Week 1: Hadoop / HDFS / MapReduce

Plan for the week

- What is Big Data
- Hadoop ecosystem to work with it
  - HDFS for storage
  - MapReduce for computation

Big Data & Hadoop

Big Data

Collected data helps companies improve their services for the user: **personally** recommend movies, music, products, other users, etc.

Big Data

Big Data has 3 important properties (**3V**):

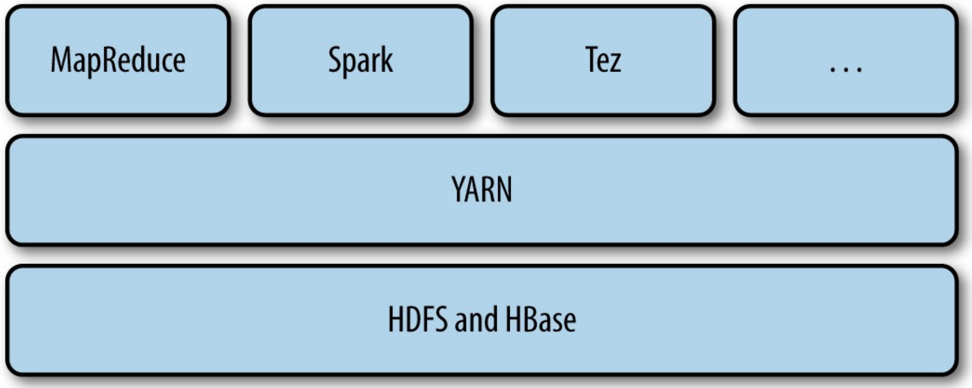
Volume – the amount of data is growing all the time.
Some organizations process 10 TB, while others process 100 PB.

Velocity – the speed of data generation is also growing.
Today billions of posts are generated daily on Facebook, Twitter, etc.

Variety – the diversity of data formats is high.
Today we analyze texts, images, audio recordings, videos, etc.

https://www.oracle.com/ru/big-data/what-is-big-data.html

Hadoop ecosystem for Big Data



HDFS – distributed file system

YARN – cluster resource manager (CPU, RAM, etc)

MapReduce – API for distributed computing

http://grut-computing.com/HadoopBook.pdf

Scalability

When data is small and structured, it can be stored in a relational database (RDBMS).

To speed up processing in RDBMS you need a more powerful server (**vertical scalability**).

In Hadoop you can use a lot of commodity servers to speed up processing (**horizontal scalability**), which is much cheaper.

RDBMS vs Hadoop

	RDBMS	**Hadoop**
Hardware	Powerful servers	Commodity hardware
Data volume	Small	Big
Response time	Instant response	Delayed response
Data format	Tables (structured)	Files (unstructured)

HDFS

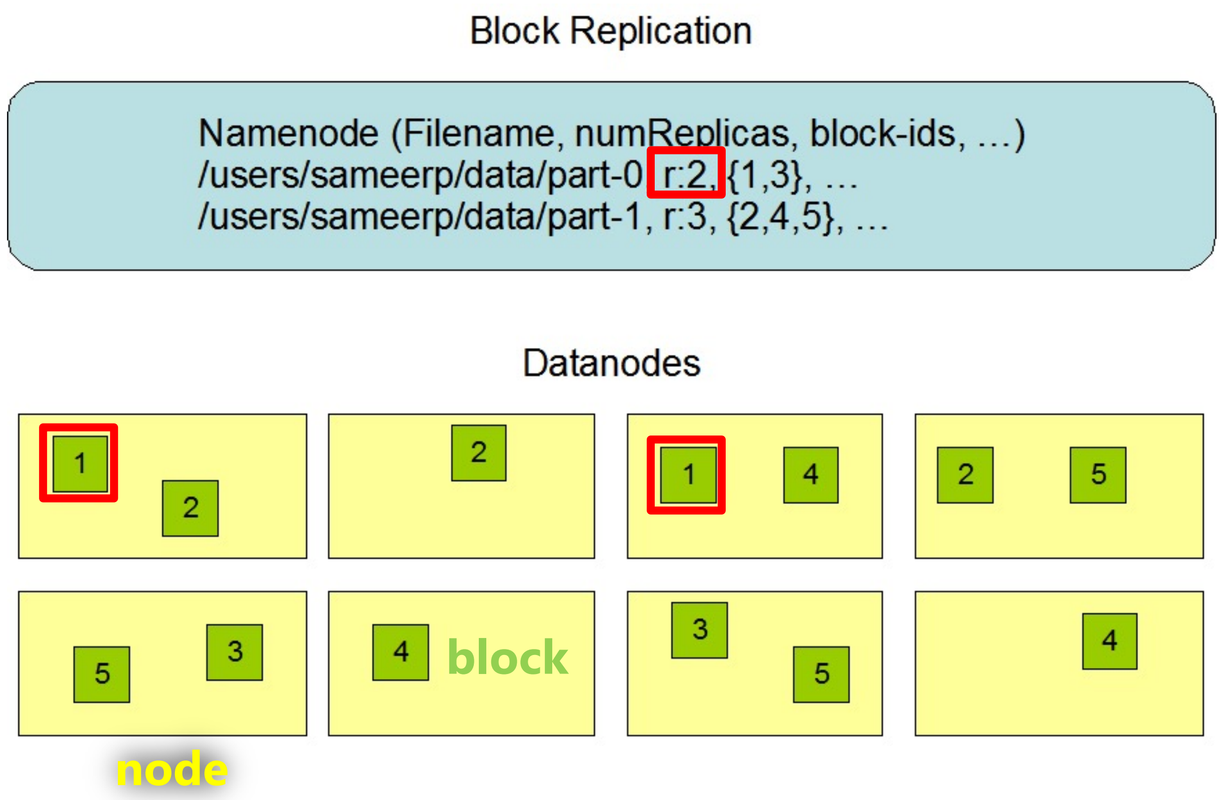
Hadoop Distributed File System (HDFS)

Files are split into **blocks**, which are stored on different **Data Nodes**

Every block is **replicated** on many **Data Nodes** (w.r.t replication factor, e.g. 3)

Mapping **filename → blocks** is stored in memory of a **Name Node**

HDFS



https://hadoop.apache.org/docs/current/hadoop-project-dist/hadoop-hdfs/HdfsDesign.html

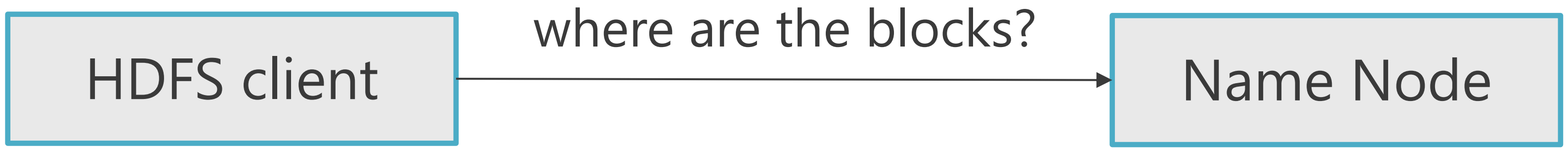
Why you need to replicate blocks

- Let's say a node fails with probability 0.001
- What is the probability that at least 1 of 500 nodes fails?

Why you need to replicate blocks

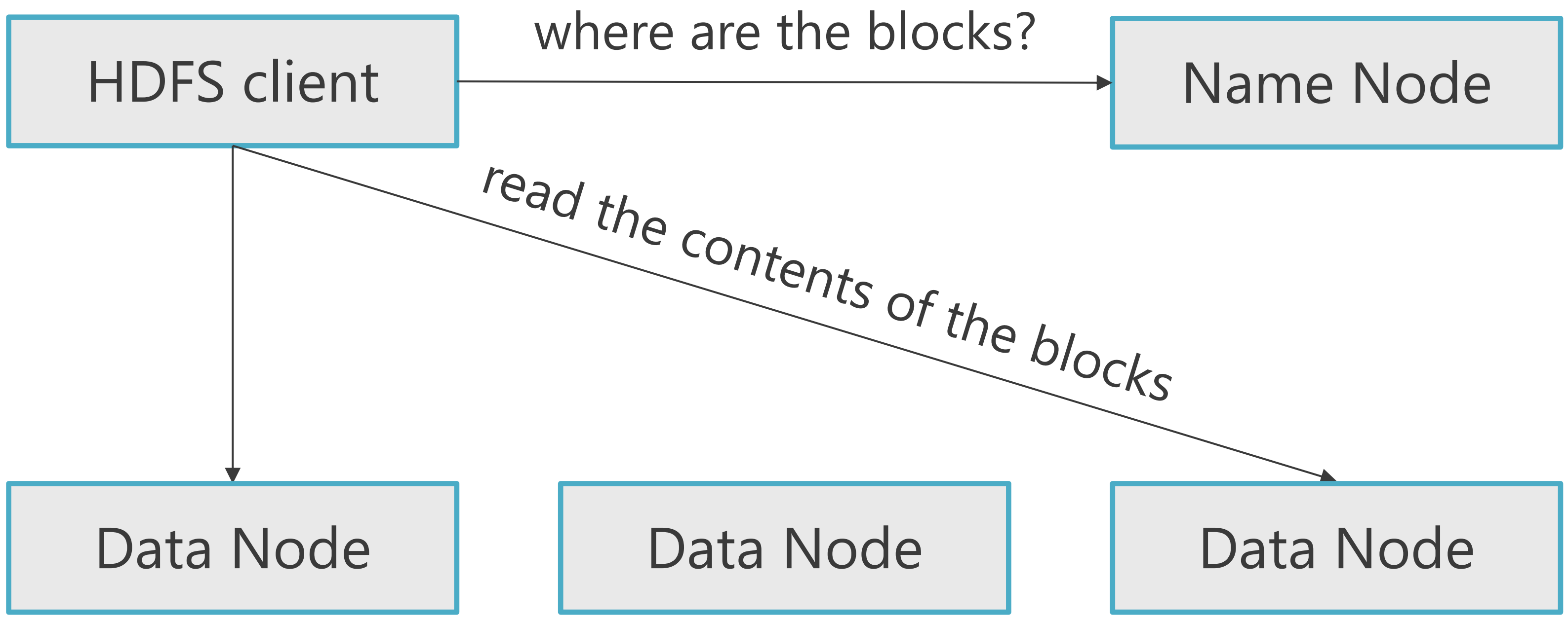
- Let's say a node fails with probability 0.001
- What is the probability that at least 1 of 500 nodes fails?
- $1 - (1 - 0.001)^{500} \approx 0.4$

Reading from HDFS



http://grut-computing.com/HadoopBook.pdf

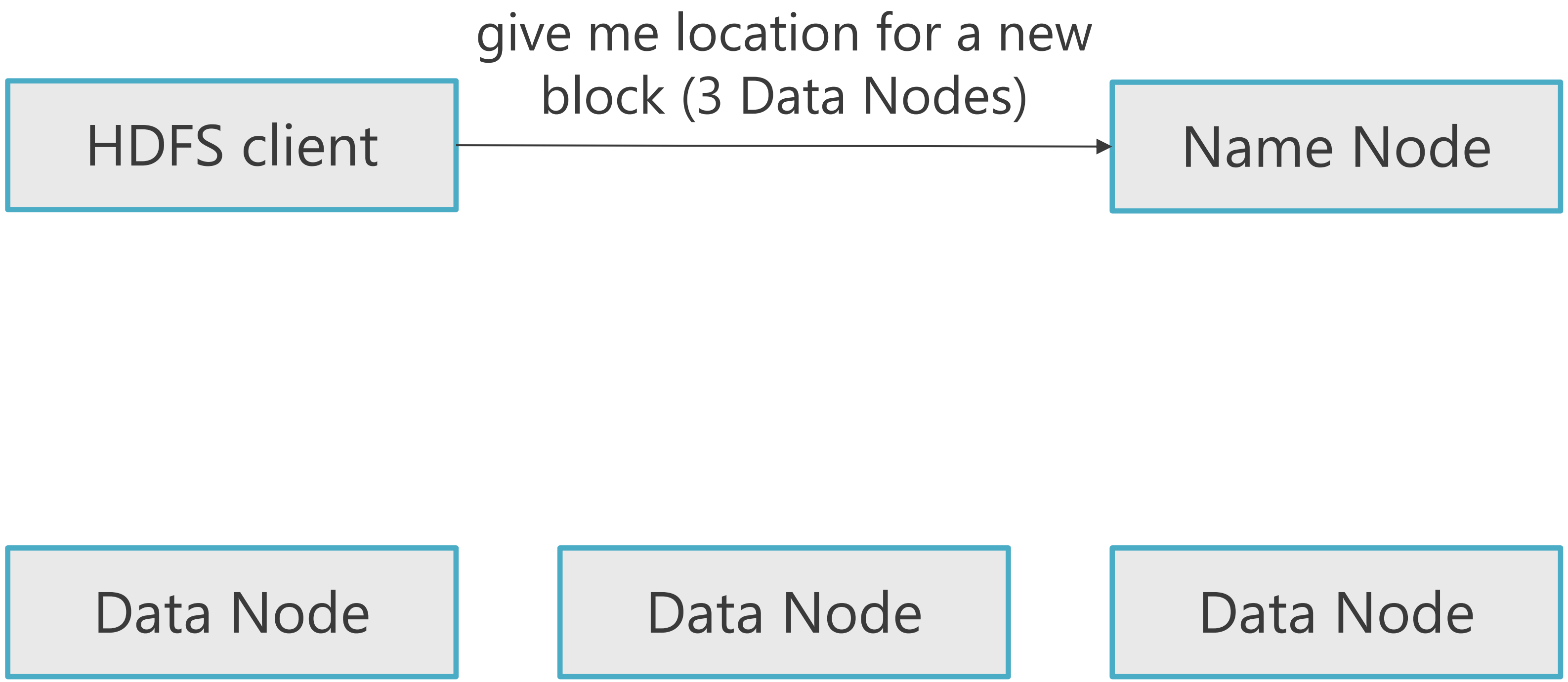
Reading from HDFS



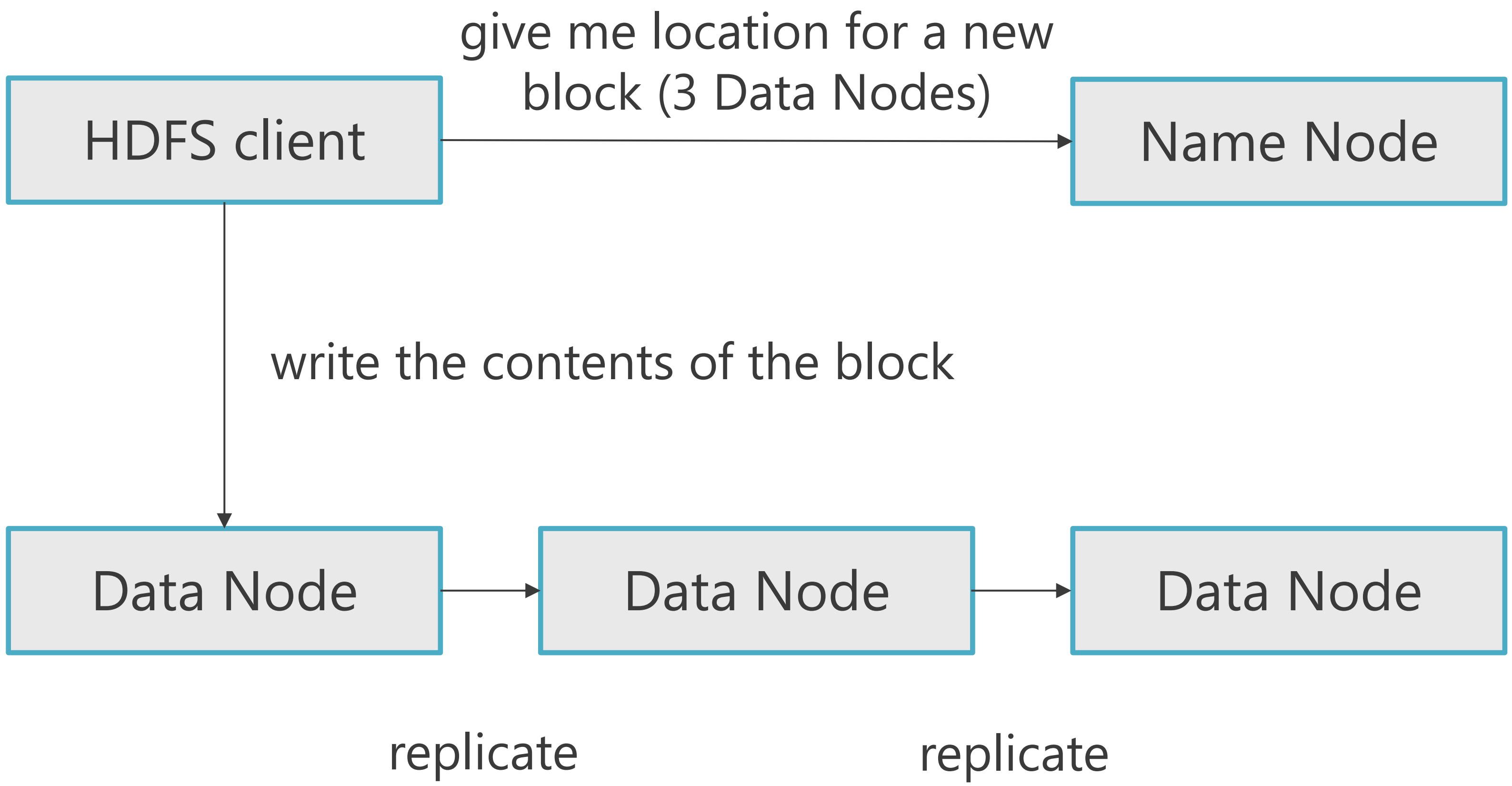
Writing into HDFS

http://grut-computing.com/HadoopBook.pdf

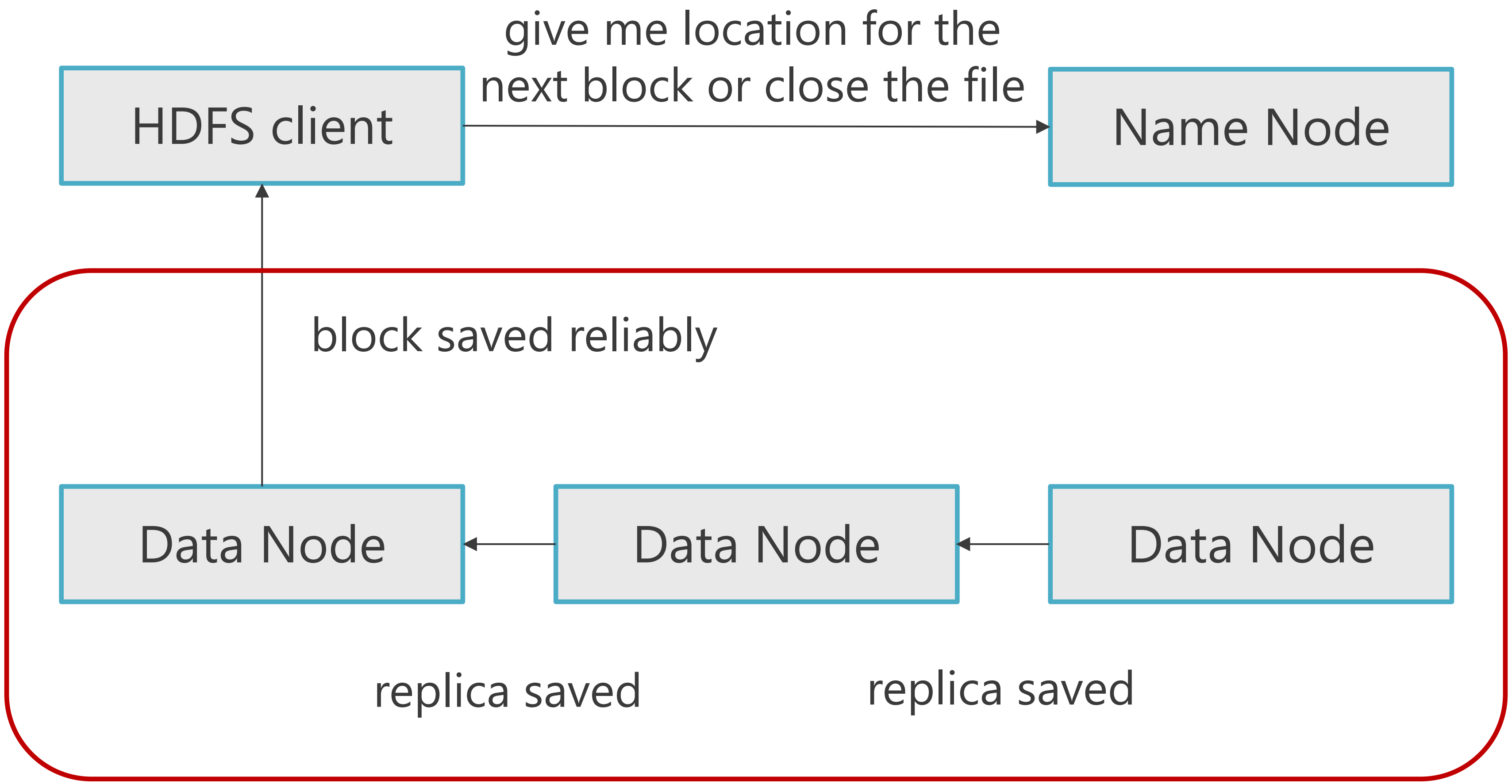
Writing into HDFS



Writing into HDFS



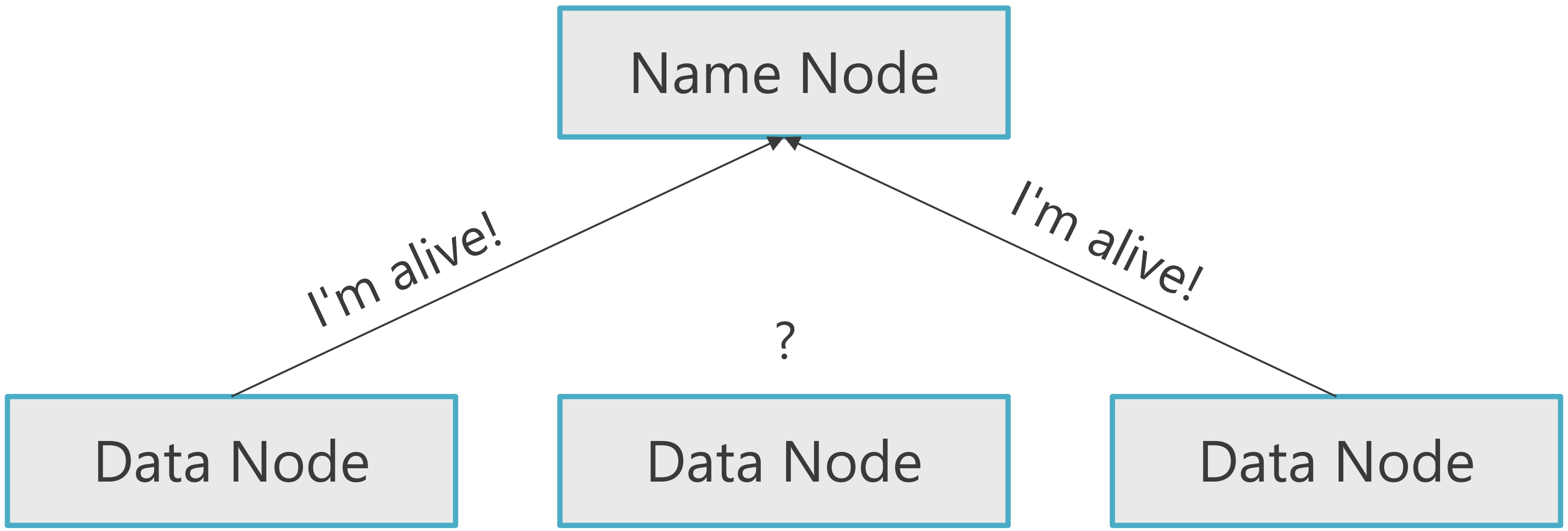
Writing into HDFS



client can wait for “block saved reliably” asynchronously,
client can immediately start writing the next block

HDFS resiliency

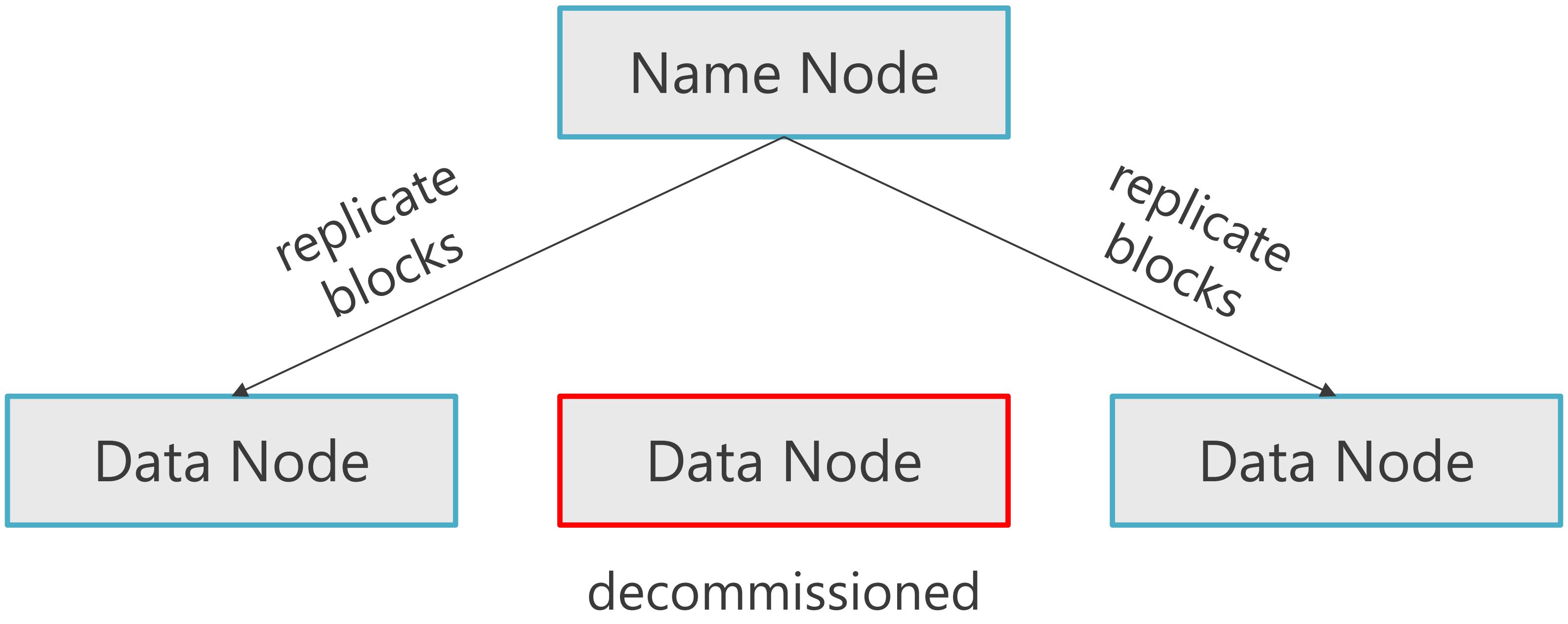
Data Nodes report to **Name Node** that they're still alive (heartbeat). If heartbeat wasn't received, **Name Node** may decide to decommission the node (pull it out of the cluster) replicating its blocks to another available node.



https://www.ibm.com/developerworks/library/wa-introhdfs/index.html

HDFS resiliency

Data Nodes report to **Name Node** that they're still alive (heartbeat). If heartbeat wasn’t received, **Name Node** may decide to decommission the node (pull it out of the cluster) replicating its blocks to another available node.



Word Count problem

Data locality

Blocks of a huge file are stored on **different nodes**

We can process them **locally** on respective nodes (reading fast from local disk)

We can achieve **ideal parallelization** for embarrassingly parallel tasks (which can be divided into independent sub-tasks, e.g. rows filtering in a huge table)

Word Count problem (Google)

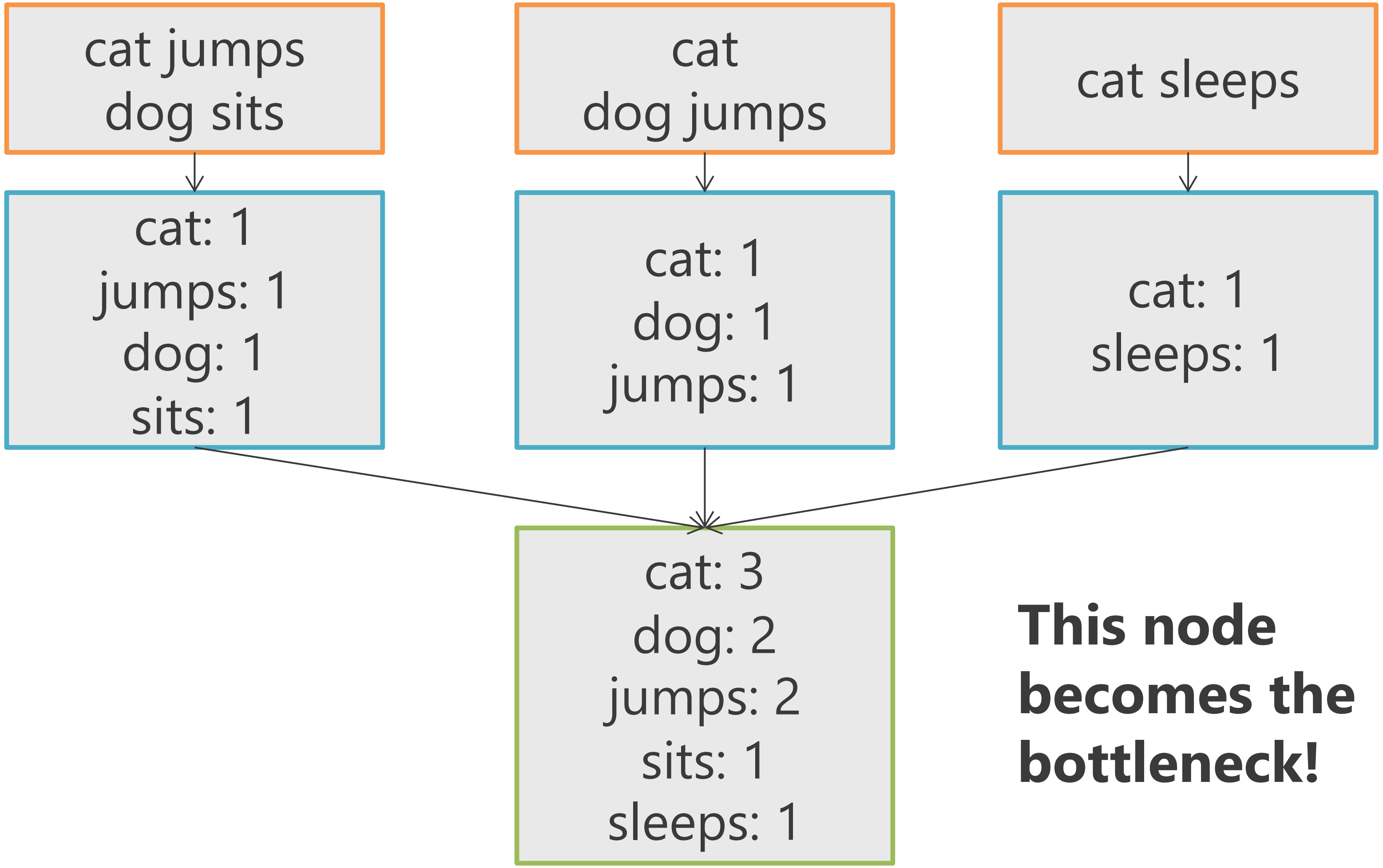
We have a huge text file (the whole Internet)

We want to know the frequency of every word (e.g. for tf-idf)

Solution:

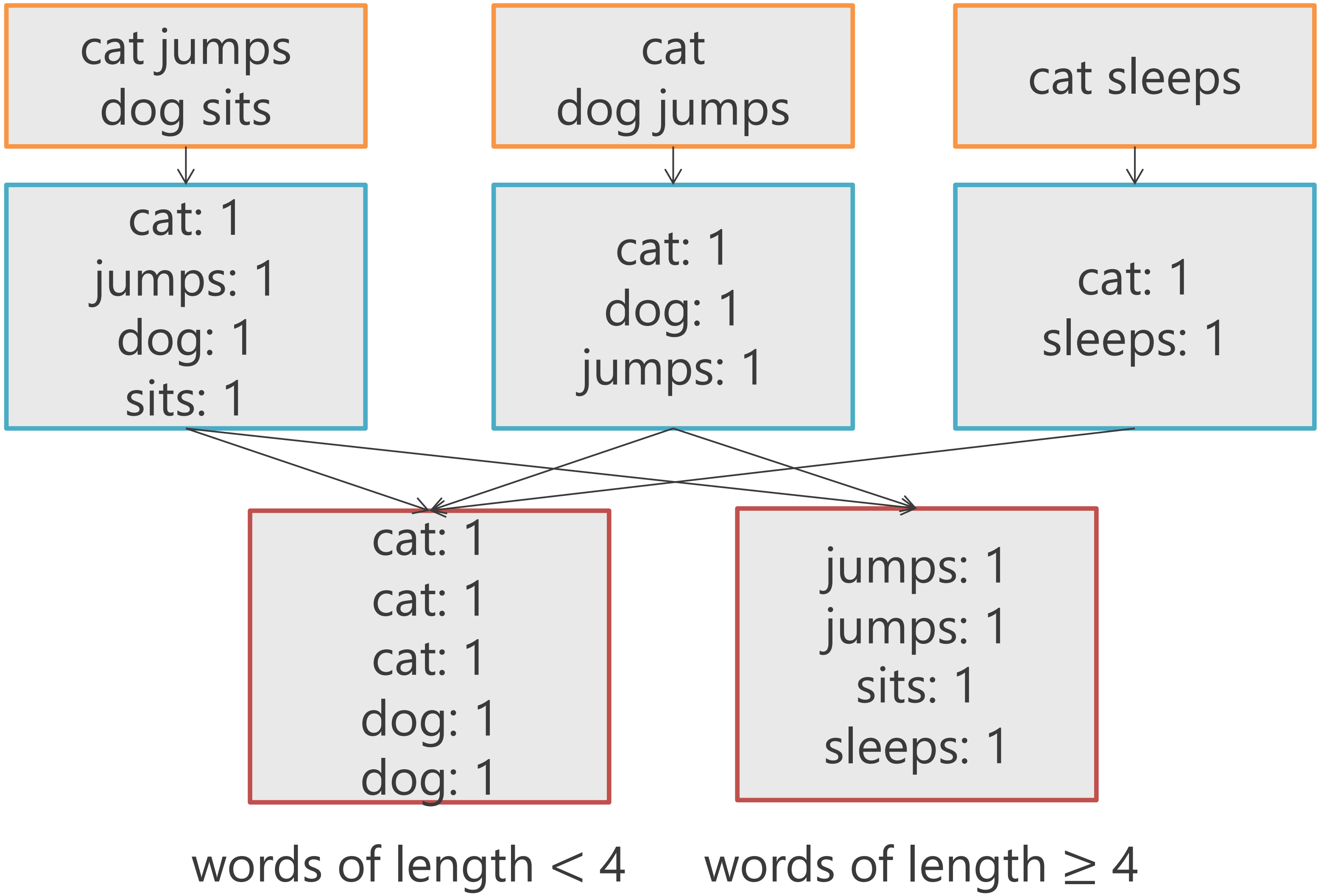
- For each block we calculate the frequency of words in it (*scales perfectly*)
- Aggregate frequencies from all blocks (*somehow*)

Naive approach

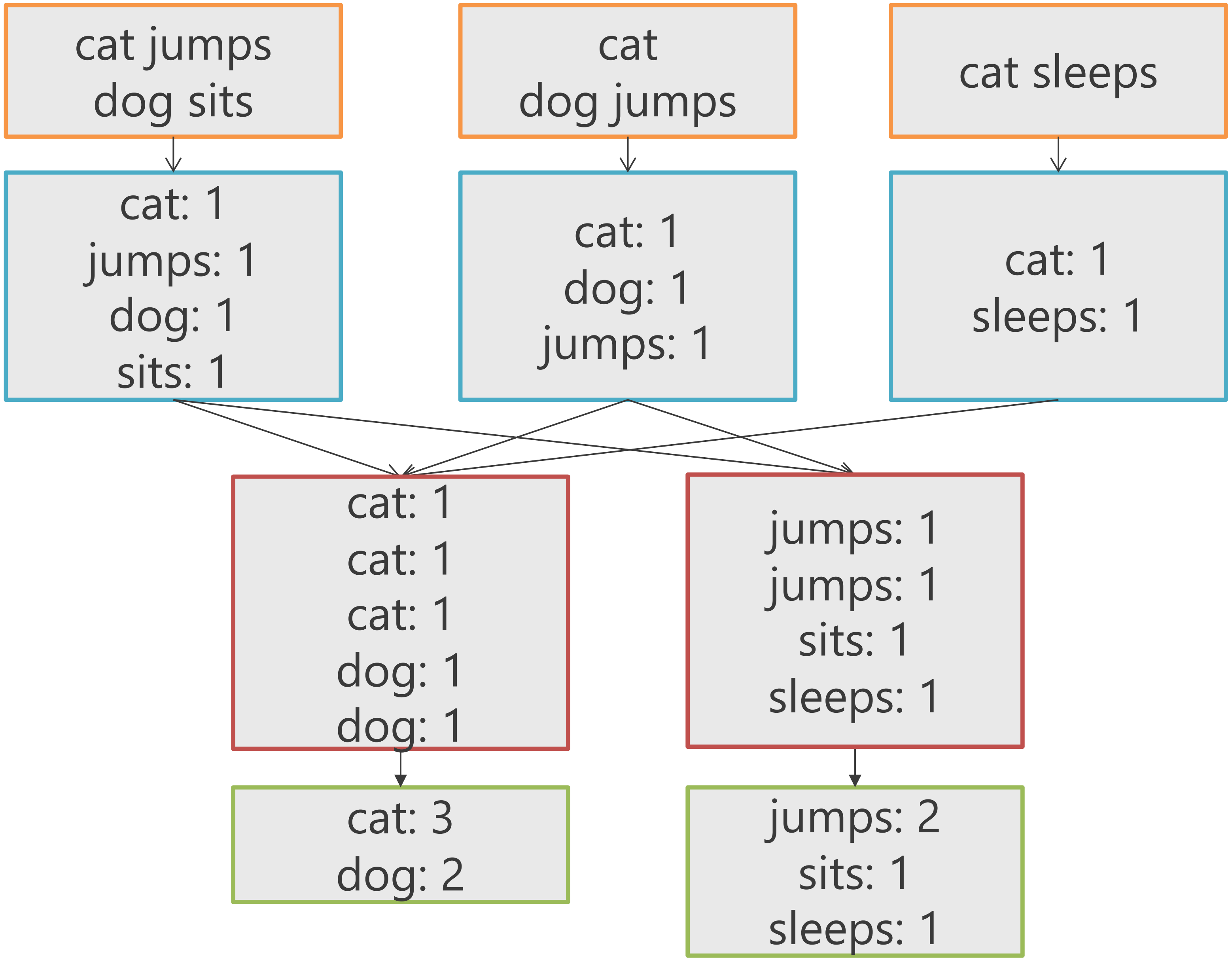


This node becomes the bottleneck!

Divide the aggregation work



Independently process halves



MapReduce

Divide the work into more tasks

Let’s say we want to divide the aggregation work into **N** tasks.

We will divide the work by **hash(word) % N**, which produces the node number where we need to send this word.

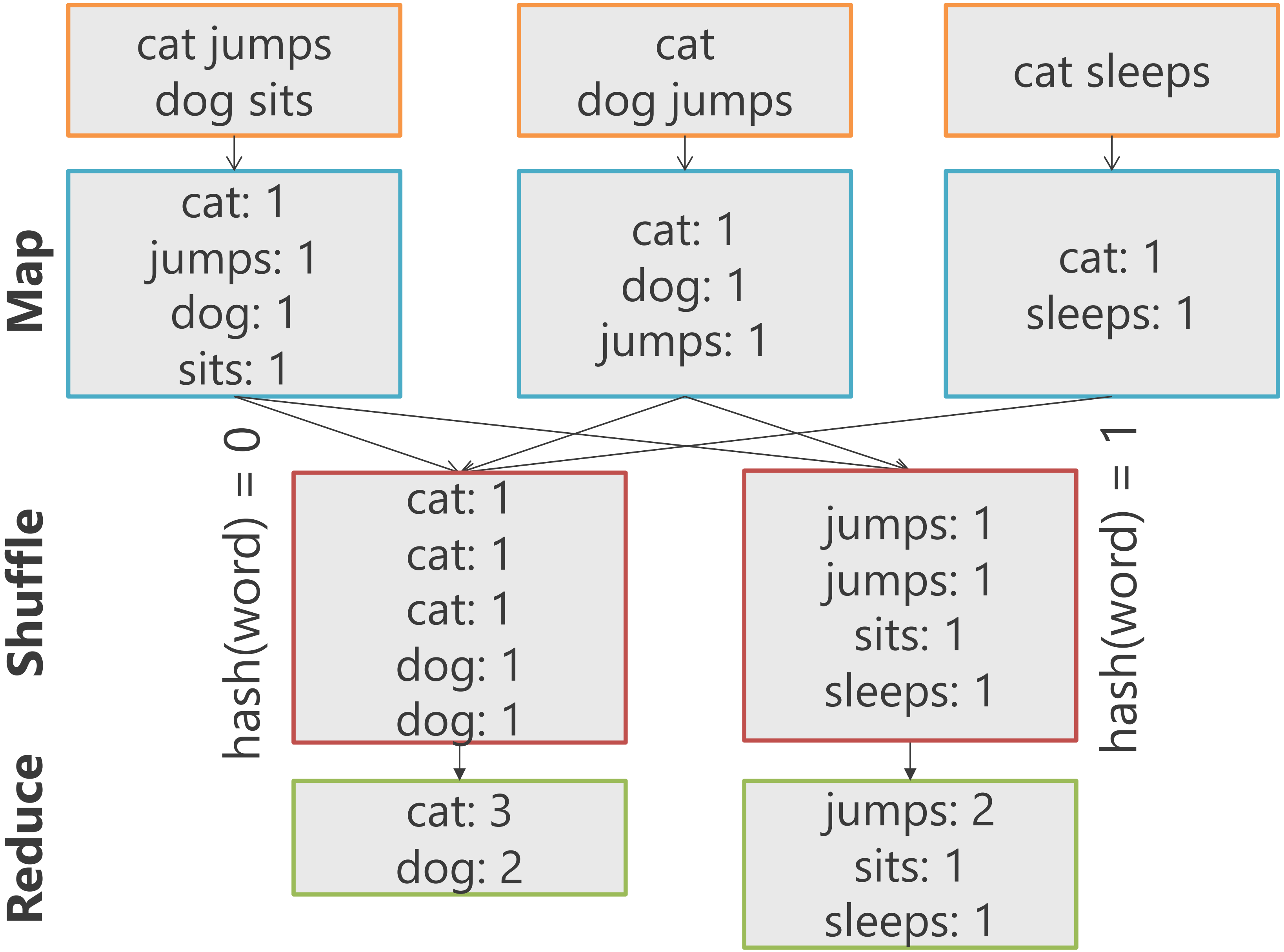
A good hash function produces numerical values that are uniformly distributed.

The example of a hash function (*polynomial*):

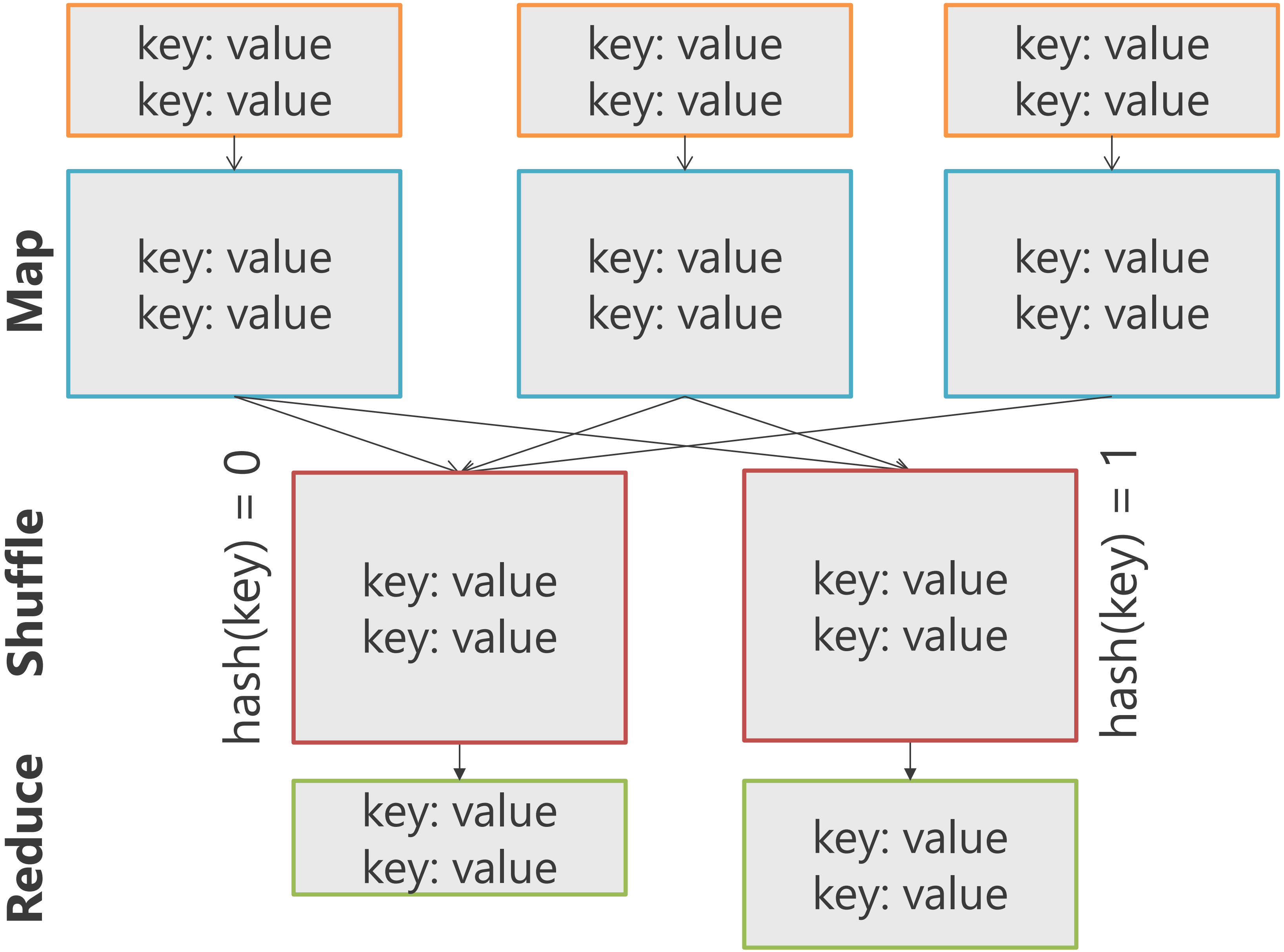
$$\mathbf{hash}(\boldsymbol{s}) = \boldsymbol{s}[\mathbf{0}] + \boldsymbol{s}[\mathbf{1}]\boldsymbol{p}^{\mathbf{1}} + \cdots + \boldsymbol{s}[\boldsymbol{n}]\boldsymbol{p}^{\boldsymbol{n}}$$

s – string
p – fixed prime number
$s[i]$ – character code

Word Count Problem



MapReduce paradigm



MapReduce paradigm

Map:

(K1, V1) → List(K2, V2)

(line number, "cat sleeps") → [("cat", 1), ("sleeps", 1)]

Shuffle:

Keys are divided by **hash(key) % N** into **N** partitions

Each partition is **sorted by key** (independently)

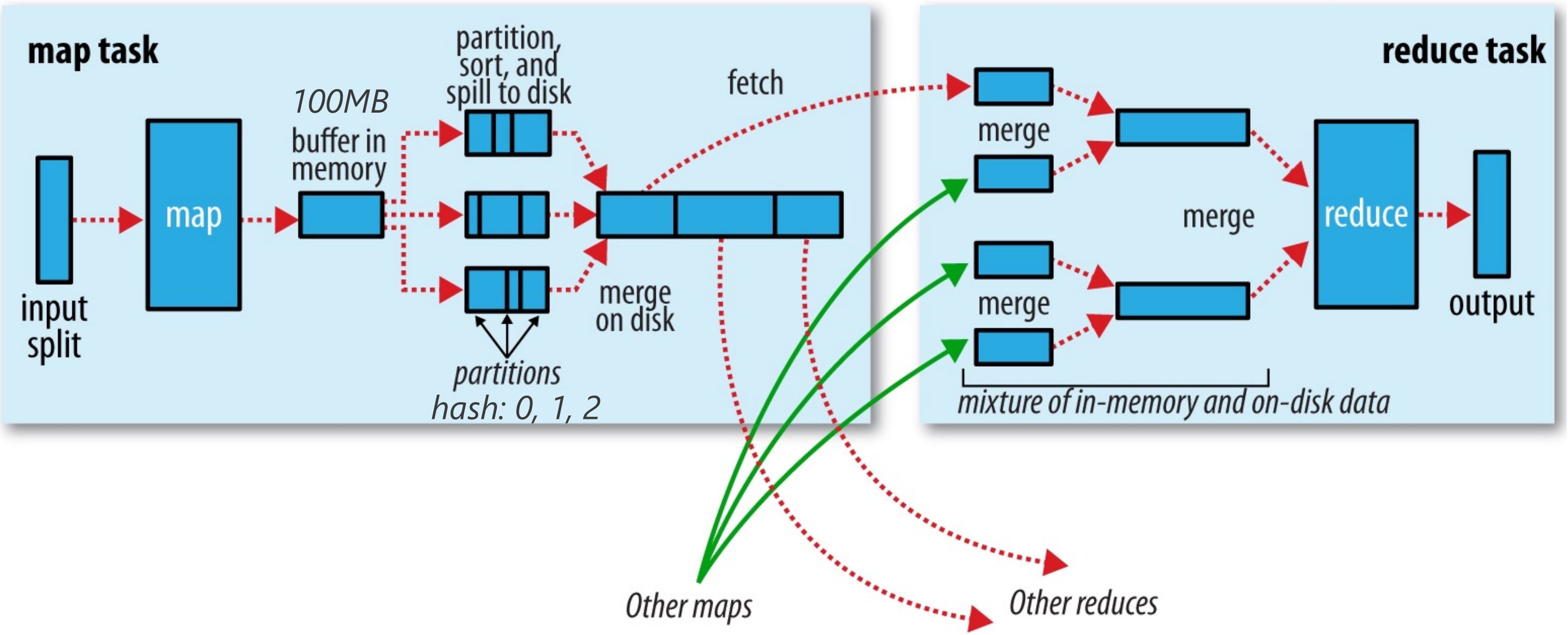
Reduce:

(K2, List(V2)) → List(K3, V3)

("cat", (1, 1, 1)) → [("cat", 3)]

Hadoop MapReduce

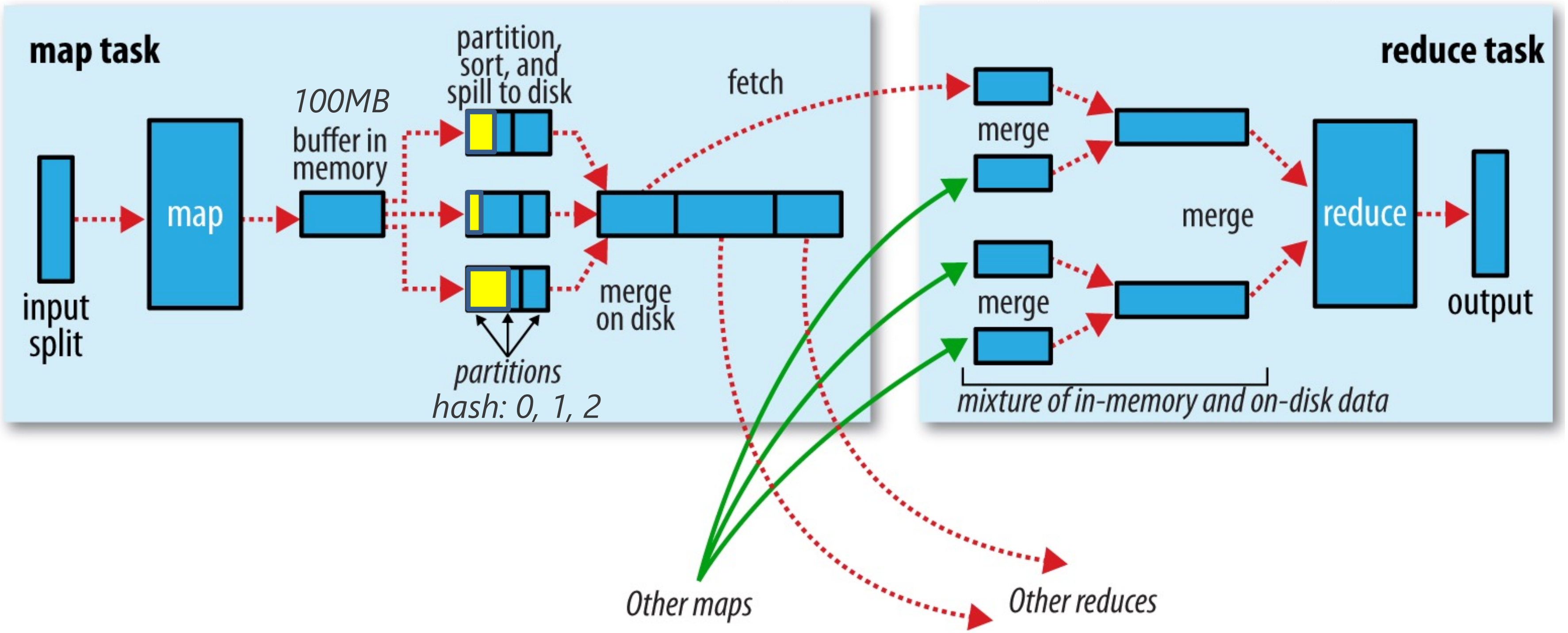
Accelerated shuffle



- Map results can be sorted locally
- You can then merge sorted Map results into one sorted result in linear time (merge sort)

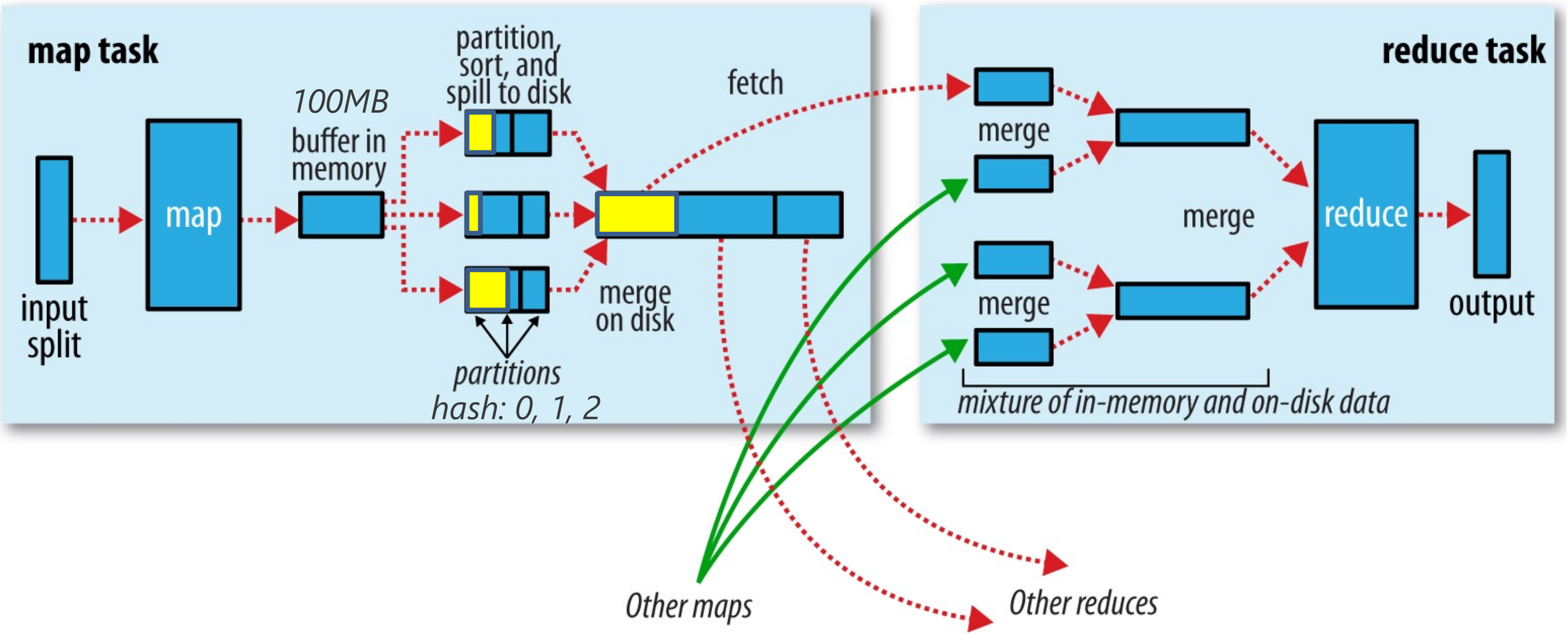
http://grut-computing.com/HadoopBook.pdf

Accelerated shuffle



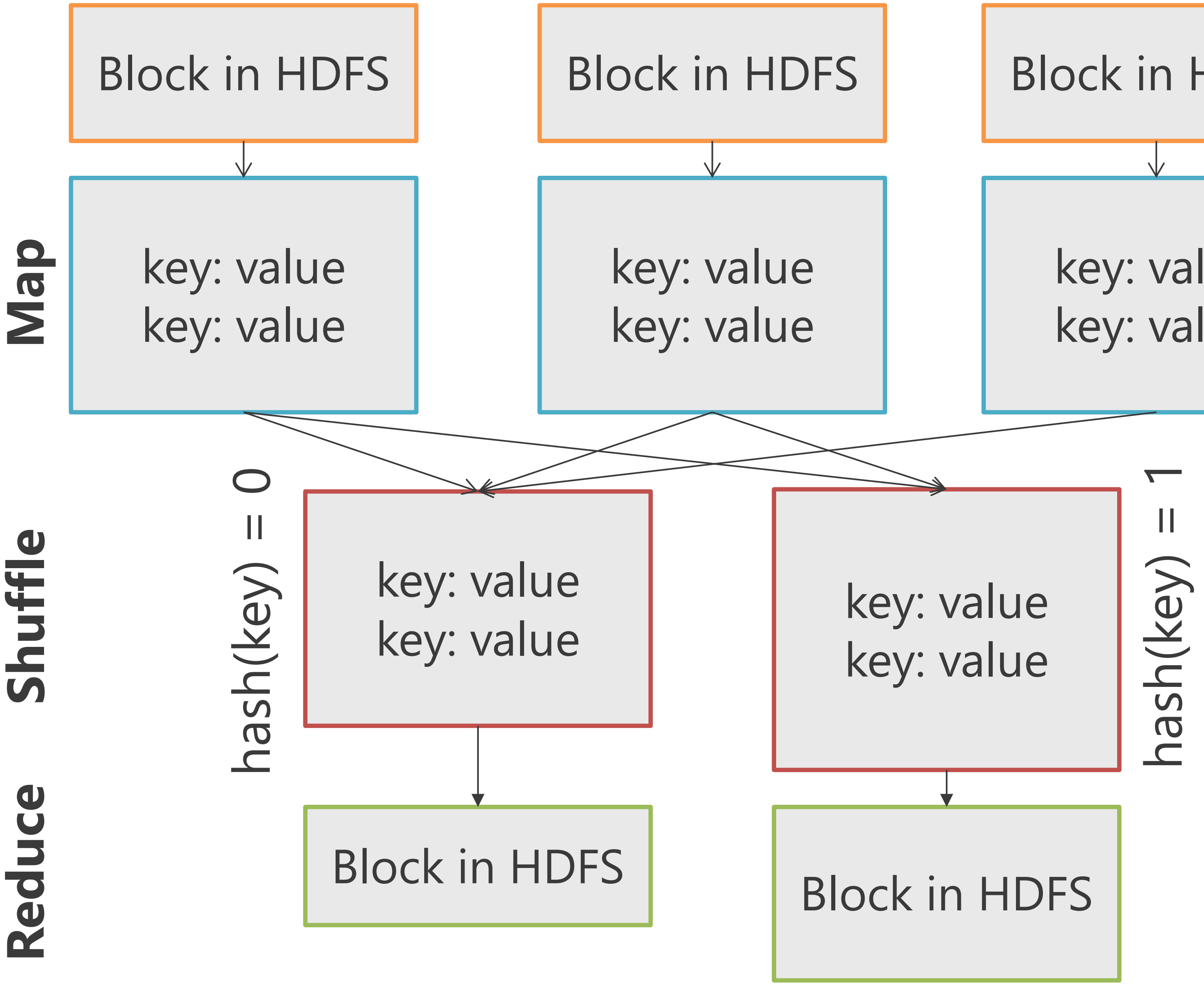
- Map results can be sorted locally
- You can then merge sorted Map results into one sorted result in linear time (merge sort)

Accelerated shuffle



- Map results can be sorted locally
- You can then merge sorted Map results into one sorted result in linear time (merge sort)

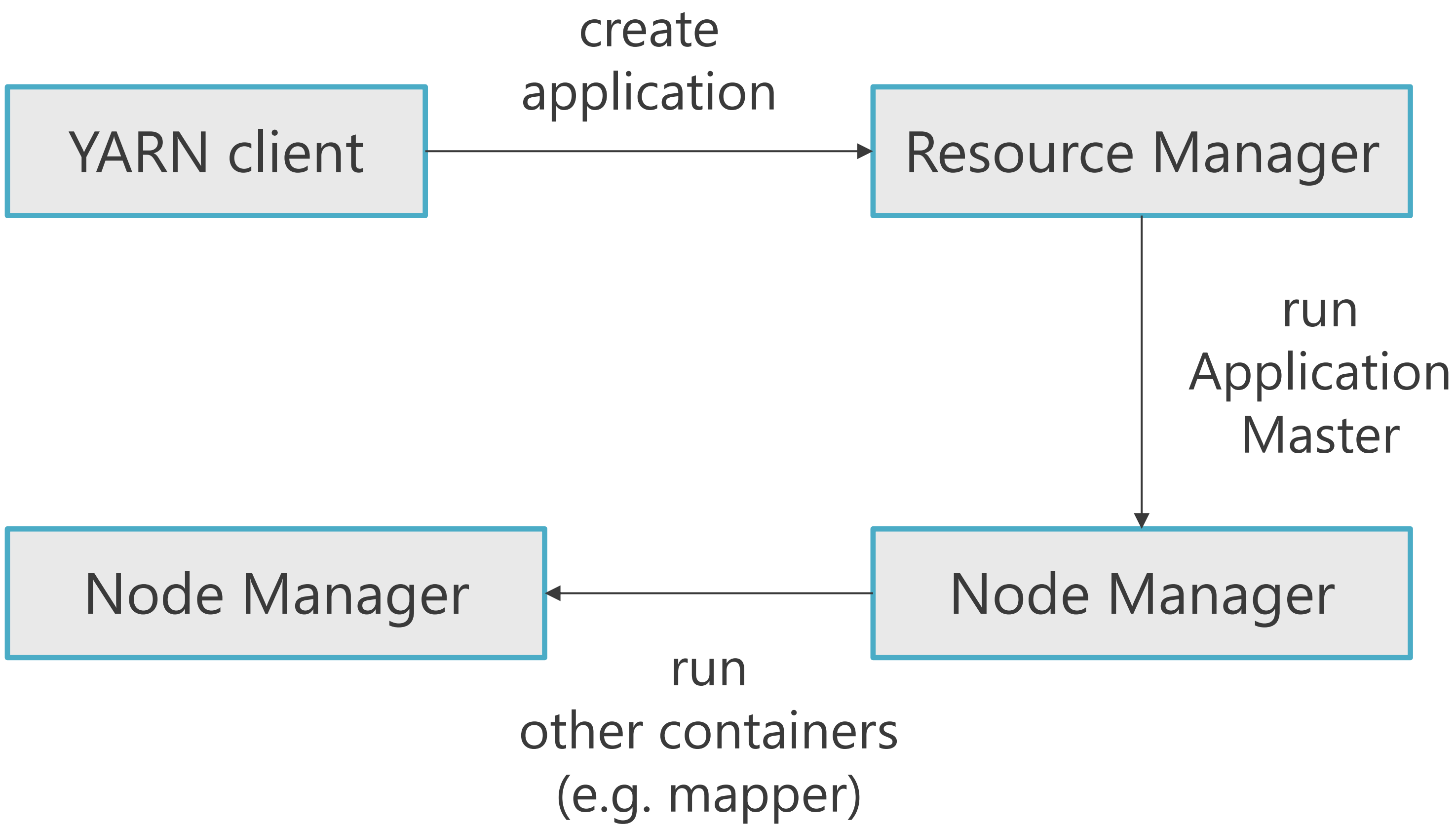
Working with HDFS



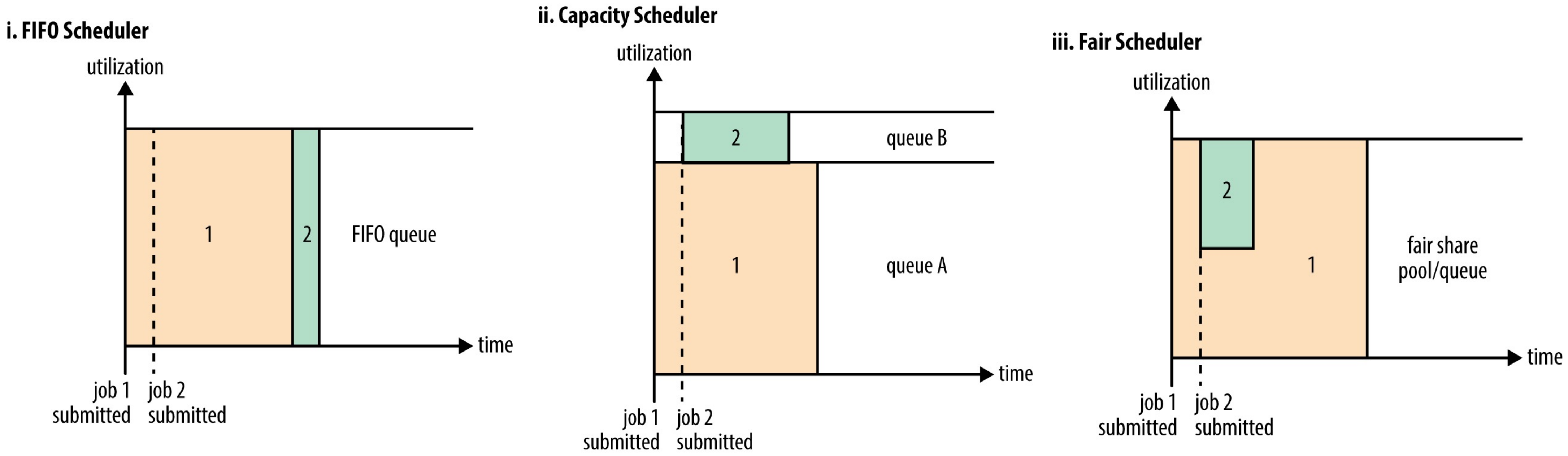
Reliability

- If you lose a mapper, you can restart the task only for its blocks
- When you lose a reducer, you re-collect data from all mappers only for its hash value

Working with YARN



YARN scheduling



Real problem example

MapReduce example

Map: (K1, V1) → List(K2, V2)

Reduce: (K2, List(V2)) → List(K3, V3)

UserId	TrackId	AlbumId
11123	4521	842
14322	3593	957
...	...	...

Let's collect listening logs on Spotify.

We want to find the most popular track in each album.

MapReduce example

Map: (K1, V1) → List(K2, V2)

Reduce: (K2, List(V2)) → List(K3, V3)

UserId	TrackId	AlbumId
11123	4521	842
14322	3593	957
...	...	...

Let's collect listening logs on Spotify.
We want to find the most popular track in each album.

M: #, (user, track, album) → **(album, track)**, 1
R: **(album, track)**, (1,1,...) → (album, track), count

MapReduce example

UserId	TrackId	AlbumId
11123	4521	842
14322	3593	957
...	...	...

Map: (K1, V1) → List(K2, V2)

Reduce: (K2, List(V2)) → List(K3, V3)

Let's collect listening logs on Spotify.
We want to find the most popular track in each album.

M: #, (user, track, album) → **(album, track)**, 1
R: **(album, track)**, (1,1,...) → (album, track), count

M: (album, track), count → **album**, (track, count)
R: **album**, tracks → album, most popular track

Conclusion

- Hadoop is created to handle Big Data and scale horizontally
- HDFS is a distributed and reliable file system
- MapReduce is a distributed and reliable way to process data stored in HDFS