```
64.242.88.10 - - [07/Mar/2004:16:05:49 -0800] "GET /twiki/bin/  
edit/Main/Double_bounce_sender?  
topicparent>Main.ConfigurationVariables HTTP/1.1" 401 12846
```



IP	Date	Time	Request Type	URL

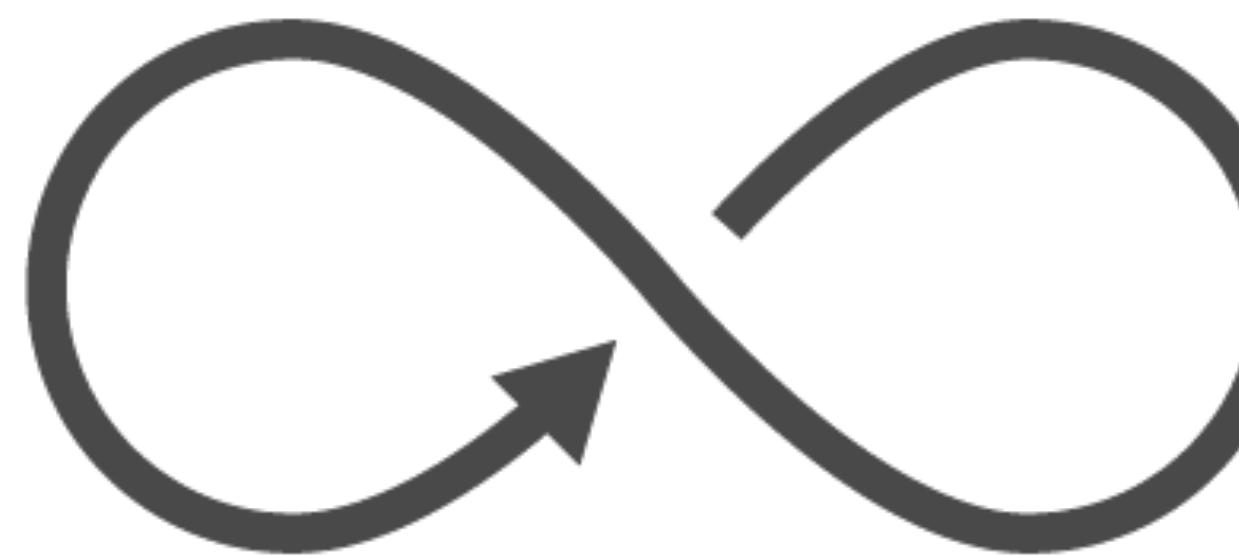
Pig Latin



A **procedural, data flow** language to extract,
transform and load data



Procedural

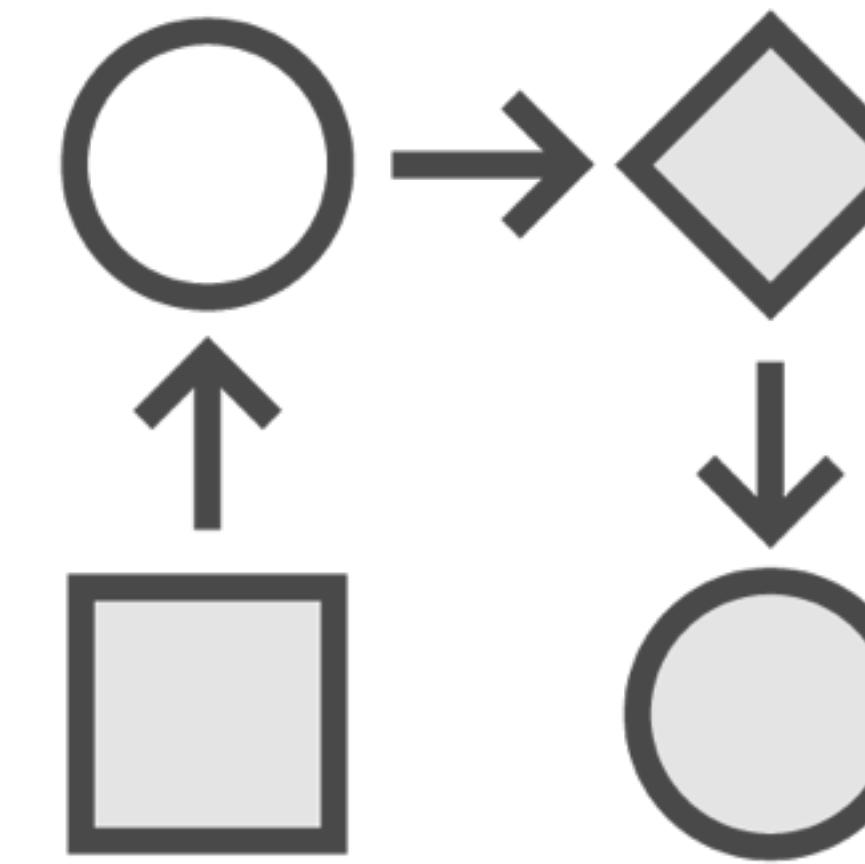


Series of **well-defined steps** to perform operations

No if statements or for loops



Data Flow



Focused on **transformations** applied to the data

Written with a **series** of data operations in mind

Pig Latin



**Data from one or more sources can be read,
processed and stored in parallel**

Pig Latin



**Cleans data, precomputes common
aggregates before storing in a data warehouse**

Pig vs. SQL

	Pig	SQL
(group revenues by dept) foreach generate sum(revenue)		select sum(revenue) from revenues group by dept

Pig vs. SQL

Pig

A **data flow language**, transforms data to store in a warehouse

Specifies **exactly how** data is to be modified at every step

Purpose of processing is to **store in a queryable format**

Used to **clean data** with inconsistent or incomplete schema

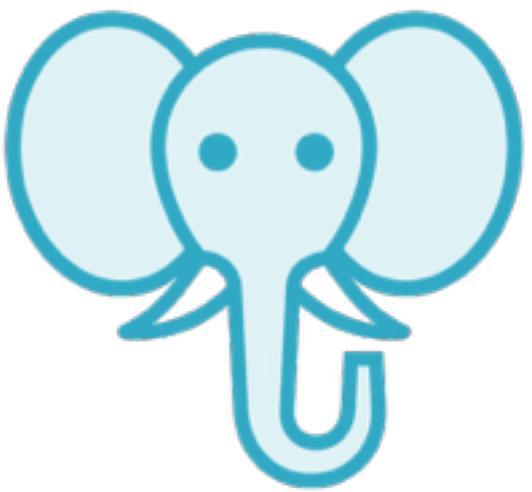
SQL

A **query language**, is used for retrieving results

Abstracts away how queries are executed

Purpose of data extraction is **analysis**

Extract insights, generate reports, drive decisions



Hadoop

HDFS

MapReduce

YARN

**File system to
manage the storage
of data**

**Framework to
process data across
multiple servers**

**Framework to run and
manage the data
processing tasks**

Pig on Hadoop



Pig runs on top of the Hadoop distributed computing framework

Pig on Hadoop



Reads files from HDFS, stores intermediate records in HDFS and writes its final output to HDFS

Pig on Hadoop



Decomposes operations into MapReduce jobs which run in parallel

Pig on Hadoop



Provides **non-trivial**, **built-in** implementations of standard data operations, which are very **efficient**

Pig on Hadoop



Pig optimizes operations **before** MapReduce jobs are run, to speed operations up

Pig on Hadoop

PIG

HDFS

MapReduce

YARN

Pig on Other Technologies

PIG

Apache Tez

Apache Spark

Pig on Other Technologies

PIG

Apache Tez

Apache Spark

Tez is an extensible framework which improves on MapReduce by making its operations faster

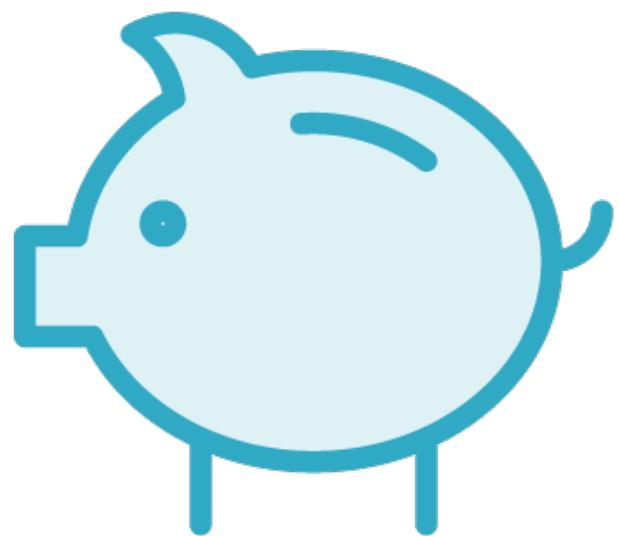
Pig on Other Technologies

PIG

Apache Tez

Apache Spark

Spark is another distributed computing technology which is scalable, flexible and fast



Pig vs. Hive

Pig

Used to extract, transform and load data **into a data warehouse**

Used by developers to bring together useful data in one place

Uses Pig Latin, a procedural, data flow language



Hive

Used to query data **from a data warehouse** to generate reports

Used by analysts to retrieve business information from data

Uses HiveQL, a structured query language

Introducing Spark

Spark

An engine for data processing and analysis



General Purpose



Interactive



Distributed
Computing

General Purpose



Exploring
Cleaning and Preparing
Applying Machine Learning
Building Data Applications

Spark

An engine for data processing and analysis



General Purpose



Interactive



Distributed
Computing

Interactive



`len([1, 2, 5])
=> 3`

REPL



Interactive

REPL

Read-Evaluate-Print-Loop

Interactive environments
Fast feedback

Spark

An engine for data processing and analysis



General Purpose



Interactive



Distributed
Computing

Distributed Computing



**Process data across a
cluster of machines**

Integrate with Hadoop

Read data from HDFS

Spark APIs

Scala

Python

Java

Almost all data is processed using

Resilient Distributed Datasets

Resilient Distributed Datasets

RDDs are the main programming abstraction in Spark

Resilient Distributed Datasets

RDDs are in-memory collections of objects

In-memory,
yet resilient!

Resilient Distributed Datasets

**With RDDs, you can interact and
play with billions of rows of data**

**...without caring about
any of the complexities**

Spark is made up of a few
different components

Spark Core

The basic functionality of Spark
RDDs

Spark Core

**Spark Core is just a computing engine
It needs two additional
components**

Spark Core

A Storage System
that stores the
data to be
processed

A Cluster Manager to
help Spark run tasks
across a cluster of
machines

Spark Core

Storage
System

Cluster
Manager

Both of these are plug
and play components

**Local file
system**

HDFS



Storage System

Spark Core

Storage
System

Cluster
Manager

**Built-in Cluster
Manager**

YARN



**Cluster
Manager**

Spark Core

Storage
System

Cluster
Manager

**Plug and Play makes it easy
to integrate with Hadoop**

A Hadoop Cluster

HDFS

For storage

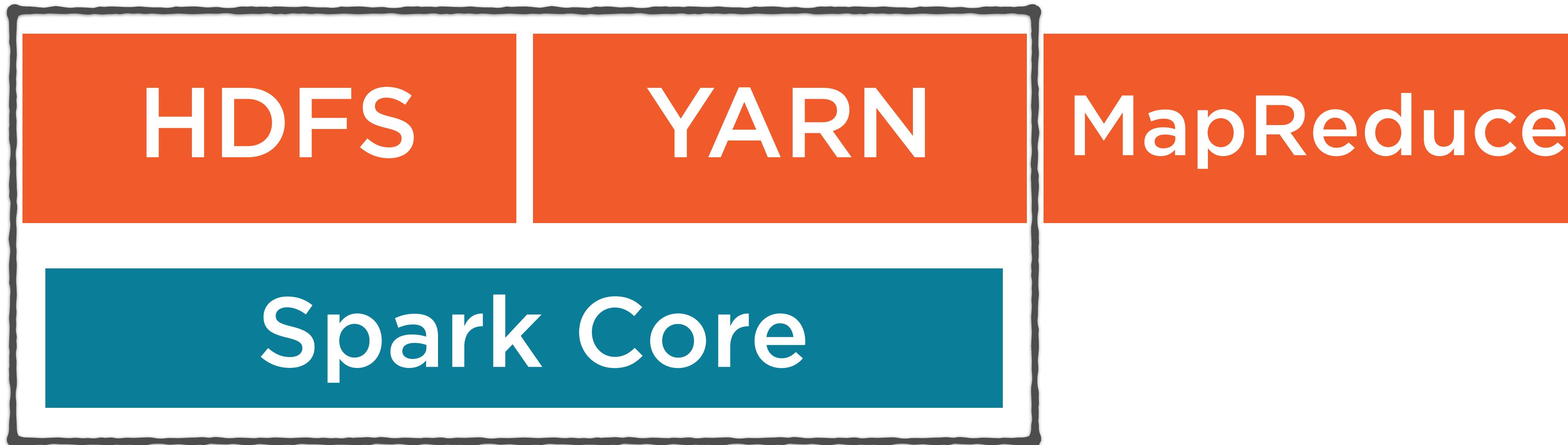
YARN

For managing
the cluster

MapReduce

For
computing

A Hadoop Cluster



Use Spark as an
alternate/additional
compute engine

Installing Spark

Prerequisites

Java 7 or above

Scala

IPython (Anaconda)

Installing Spark

Download Spark binaries

Update environment variables

Configure iPython Notebook for Spark

Spark Environment Variables

SPARK_HOME

Point to the folder where Spark has been extracted

PATH

`$SPARK_HOME/bin` Linux/Mac OS X

`%SPARK_HOME%/bin` Windows

Configuring IPython

PYSPARK_DRIVER_PYTHON

ipython

PYSPARK_DRIVER_PYTHON_OPTS

“notebook”

Launch PySpark

> pyspark

Launch PySpark

This is just like a Python shell

Launch PySpark

Use Python functions, dicts, lists etc

Launch PySpark

You can import and use any installed Python modules

Launch PySpark

Launches by default in a local non-distributed mode

SparkContext

When the shell is launched it initializes a SparkContext

SparkContext

```
Welcome to  
 _____  
 \  V /  
  \  V /  
   \  T /  
    \  /  
     \ /  
      \/  
      version 1.6.1  
  
Using Python version 2.7.11 (default, Jan 22 2016 08:29:18)  
SparkContext available as sc, HiveContext available as sqlContext.  
->>>
```

The **SparkContext** represents a connection to the Spark Cluster

SparkContext

```
Welcome to  
_____  
\\VV--VV--'T--/T--  
/ / . / , / / / / \ \ / /  
/ /  
version 1.6.1  
  
Using Python version 2.7.11 (default, Jan 22 2016 08:29:18)  
SparkContext available as sc, HiveContext available as sqlContext.  
>>> |
```

Used to load data into memory
from a specified source

SparkContext

The data gets loaded into an RDD

Resilient
Distributed
Datasets

Partitions
Read-only
Lineage

RDDs
represent data
in-memory

Partitions

1	Swetha	30	Bangalore
2	Vitthal	35	New Delhi
3	Navdeep	25	Mumbai
4	Janani	35	New Delhi
5	Navdeep	25	Mumbai
6	Janani	35	New Delhi

**Data is divided
into partitions**

**Distributed to
multiple
machines**

Partitions

1	Swetha	30	Bangalore
2	Vitthal	35	New Delhi
3	Navdeep	25	Mumbai
4	Janani	35	New Delhi
5	Navdeep	25	Mumbai
6	Janani	35	New Delhi

**Data is divided
into partitions**

Partitions

1	Swetha	30	Bangalore
2	Vitthal	35	New Delhi
3	Navdeep	25	Mumbai
4	Janani	35	New Delhi
5	Navdeep	25	Mumbai
6	Janani	35	New Delhi

**Data is divided
into partitions**

Partitions

1	Swetha	30	Bangalore
2	Vitthal	35	New Delhi

3	Navdeep	25	Mumbai
4	Janani	35	New Delhi

5	Navdeep	25	Mumbai
6	Janani	35	New Delhi

**Distributed to
multiple
machines,
called nodes**

Partitions

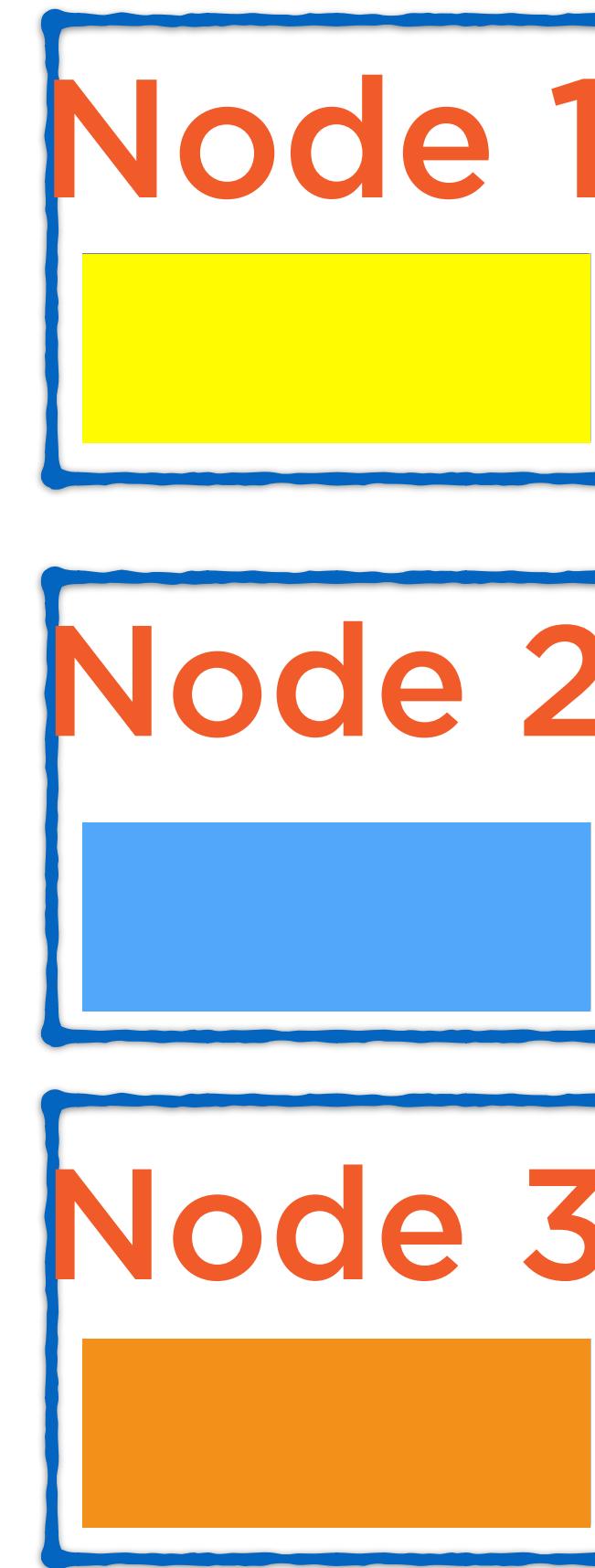
1	Swetha	30	Bangalore
2	Vitthal	35	New Delhi

3	Navdeep	25	Mumbai
4	Janani	35	New Delhi

5	Navdeep	25	Mumbai
6	Janani	35	New Delhi

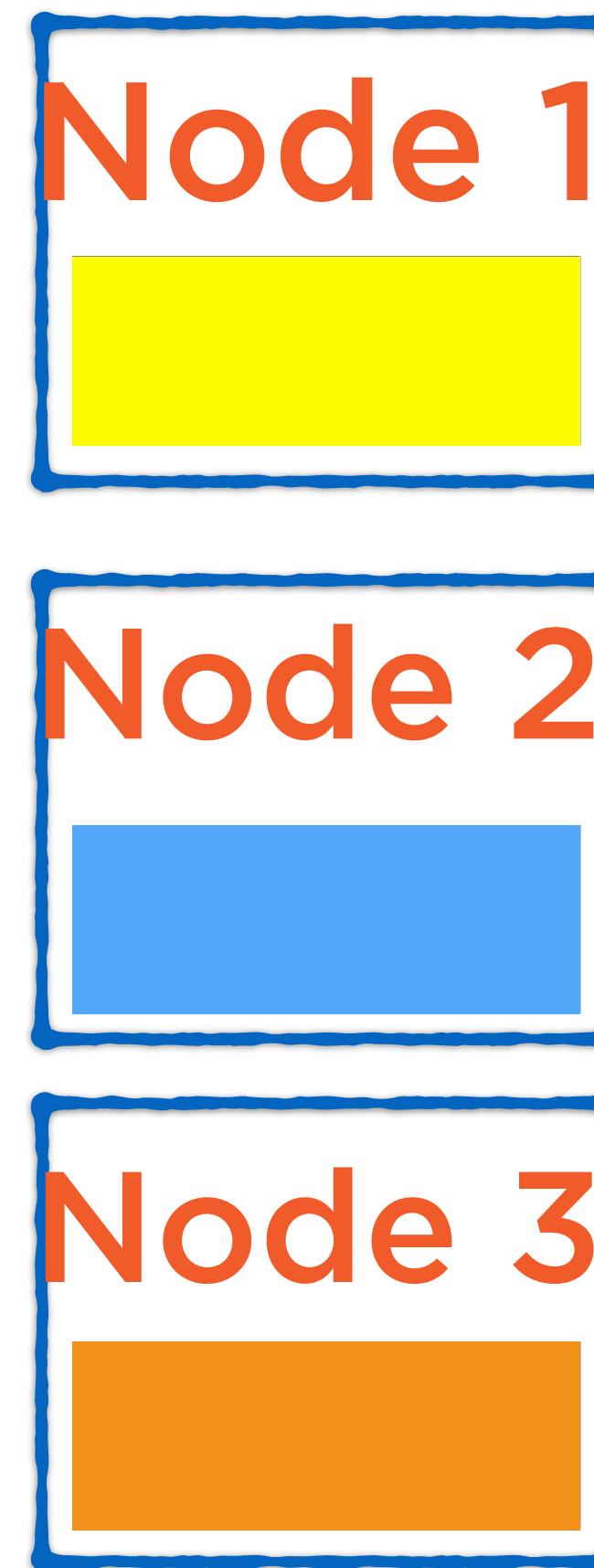
**Distributed to
multiple
machines,
called nodes**

Partitions



**Nodes
process data
in parallel**

Partitions



Resilient
Distributed
Datasets

Partitions
Read-only
Lineage

Read-only

RDDs are immutable

Only Two Types of Operations



Transformation

**Transform into
another RDD**

Action

Request a result

Transformation

1	Swetha	30	Bangalore
2	Vitthal	35	New Delhi
3	Navdeep	25	Mumbai
4	Janani	35	New Delhi
5	Navdeep	25	Mumbai
6	Janani	35	New Delhi

A data set
loaded into
an RDD

Transformation

1	Swetha	30	Bangalore
2	Vitthal	35	New Delhi
3	Navdeep	25	Mumbai
4	Janani	35	New Delhi
5	Navdeep	25	Mumbai
6	Janani	35	New Delhi

The user may define a chain of transformations on the dataset

Transformation

1	Swetha	30	Bangalore
2	Vitthal	35	New Delhi
3	Navdeep	25	Mumbai
4	Janani	35	New Delhi
5	Navdeep	25	Mumbai
6	Janani	35	New Delhi

- 1. Load data**
- 2. Pick only the 3rd column**
- 3. Sort the values**

Transformation

1. Load data
2. Pick only the 3rd column
3. Sort the values



Transformation

**Wait until a result is
requested before
executing any of
these transformations**



Only Two Types of Operations



Transformation

**Transform into
another RDD**

Action

Request a result

Request a result using an action

Action

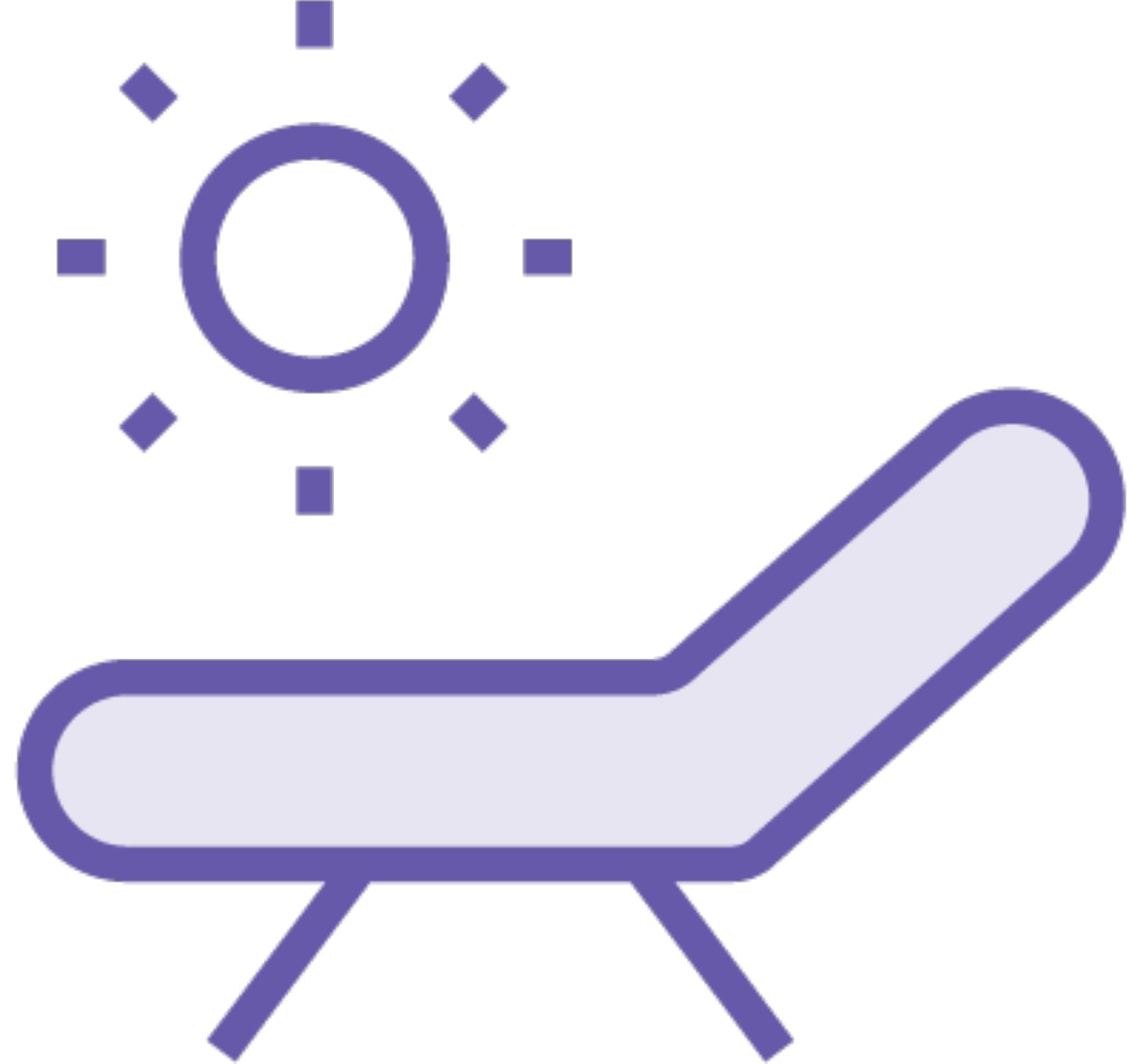
1. The first 10 rows
2. A count
3. A sum

Action

Data is processed only when the user requests a result

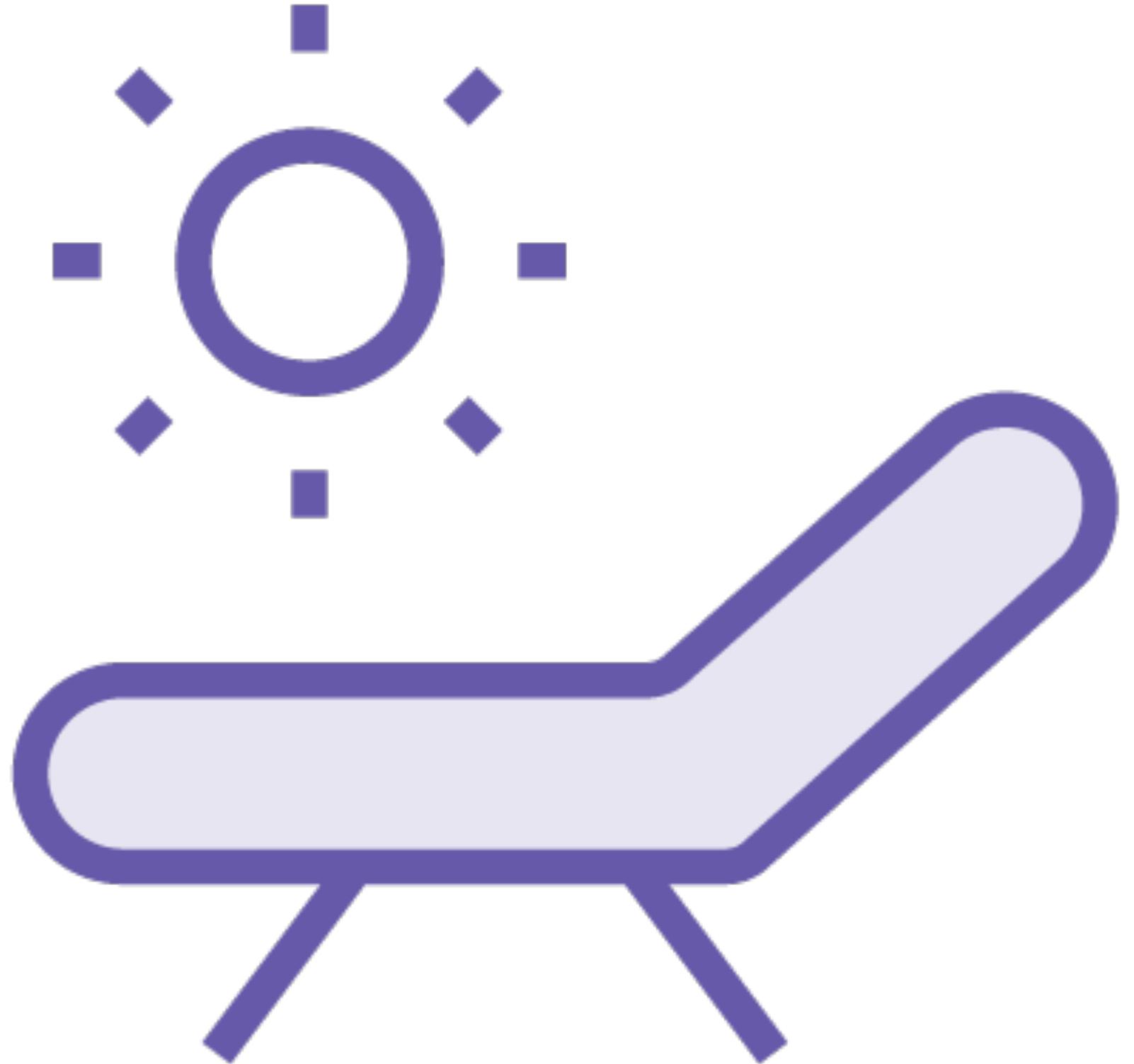
The chain of transformations defined earlier is executed

Lazy Evaluation



**Spark keeps a record
of the series of
transformations
requested by the user**

Lazy Evaluation



**It groups the
transformations in an
efficient way when an
Action is requested**

Resilient
Distributed
Datasets

Partitions
Read-only
Lineage

Lineage

**When created, an RDD
just holds metadata**

- 1. A transformation**
- 2. It's parent RDD**

Lineage

**Every RDD
knows where it
came from**

RDD 2

Transformation 1

RDD 1

Lineage

RDD 2

Lineage can be
traced back all the
way to the source

Transformation 1

RDD 1

Lineage

RDD 2

Lineage can be traced back all the way to the source

Transformation 1

RDD 1

Load data

data.csv

Lineage

When an action is requested on an RDD

All its parent RDDs are materialized

RDD 2

Transformation 1

RDD 1

Load data

data.csv

Lineage

Resilience

In-built fault tolerance

If something goes wrong,
reconstruct from source

Lazy Evaluation

**Materialize only when
necessary**

Introducing Flink

Bounded datasets are
processed in **batches**

Unbounded datasets are
processed as **streams**

Batch vs. Stream Processing

Batch

Bounded, finite datasets

Slow pipeline from data ingestion to analysis

Periodic updates as jobs complete

Order of data received unimportant

Single global state of the world at any point in time

Stream

Unbounded, infinite datasets

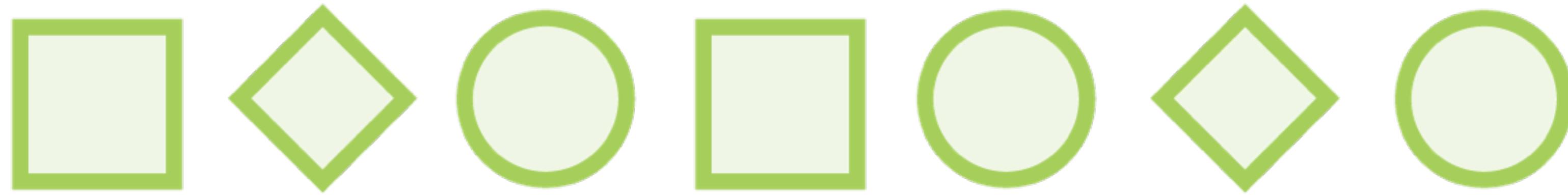
Processing immediate, as data is received

Continuous updates as jobs run constantly

Order important, out of order arrival tracked

No global state, only history of events received

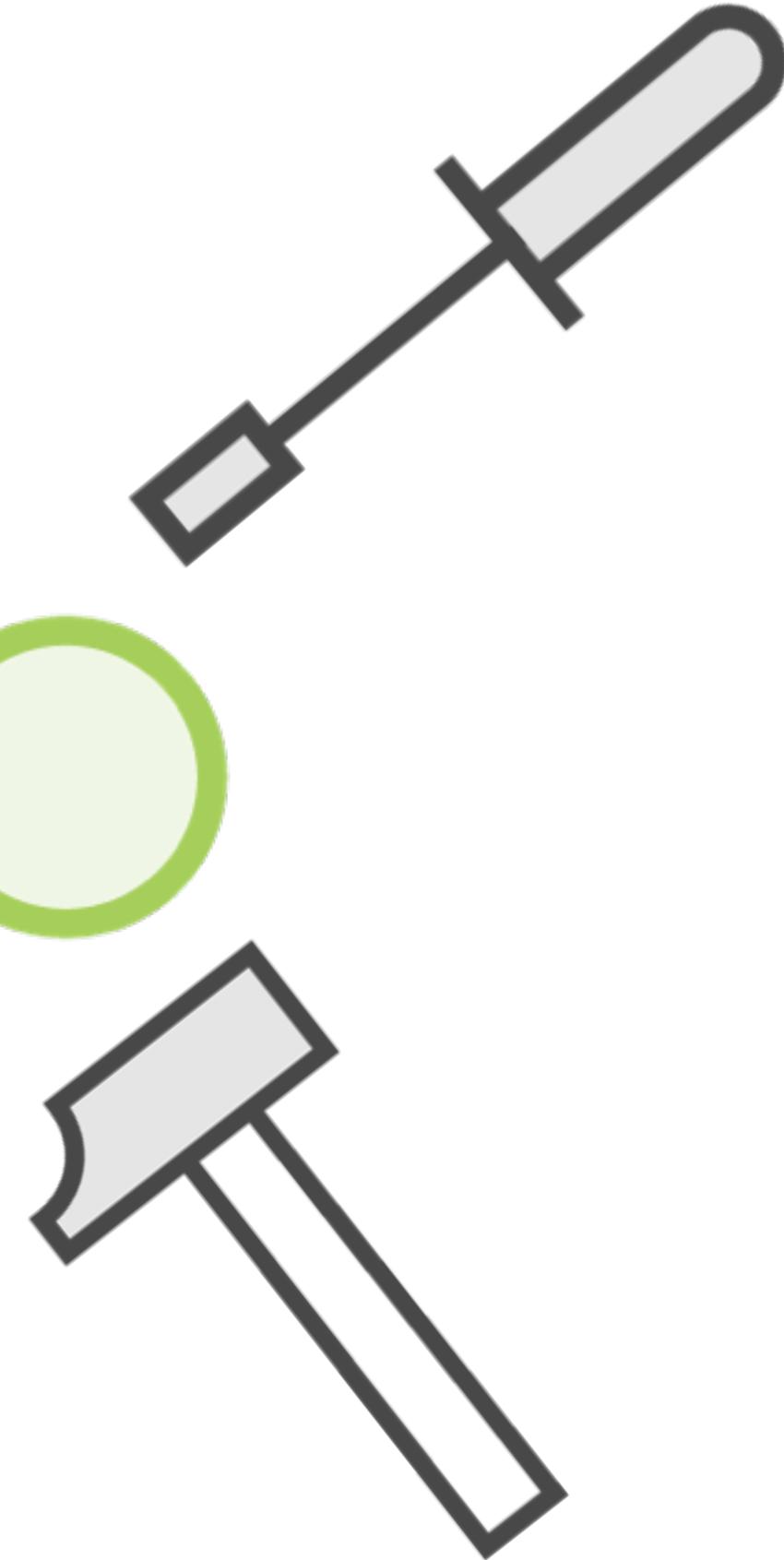
Stream Processing



Data is received as a stream

- Log messages
- Tweets
- Climate sensor data

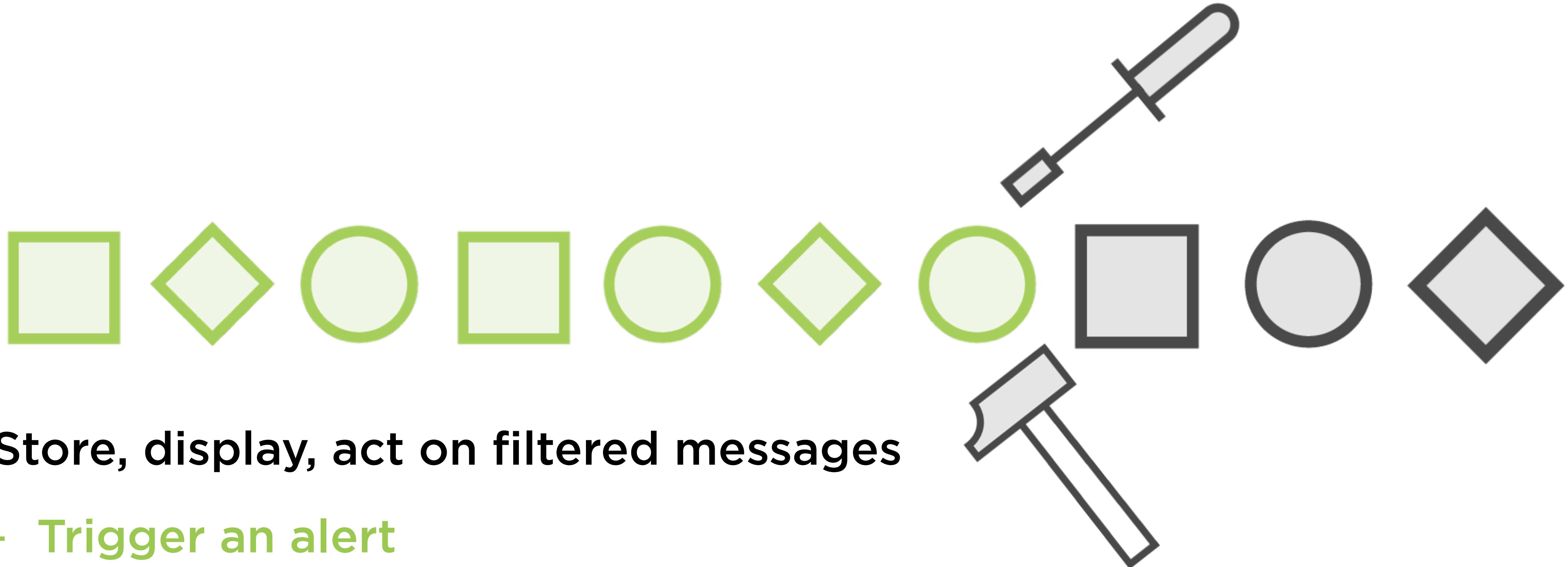
Stream Processing



Process the data one entity at a time

- Filter error messages
- Find references to the latest movies
- Track weather patterns

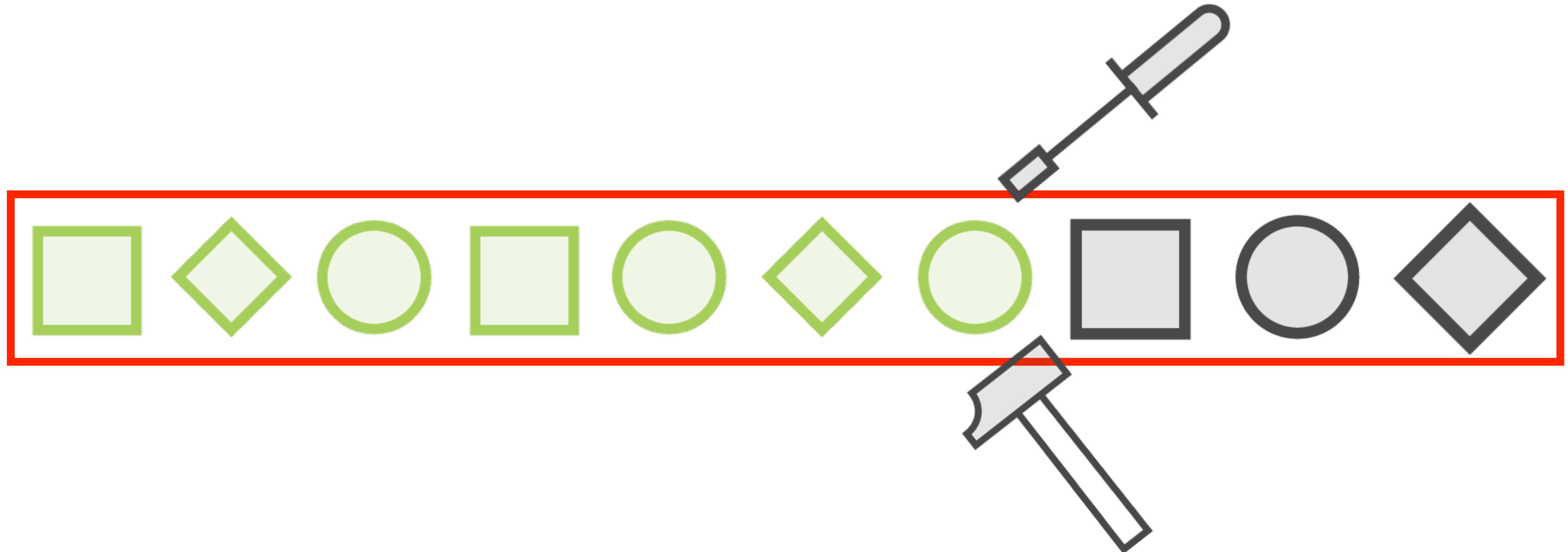
Stream Processing



Store, display, act on filtered messages

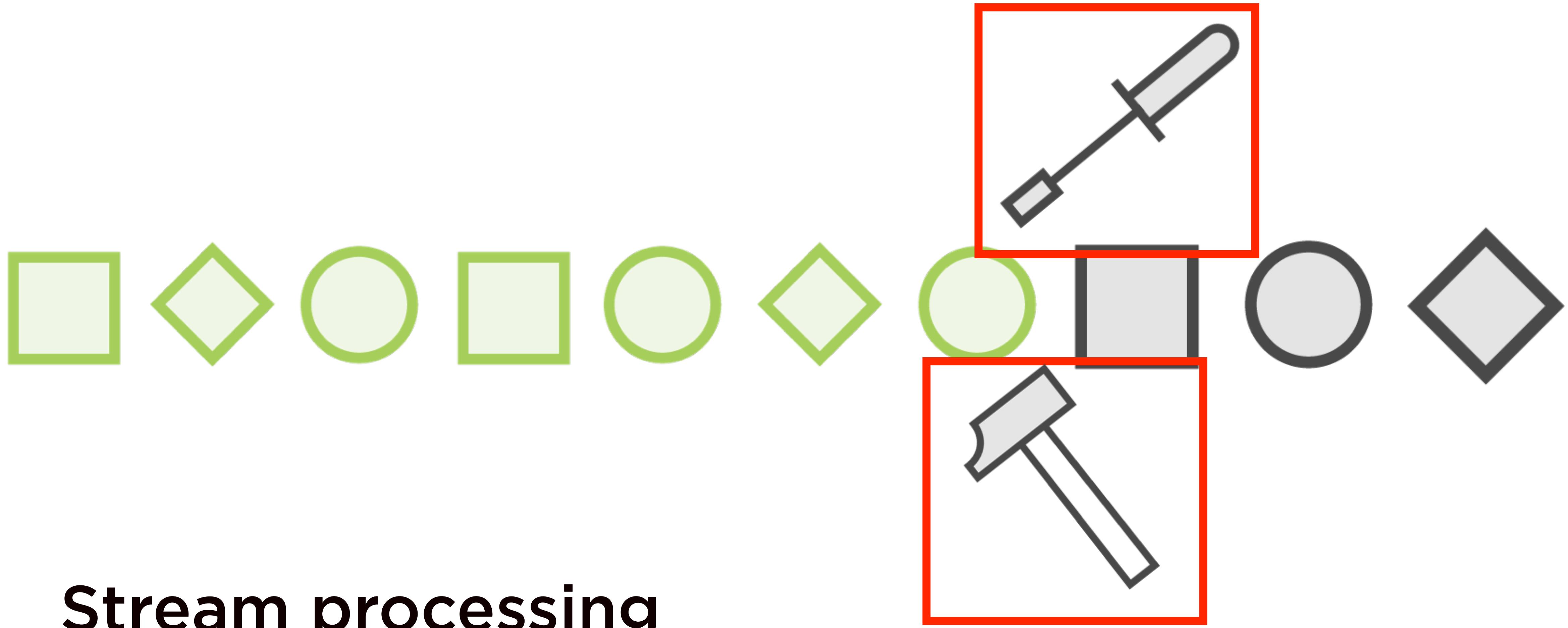
- Trigger an alert
- Show trending graphs
- Warn of sudden squalls

Stream Processing



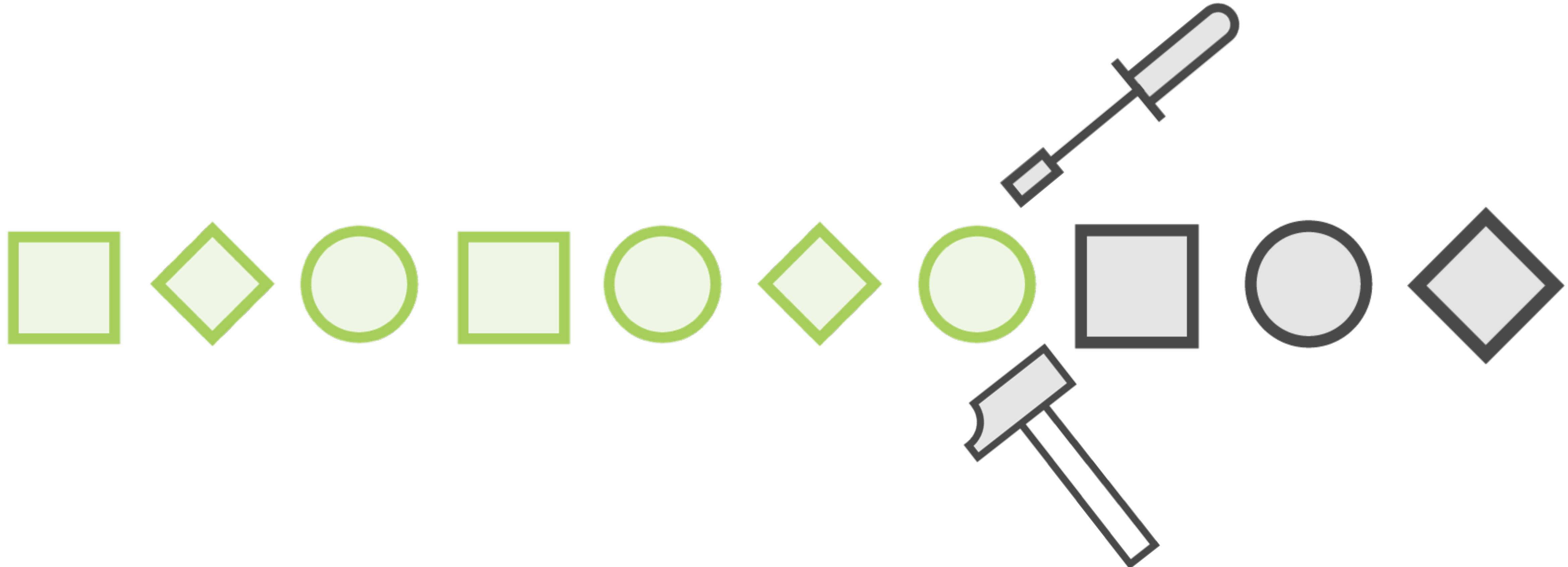
Streaming data

Stream Processing



Stream processing

Stream Processing



Traditional Systems



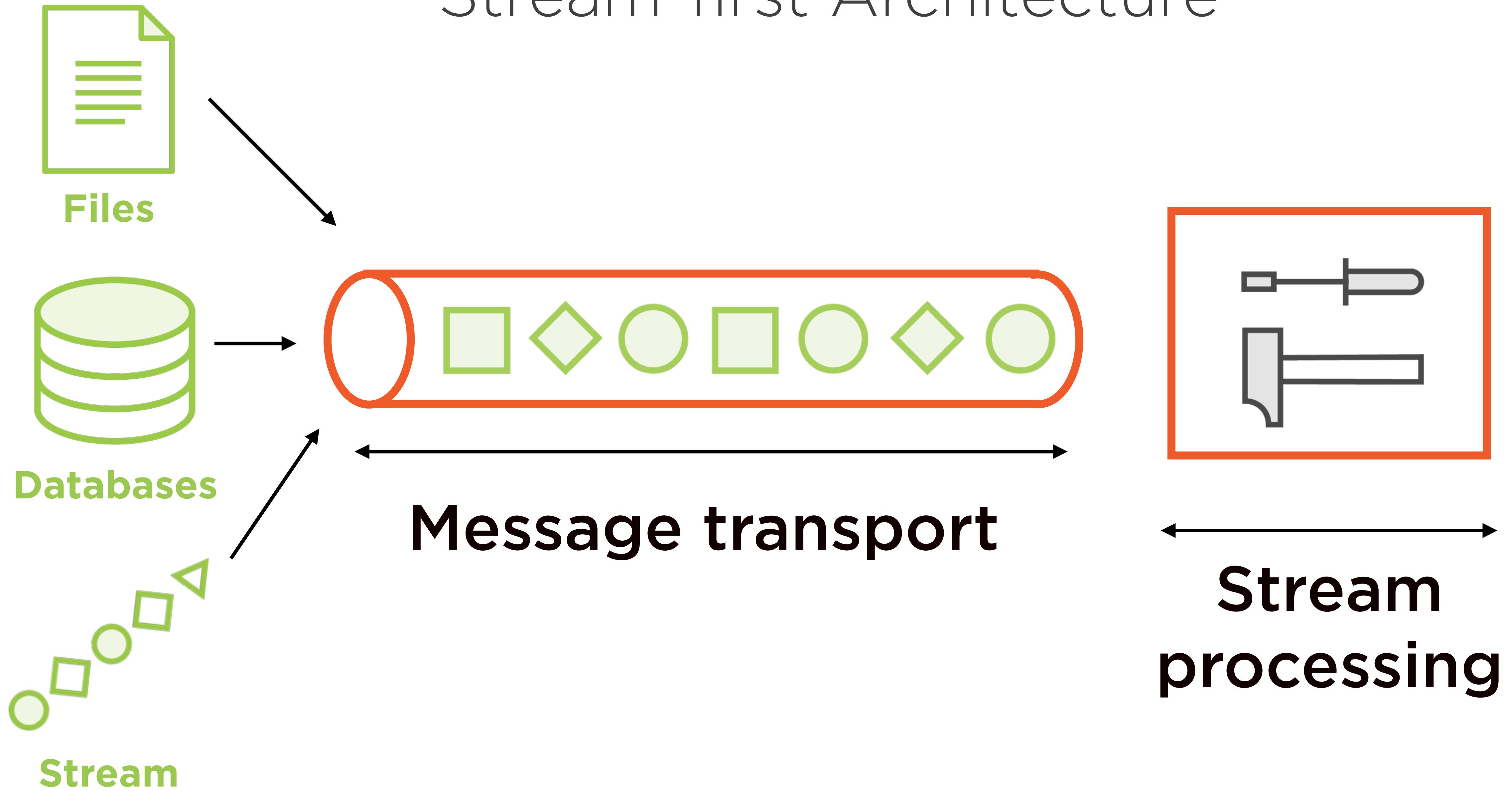
Files



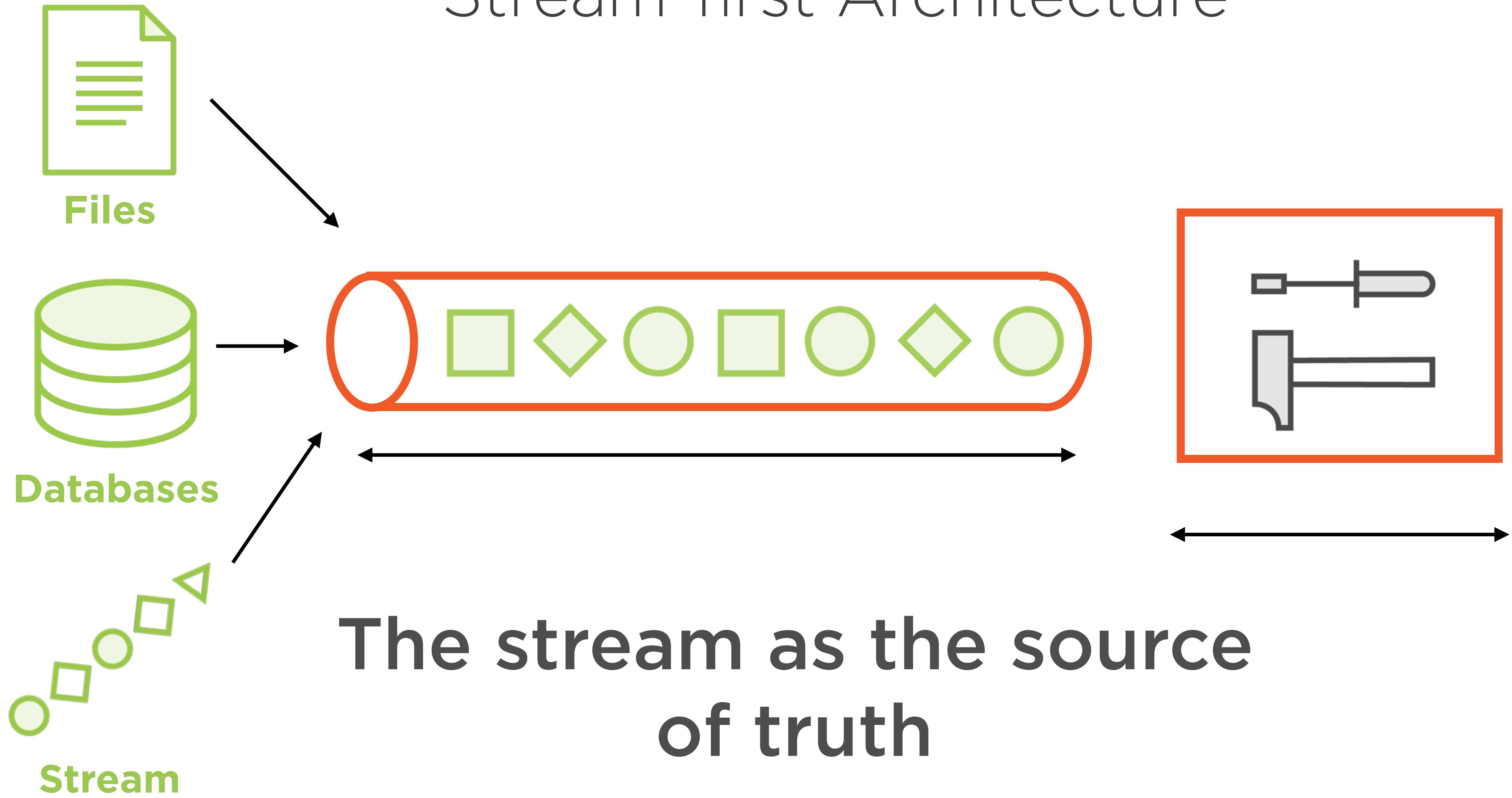
Databases

**Reliable storage as the
source of truth**

Stream-first Architecture

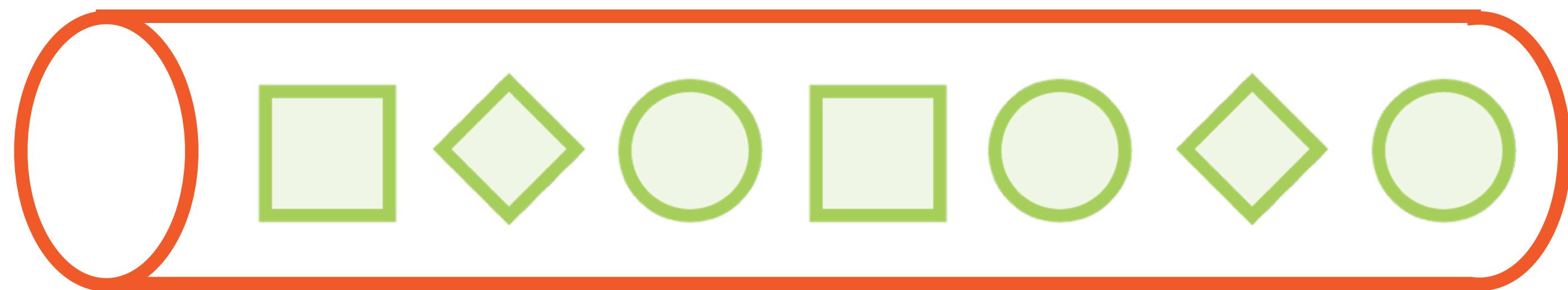


Stream-first Architecture

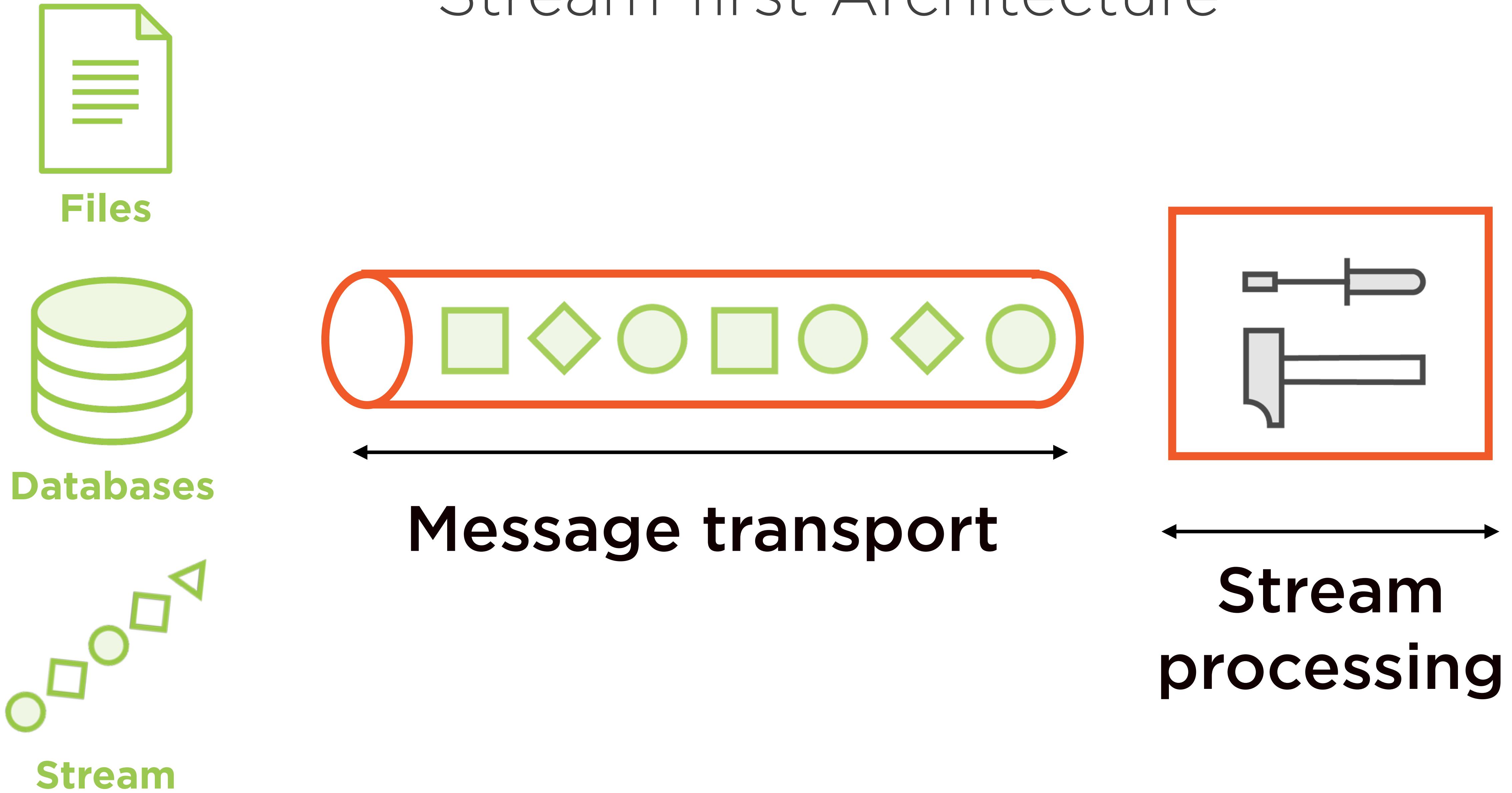


Message Transport

Buffer for event data
Performant and persistent
Decoupling multiple sources from processing
Kafka, MapR streams



Stream-first Architecture



High throughput, low latency

Fault tolerance with low overhead

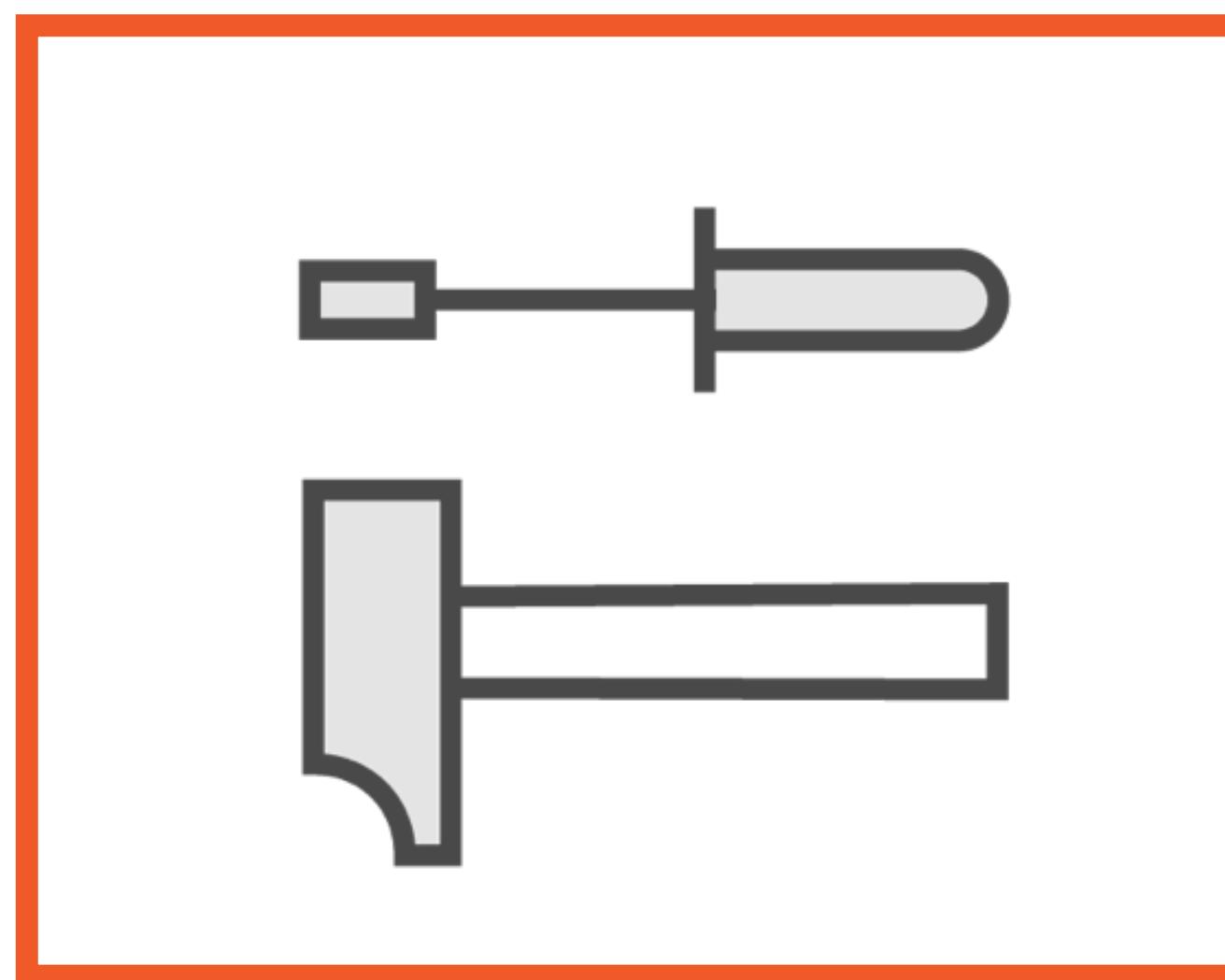
Manage out of order events

Easy to use, maintainable

Replay streams

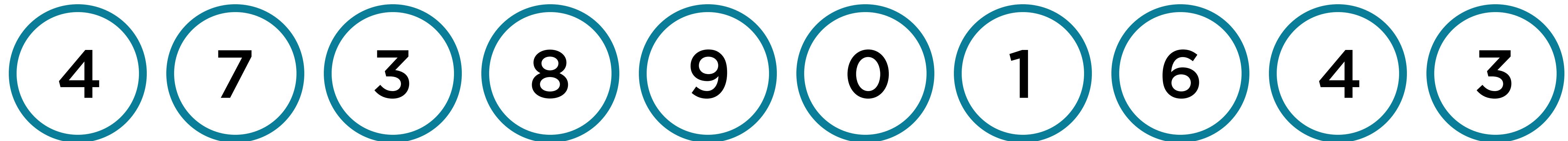
Streaming Spark, Storm, Flink

Stream Processing



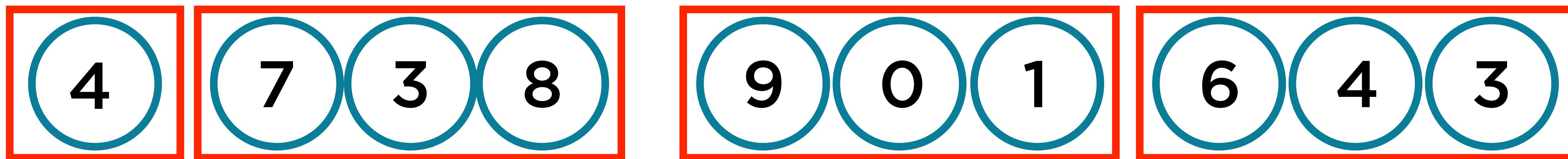
A good approximation to
stream processing is the use of
micro-batches

Mimicking Streams Using Micro-batches



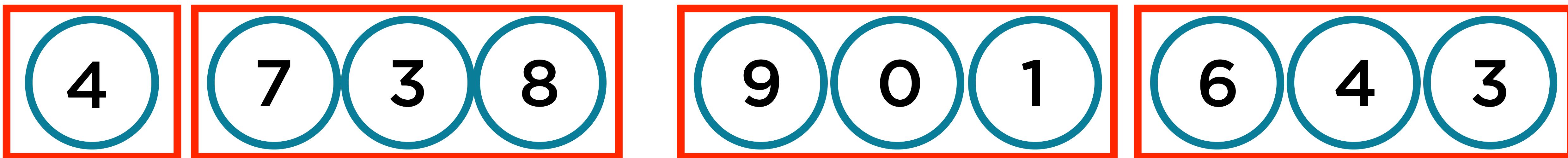
A stream of integers

Mimicking Streams Using Micro-batches



Grouped into batches

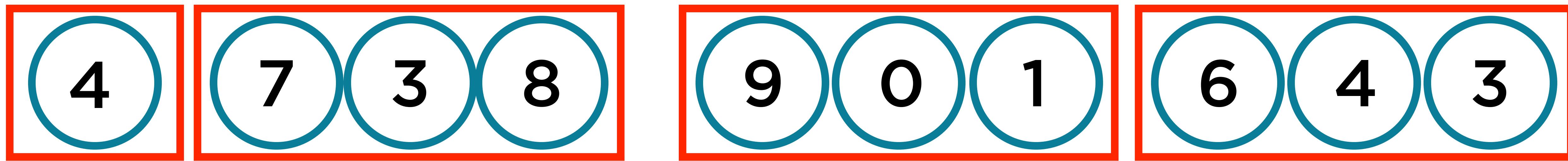
Mimicking Streams Using Micro-batches



If the batches are **small** enough...

It **approximates** real-time
stream processing

Mimicking Streams Using Micro-batches



Exactly once semantics, replay micro-batches

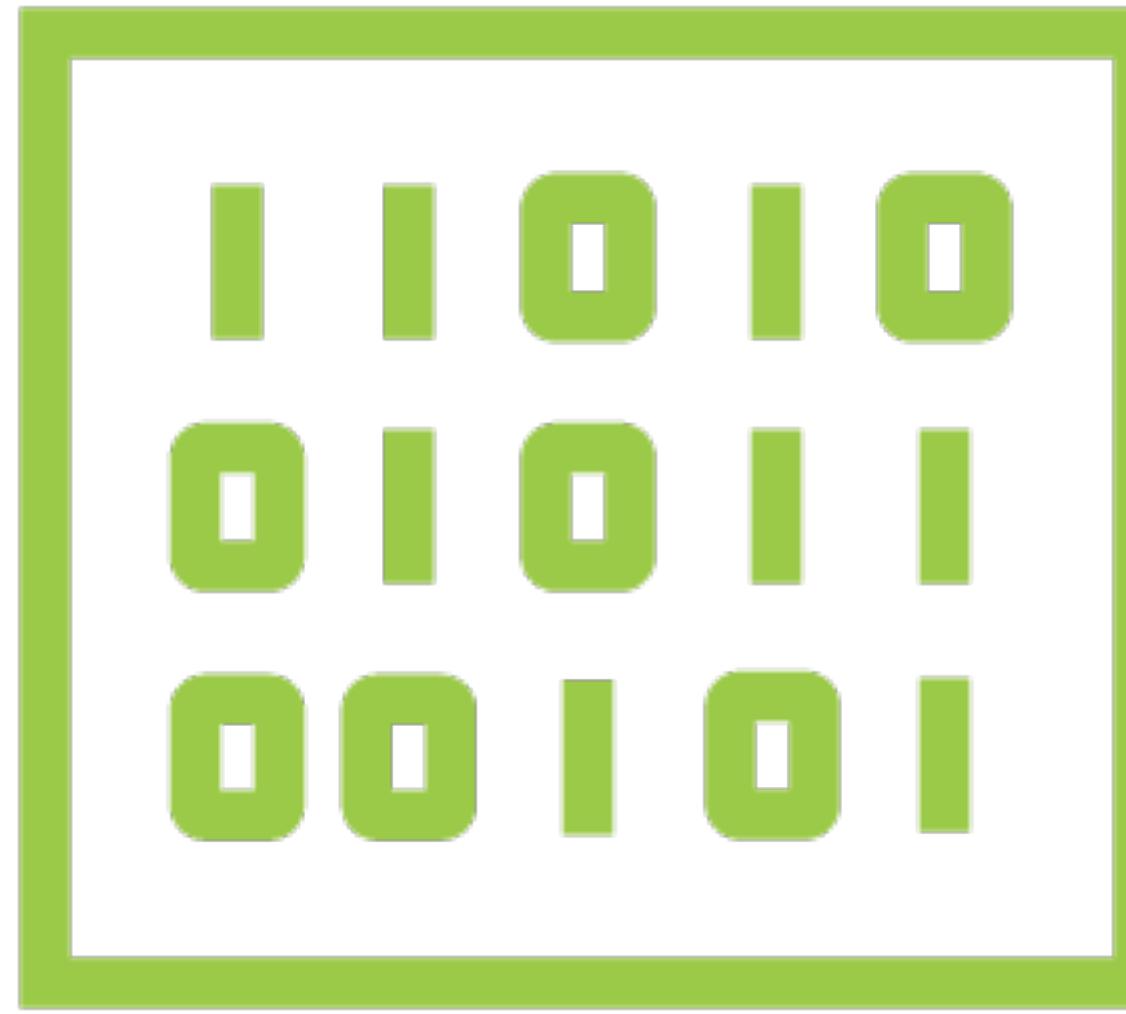
Latency-throughput trade-off based on batch sizes

Spark Streaming, Storm Trident

Apache Flink is an open source,
distributed system built using
the **stream-first architecture**

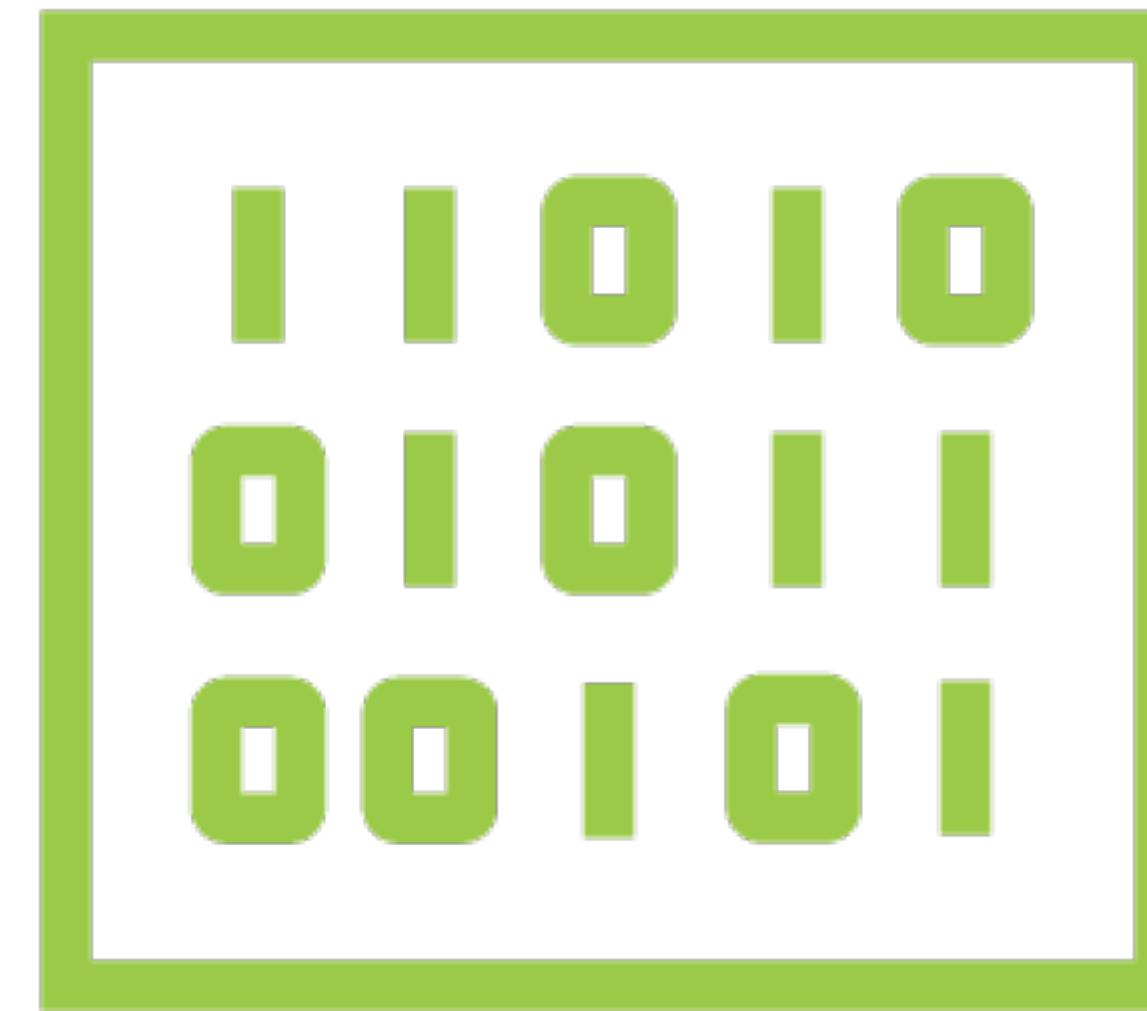
The stream is the source of truth

Apache Flink



Streaming execution model
Processing is continuous, one
event at a time

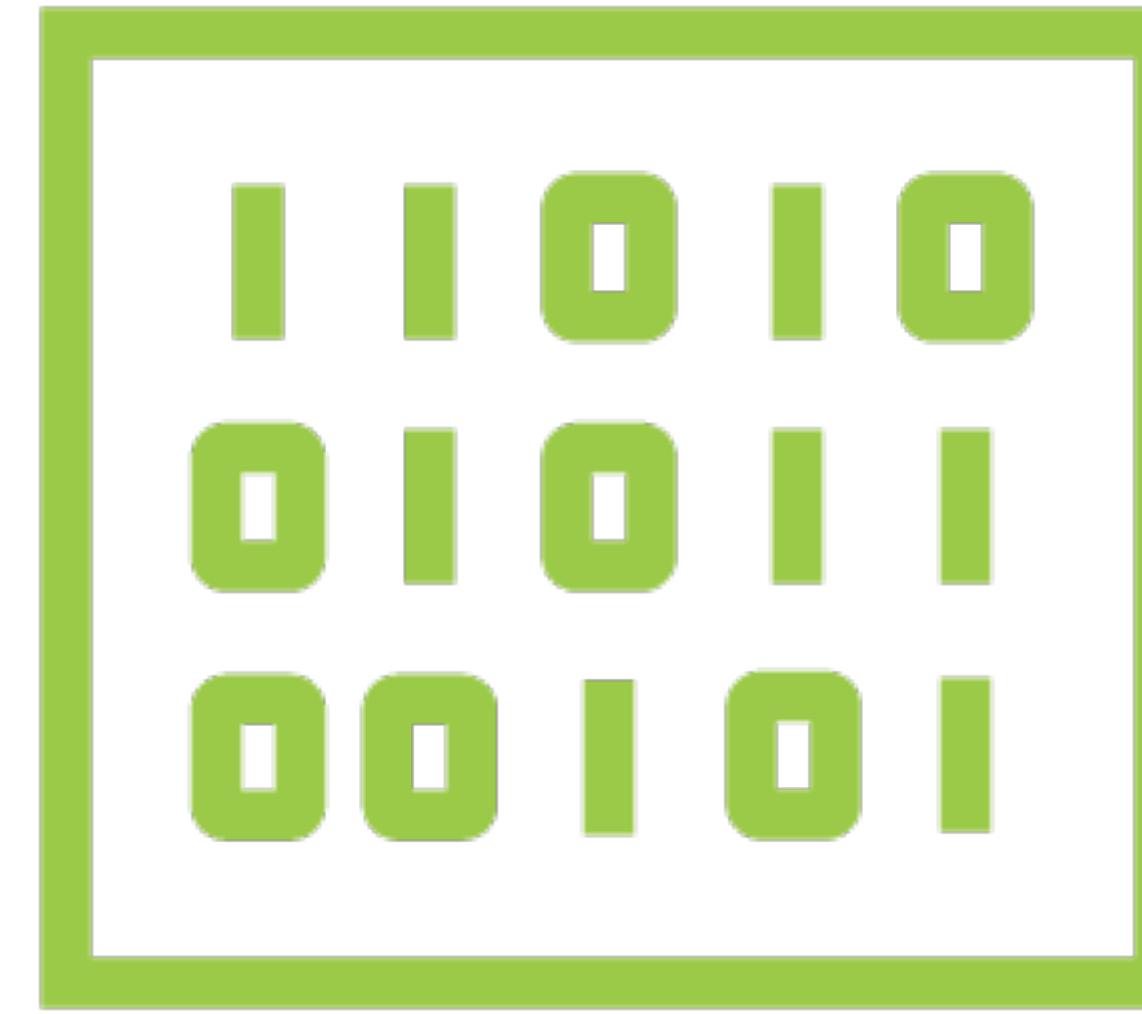
Apache Flink



Everything is a stream

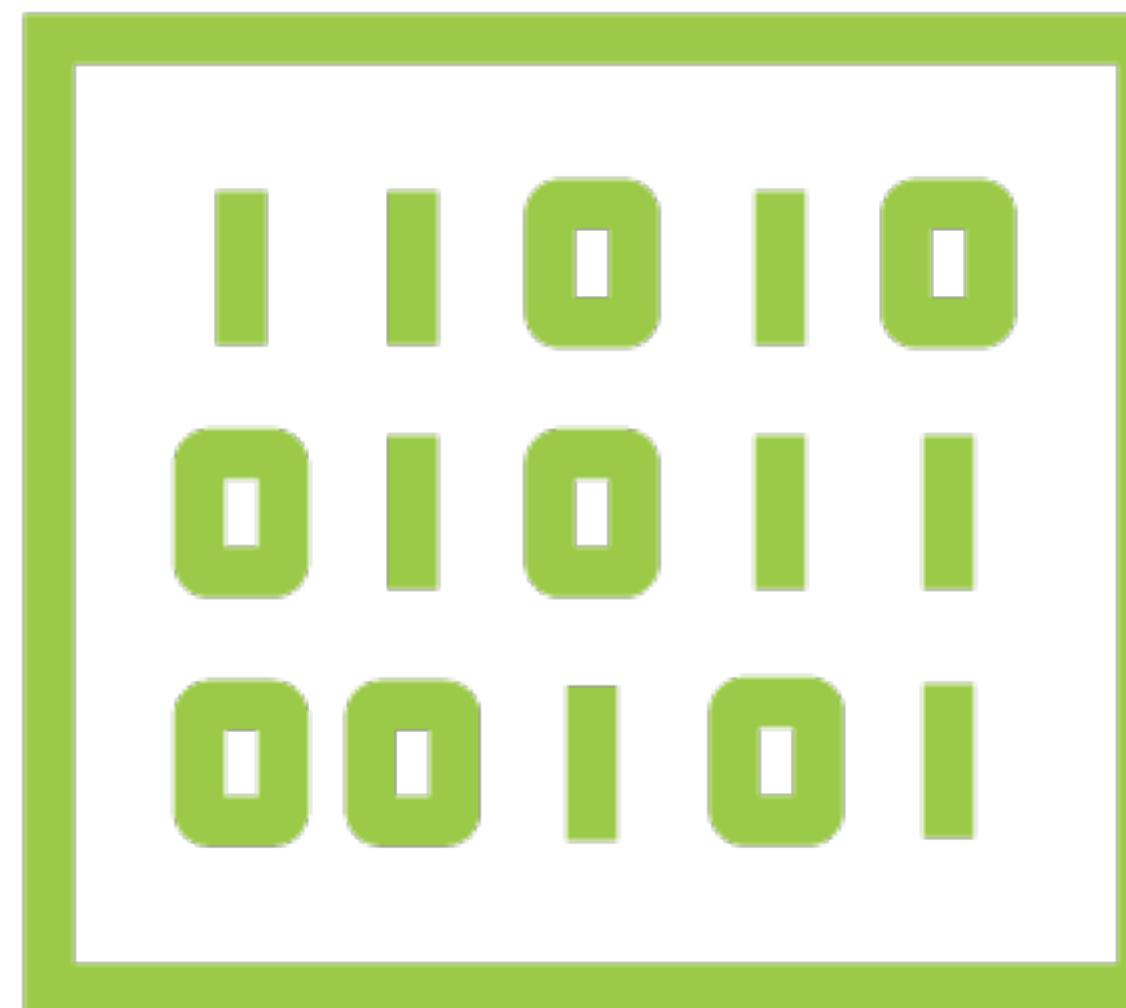
Batch processing with bounded
datasets are a **special** case of
the unbounded dataset

Apache Flink



Same engine for all
Streaming and batch APIs both use
the same underlying architecture

Stream Processing with Flink



Handles **out of order** or late arriving data

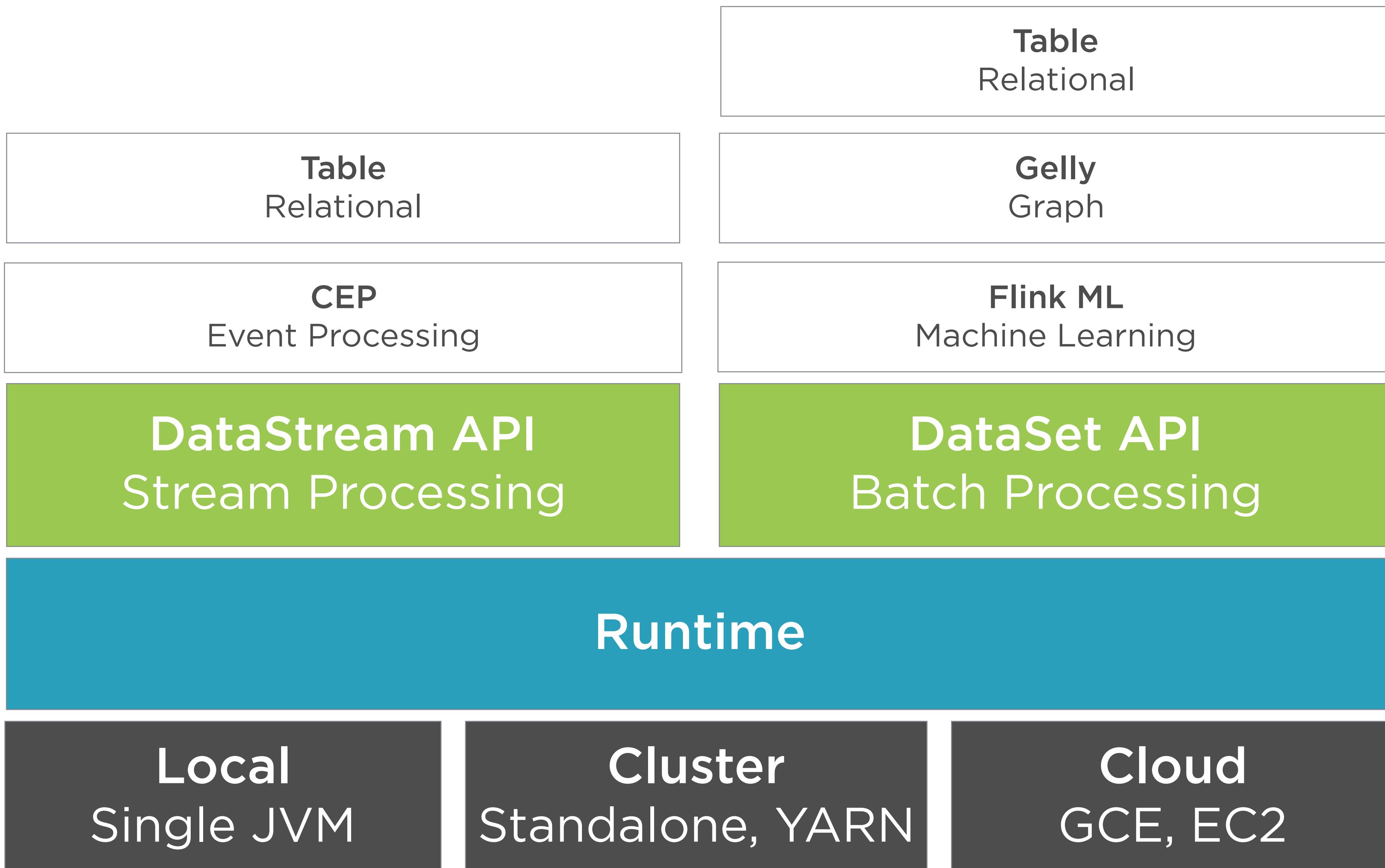
Exactly once processing for stateful computations

Flexible windowing based on time, sessions, count, etc.

Lightweight fault tolerance and checkpointing

Distributed, runs in large scale clusters

Apache Flink



Apache Flink

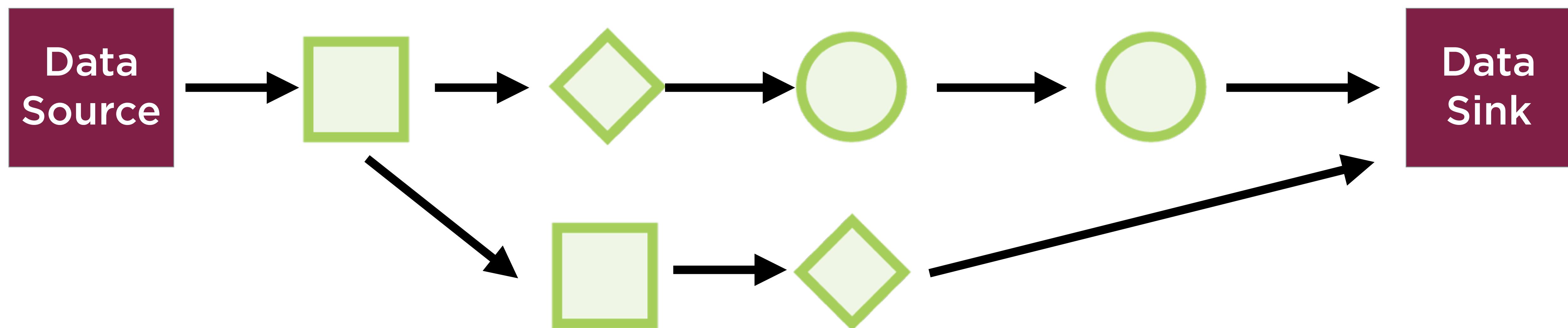
DataStream API
Stream Processing

Flink Programming Model



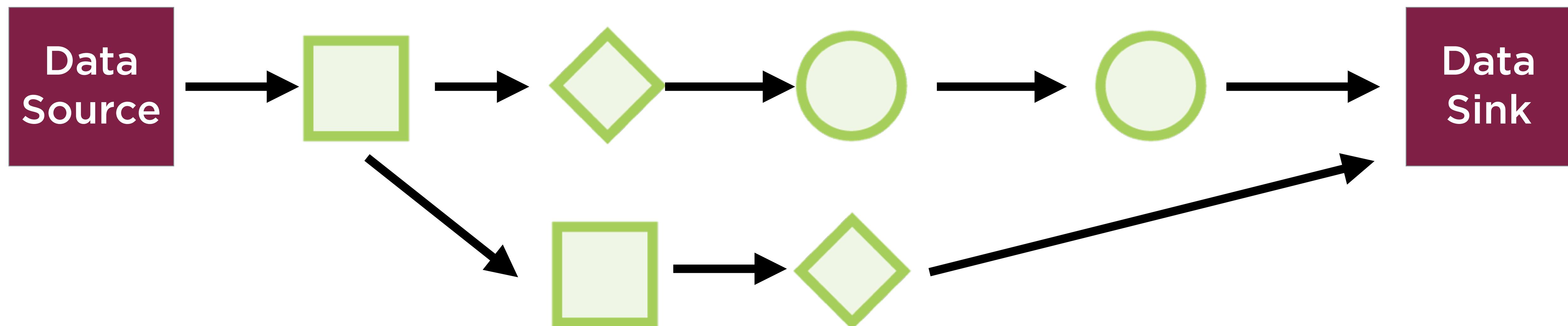
Flink Programming Model

Transformations



Transformations

A directed-acyclic graph



Mapping to Hadoop ecosystem

BigQuery

BigQuery

- Enterprise Data Warehouse
- SQL queries - with Google storage underneath
- Fully-managed - no server, no resources deployed
- Access through: Web UI, REST API, clients
- Third party tools

Data Model

- Dataset = set of tables and views
- Table must belong to dataset
- Dataset must belong to a project
- Tables contain records with rows and columns (fields)
- Nested and repeated fields are OK too

Table Schema

- Can be specified at creation time
- Can also specify schema during initial load
- Can update schema later too

Table Types

- Native tables: BigQuery storage
- External tables
 - BigTable
 - Cloud Storage
 - Google Drive
- Views

Schema Auto-Detection

- Available while
 - Loading data
 - Querying external data
- BigQuery selects a random file in the data source and scans up to 100 rows of data to use as a representative sample
- Then examines each field and attempts to assign a data type to that field based on the values in the sample

Loading Data

- Batch loads
 - CSV
 - JSON (newline delimited)
 - Avro
 - GCP Datastore backups
- Streaming loads
 - High volume event tracking logs
 - Realtime dashboards

Other Google Sources

- Cloud storage
- Analytics 360
- Datastore
- Dataflow
- Cloud storage logs

Data Formats

- CSV
- JSON (newline delimited)
- Avro (open source data format that bundles serialized data with the data's schema in the same file)
- Cloud Datastore backups (BigQuery converts data from each entity in Cloud Datastore backup files to BigQuery's data types)

Alternatives to Loading

- Public datasets
- Shared datasets
- Stackdriver log files (needs export - but direct)

Querying and Viewing

- Interactive queries
- Batch queries
- Views
- Partitioned tables

Interactive Queries

- Default mode (executed as soon as possible)
- Count towards limits on
 - Daily usage
 - Concurrent usage

Batch Queries

- BigQuery will schedule these to run whenever possible (idle resources)
- Don't count towards limit on concurrent usage
- If not started within 24 hours, BigQuery makes them interactive

Views

- BigQuery views are logical - not materialised
- Underlying query will execute each time view is accessed
- Billing will happen accordingly
- Can't assign access control - based on user running view
- Can create **authorised view**: share query results with groups without giving read access to underlying data
- Can give **row-level permissions** to different users within same view

Views

- Can't export data from a view
- Can't use JSON API to retrieve data
- Can't mix standard and legacy SQL
 - eg standard SQL query can't access legacy-SQL view
- No user-defined functions allowed
- No wildcard table references allowed
- Limit of 1000 authorised views per dataset

Partitioned Tables

- Special table where data is partitioned for you
- No need to create partitions manually or programmatically
- Manual partitions - performance degrades
- Limit of 1000 tables per query does **not** apply

Partitioned Tables

- Date-partitioned tables offered by BigQuery
- Need to declare table as partitioned at creation time
- No need to specify schema (can do while loading data)
- BigQuery automatically creates date partitions

Query Plan Explanation

- In the web UI, click on “Explanation”

Slots

- Unit of Computational capacity needed to run queries
- BigQuery calculates on basis of query size, complexity
- Usually default slots sufficient
- Might need to be expanded for very large, complex queries
- Slots are subject to quota policies
- Can use StackDriver Monitoring to track slot usage