

Hadoop

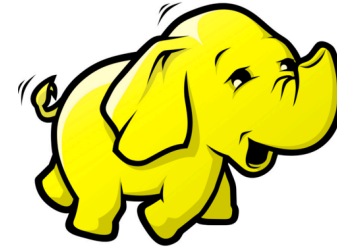
Enterprise Architectures for Big Data

Hadoop

- For many synonymous to Big Data Analytics
- Framework for solving data-intensive processes
- Need to parallelize computation across thousands of nodes
- Designed
 - to scale massively
 - for hardware and software failures
 - to run on commodity hardware
- Distributors: Cloudera, MapR
- Users: Facebook, Ebay, Amazon, Baidu, Yahoo, IBM, Apple, Microsoft, any many more

Origins of Hadoop

- Google
 - Google File System
 - MapReduce



The Google File System

Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung

Google

MapReduce: Simplified Data Processing on Large Clusters

Jeffrey Dean and Sanjay Ghemawat

jeff@google.com, sanjay@google.com

Google, Inc.

- Hadoop
 - Creator: Doug Cutting
 - Open Source
 - Hadoop Distributed File System
 - MapReduce

ABSTRACT

We have designed and implemented the Google File System, a scalable distributed file system for large distributed data-intensive applications. It provides fault tolerance while running on inexpensive commodity hardware, and it delivers high aggregate performance to a large number of clients.

While sharing many of the same goals as previous distributed file systems, our design has been driven by observations of our application workloads and technological environment, both current and anticipated, that reflect a marked departure from some earlier file system assumptions. This has led us to reexamine traditional choices and explore radically different design points.

The file system has successfully met our storage needs. It is widely deployed within Google as the storage platform for the generation and processing of data used by our service as well as research and development efforts that require large data sets. The largest cluster to date provides hundreds of terabytes of storage across thousands of data on over a thousand machines, and it is concurrently accessed by hundreds of clients.

In this paper, we present file system interface extensions designed to support distributed applications, discuss many aspects of our design, and report measurements from both micro-benchmarks and real world use.

Categories and Subject Descriptors

D [4]: 3—Distributed file systems

General Terms

Design, reliability, performance, measurement

Keywords

Fault tolerance, scalability, data storage, clustered storage

*The authors can be reached at the following addresses: {sanjay, gobioff, shuntak}@google.com.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

SOOP'05, October 19–22, 2005, Boston Landing, New York, USA.
Copyright 2005 ACM 1-58113-757-5/05/0010...\$5.00.

1. INTRODUCTION

We have designed and implemented the Google File System (GFS) to meet the rapidly growing demands of Google's data processing needs. GFS shares many of the same goals as previous distributed file systems such as performance, scalability, reliability, and availability. However, its design has been driven by key observations of our application workloads and technological environment, both current and anticipated, that reflect a marked departure from some earlier file system design assumptions. We have reexamined traditional choices and explored radically different points in the design space.

First, component failures are the norm rather than the exception. The file system consists of hundreds or even thousands of storage machines built from inexpensive commodity parts and is accessed by a comparable number of client machines. The quantity and quality of the components virtually guarantee that some are not functional at any given time and some will not recover from their current failure. We have seen problems caused by application bugs, operating system bugs, human errors, and the failure of disks, memory, connectors, networking, and power supplies. Therefore, constant monitoring, error detection, fault tolerance, and automatic recovery must be integral to the system.

Second, files are huge by traditional standards. Multi-GB files are common. Each file typically contains many application objects such as web documents. When we are regularly working with fast growing data sets of many Tfs comprising billions of objects, it is unwise to manage billions of approximately KB-sized files even when the file system could support it. As a result, design assumptions and parameters such as I/O operation and block sizes have to be revisited.

Third, most files are created by appending new data rather than overwriting existing data. Random writes within a file are practically non-existent. Once written, the files are only read, and often only sequentially. A variety of data share those characteristics. Some may constitute large repositories that data analysis programs scan through. Some may be data streams continuously generated by running applications. Some may be archival data. Some may be intermediate results produced on one machine and processed on another, whether simultaneously or later in time. Given this access pattern on large files, appending becomes the focus of performance optimization and slowness guarantees, while caching data blocks in the client loses its appeal.

Fourth, co-designing the applications and the file system API benefits the overall system by increasing our flexibility.

Abstract

MapReduce is a programming model and an associated implementation for processing and generating large data sets. Users specify a *map* function that processes a key/value pair to generate a set of intermediate key/value pairs, and a *reduce* function that merges all intermediate values associated with the same intermediate key. Many real world tasks are expressible in this model, as shown in the paper.

Programs written in this functional style are automatically parallelized and executed on a large cluster of commodity machines. The run-time system takes care of the details of partitioning the input data, scheduling the program's execution across a set of machines, handling machine failures, and managing the required inter-machine communication. This allows programmers without any experience with parallel and distributed systems to easily utilize the resources of a large distributed system.

Our implementation of MapReduce runs on a large cluster of commodity machines and is highly scalable: a typical MapReduce computation processes many terabytes of data on thousands of machines. Programmers find the system easy to use: hundreds of MapReduce programs have been implemented and upwards of one thousand MapReduce jobs are executed on Google's clusters every day.

1 Introduction

Over the past five years, the authors and many others at Google have implemented hundreds of special-purpose computations that process large amounts of raw data, such as crawled documents, web request logs, etc., to compute various kinds of derived data, such as inverted indices, various representations of the graph structure of web documents, summaries of the number of pages crawled per host, the set of most frequent queries in a

given day, etc. Most such computations are conceptually straightforward. However, the input data is usually large and the computations have to be distributed across hundreds or thousands of machines in order to finish in a reasonable amount of time. The issues of how to parallelize the computation, distribute the data, and handle failures conspire to obscure the original simple computation with large amounts of complex code to deal with these issues.

As a reaction to this complexity, we designed a new abstraction that allows us to express the simple computations we were trying to perform but hides the messy details of parallelization, fault-tolerance, data distribution and load balancing in a library. Our abstraction is inspired by the *map* and *reduce* primitives present in Lisp and many other functional languages. We realized that most of our computations involved applying a *map* operation to each logical "record" in our input in order to compute a set of intermediate key/value pairs, and then applying a *reduce* operation to all the values that shared the same key, in order to combine the derived data appropriately. Our use of a functional model with user-specified *map* and *reduce* operations allows us to parallelize large computations easily and to use re-execution as the primary mechanism for fault tolerance.

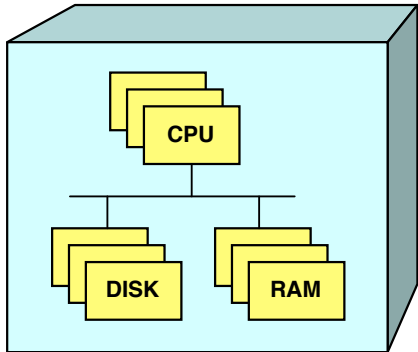
The major contributions of this work are a simple and powerful interface that enables automatic parallelization and distribution of large-scale computations, combined with an implementation of this interface that achieves high performance on large clusters of commodity PCs.

Section 2 describes the basic programming model and gives several examples. Section 3 describes an implementation of the MapReduce interface tailored towards our cluster-based computing environment. Section 4 describes several refinements of the programming model that we have found useful. Section 5 has performance measurements of our implementation for a variety of tasks. Section 6 explores the use of MapReduce within Google including our experiences in using it as the basis

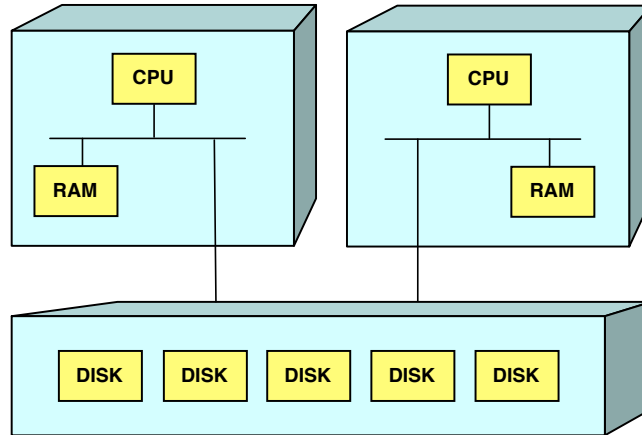
To appear in OSDI 2004

Scaling Architectures: Scale up vs. Scale out

Shared Everything
(Scale Up)

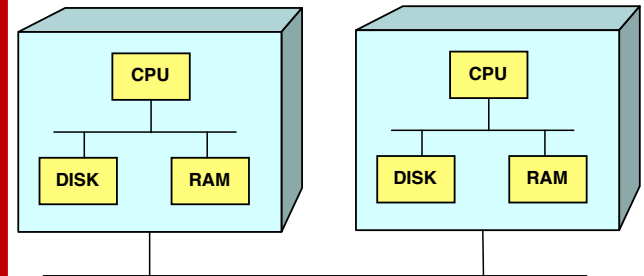


Shared Disk (Scale Out)



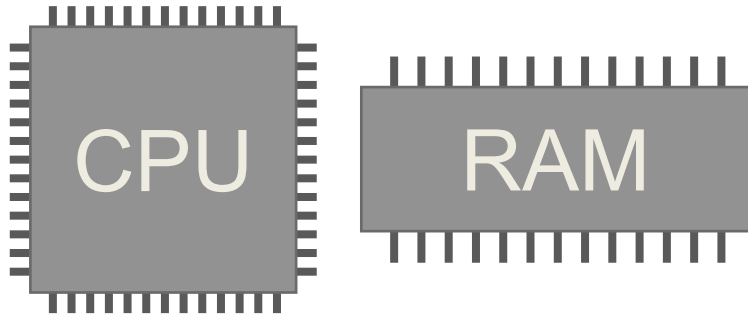
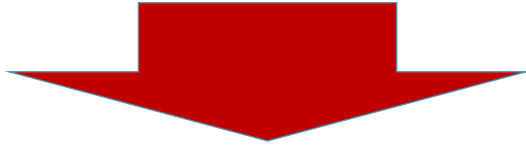
Hadoop

Shared Nothing (Scale Out)

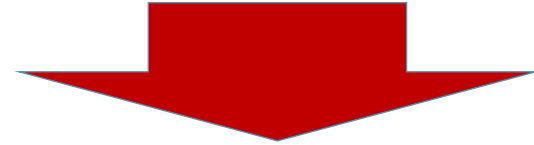


Hadoop Core = Computation + Storage

**Yet Another Resource
Negotiator (YARN)**

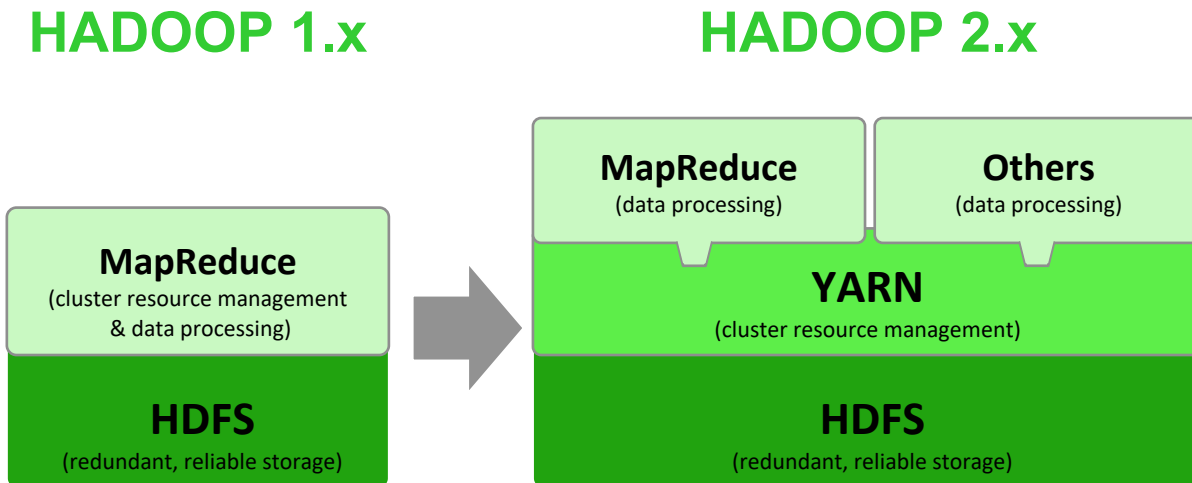


**Hadoop Distributed
File System (HDFS)**

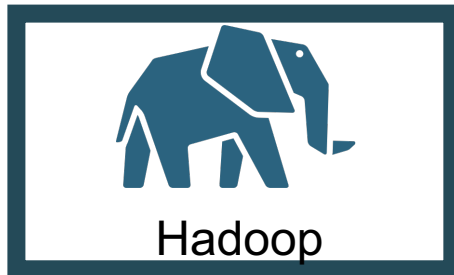


From Hadoop 1.x to Hadoop 2.x

- YARN is a re-architecture of Hadoop that allows multiple applications to run on the same platform

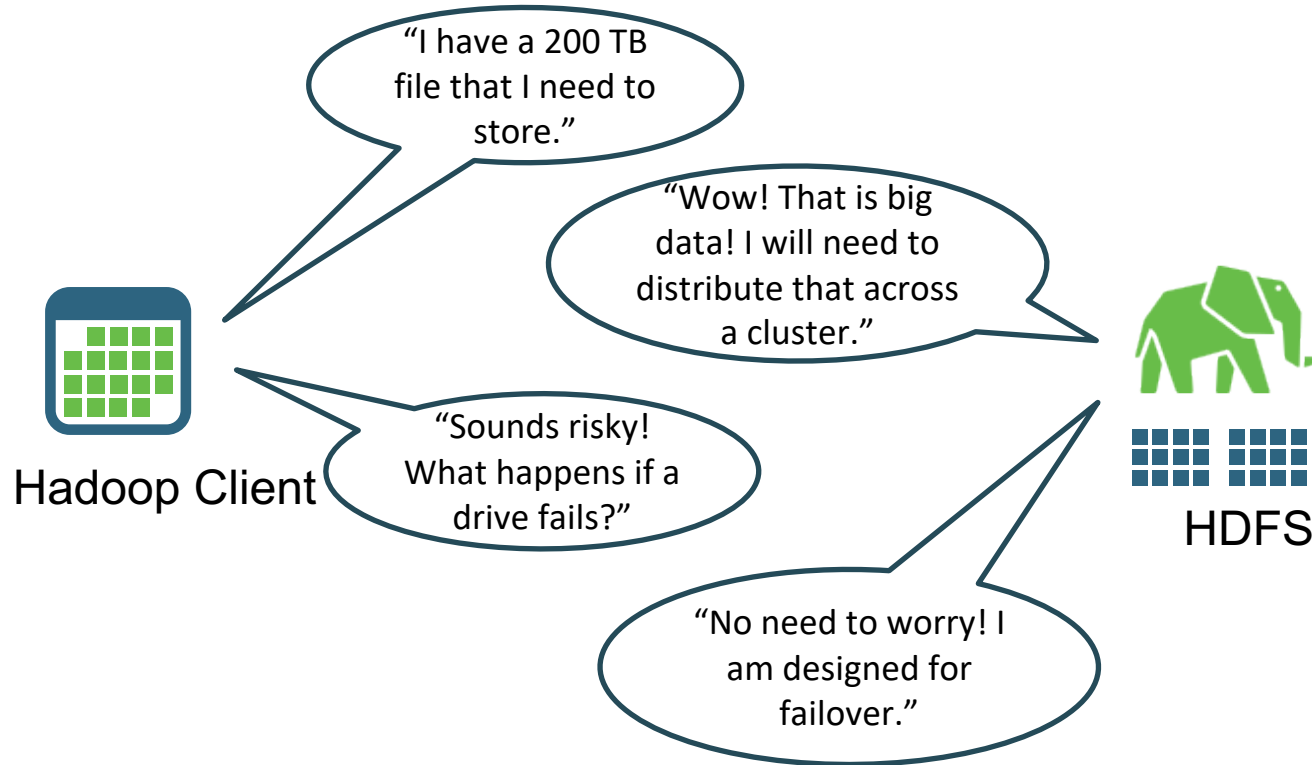


The Hadoop Ecosystem



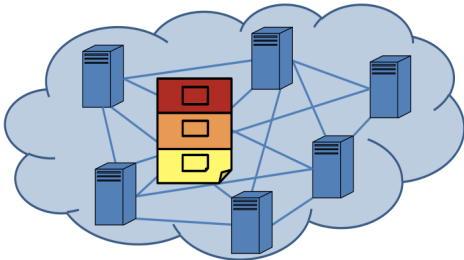
Hadoop Distributed File System (HDFS)

Hadoop Distributed File System (HDFS)

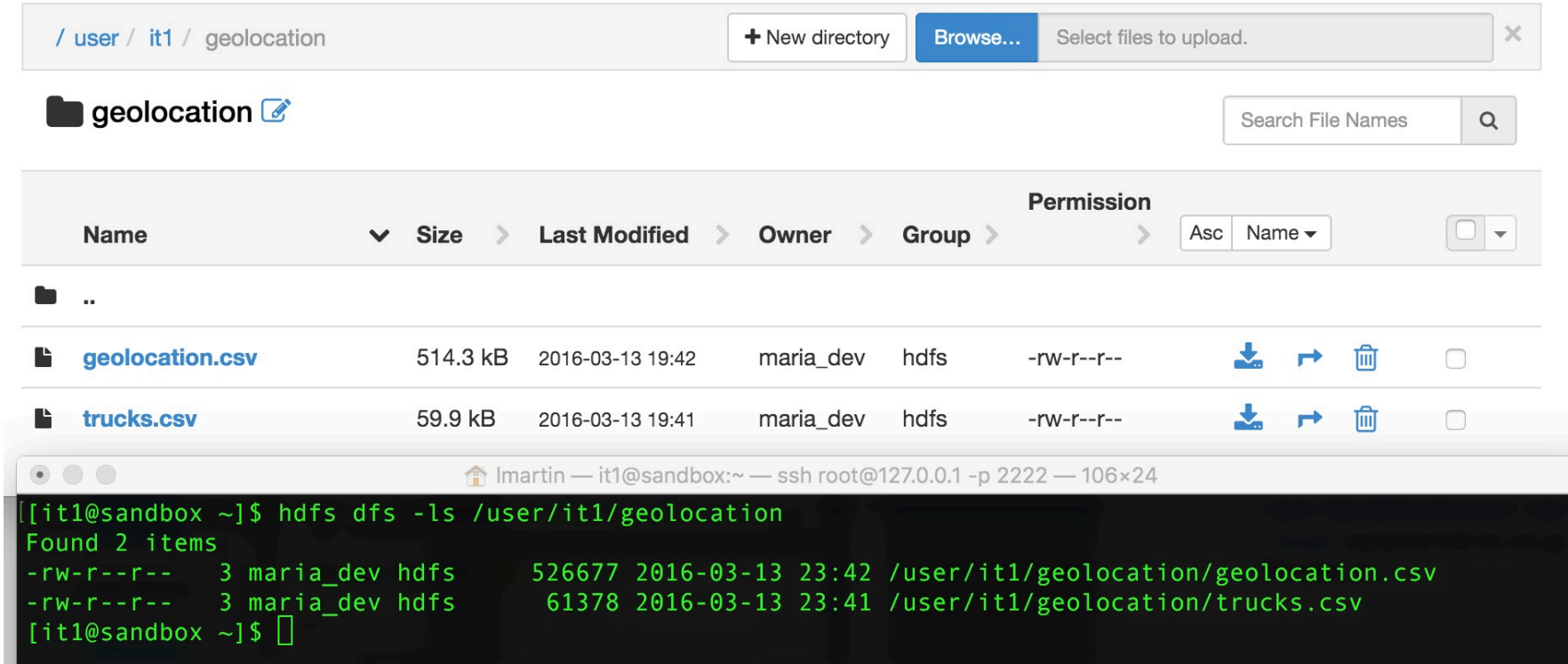


Hadoop Distributed File System (HDFS)

- Inspired by Google File System
- Distributed storage for large files
- Files are split up in multiple parts
 - Default block size 64MB (many use even 128MB block size)
 - Block size of a normal File system 4KB
- Parts are spread over the HDFS nodes
- Each part is replicated (default 3 times)



HDFS Looks Like a File System (but is distributed)



The image shows a web-based file browser interface for HDFS and a terminal window. The web interface displays the path `/ user / it1 / geolocation` and contains a table of files. The table has columns for Name, Size, Last Modified, Owner, Group, and Permission. Two files are listed: `geolocation.csv` (514.3 kB) and `trucks.csv` (59.9 kB), both owned by `maria_dev` in the `hdfs` group with permissions `-rw-r--r--`. Below the table, a terminal window shows the command `hdfs dfs -ls /user/it1/geolocation` being executed, which returns the same file listing information.

Name	Size	Last Modified	Owner	Group	Permission
..					
geolocation.csv	514.3 kB	2016-03-13 19:42	maria_dev	hdfs	-rw-r--r--
trucks.csv	59.9 kB	2016-03-13 19:41	maria_dev	hdfs	-rw-r--r--

```
[it1@sandbox ~]$ hdfs dfs -ls /user/it1/geolocation
Found 2 items
-rw-r--r--  3 maria_dev hdfs      526677 2016-03-13 23:42 /user/it1/geolocation/geolocation.csv
-rw-r--r--  3 maria_dev hdfs      61378 2016-03-13 23:41 /user/it1/geolocation/trucks.csv
[it1@sandbox ~]$
```

HDFS on the Command Line

- Like Unix Command Line commands (but distributed)

- E.g. List (ls) on HDFS

```
hadoop fs -ls
```

- The put command to upload local files to HDFS

```
hadoop fs -put mylocalfile /some/hdfs/path
```

- The get command to download files from HDFS

```
hadoop fs -get /some/hdfs/path/filename.txt
```

HDFS on the Command Line

```
hadoop fs -command [args]
```

A few of the many HDFS commands:

- -cat: display file content (uncompressed)
- -text: just like cat but works on compressed files
- -chgrp,-chmod,-chown: changes file permissions
- -put,-get,-copyFromLocal,-copyToLocal: copies files from the local file system to the HDFS and vice versa.
- -ls, -ls -R: list files/directories
- -mv,-moveFromLocal,-moveToLocal: moves files
- -stat: statistical info for any given file (block size, number of blocks, file type, etc.)

HDFS Components

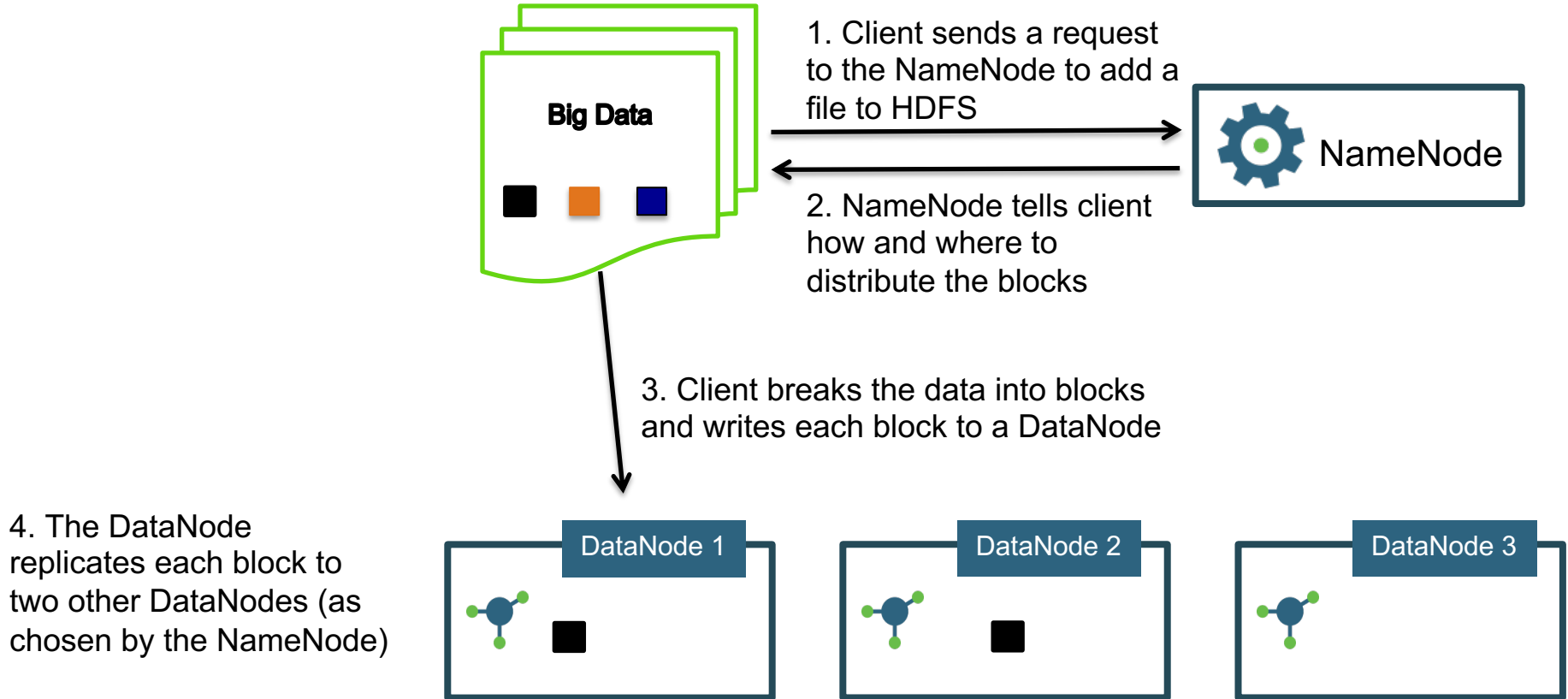
- NameNode

- Is the “master” node of HDFS
- Determines and maintains how the chunks of data are distributed across the DataNodes

- DataNode

- Stores the chunks of data, and is responsible for replicating the chunks across other DataNodes

How does HDFS work?



HDFS Architecture

The NameNode (master node) and DataNodes (worker nodes) are daemons running in a Java virtual machine.

NameNode

Namespace

- Hierarchy
- Directory names
- File names

Block Map

- File names > block IDs

Metadata

- Permissions and ownership
- ACLs
- Block size and replication level
- Access and last modification times
- User quotas

Block Storage

- Data blocks

DataNode



DataNode



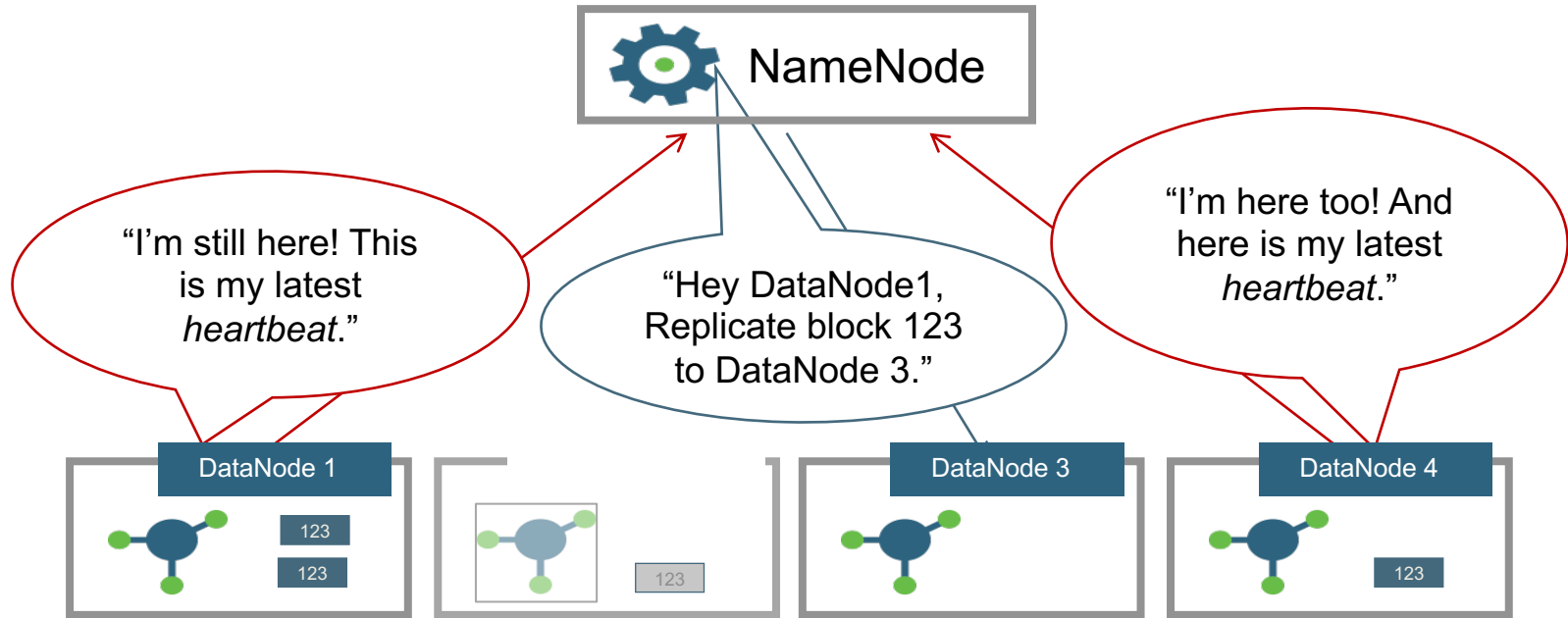
DataNode



DataNode



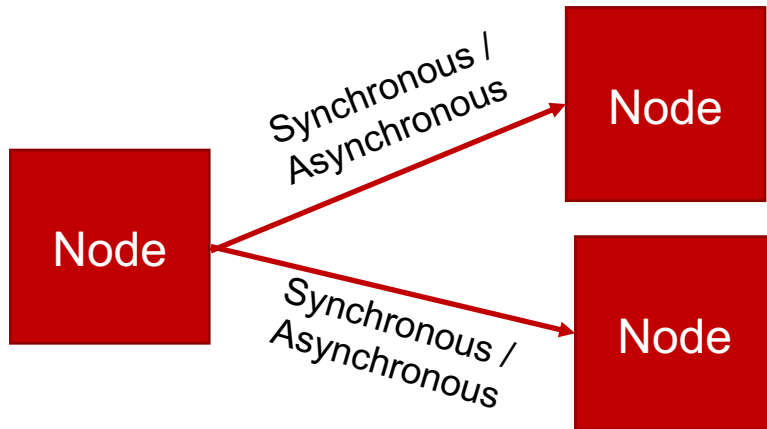
HDFS Automatic Replication and Failover



HDFS uses Replication and Partitioning (Sharding)

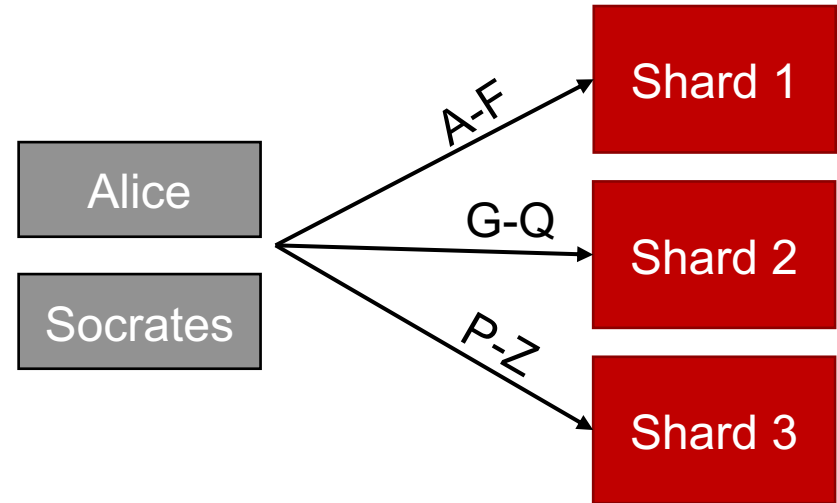
Replication

- Stores N copies of each data item



Partitioning

- Horizontal distribution of data over nodes



Hadoop Distributed File System (HDFS)

Advantages

- Scales to Petabytes
- Fault tolerant (Nodes and Disks)
- Integrates well with other Hadoop Services like MapReduce
- Runs on commodity hardware
- Commands similar to Unix Command Line
- Easy to manage
 - Automatically manage addition/removal of nodes
 - Failed Disks and Nodes do not need to be replaced immediately
 - One operator can manage thousands of nodes

Disadvantages

- Overhead of replication storage
- For small files, overhead because of minimal block size
- Immutable files
 - Files cannot be modified.
 - Need to be created from scratch.
 - Append action is the only action possible
- Not mountable
- Optimized for streaming reads of large files not random access of files
- Cluster management is difficult

MapReduce

Distributed processing is non-trivial

- How to assign tasks to different workers in an efficient way?
- What happens if tasks fail?
- How do workers exchange results?
- How to synchronize distributed tasks allocated to different workers?



Image courtesy of Master isolated images at FreeDigitalPhotos.net

Map

```
map(len, mylist)
```

len (

"aaa"
"b"
"cc"
"dddd"
"e"
"ff"

)



3
1
2
4
1
2

mylist

Map

„By abstracting away the very concept of looping, you can implement looping any way you want, including implementing it in a way that scales nicely with extra hardware.“

Joel Spolsky

Map

- Embarrassingly Parallel Task
- Can be executed in parallel
- No side effects
- No global variables
- No communication needs during the map phase
- Pure function

Reduce

```
reduce(lambda x, y: x+y, mylist)
```

sum(

3

)



1

2

4

1

2

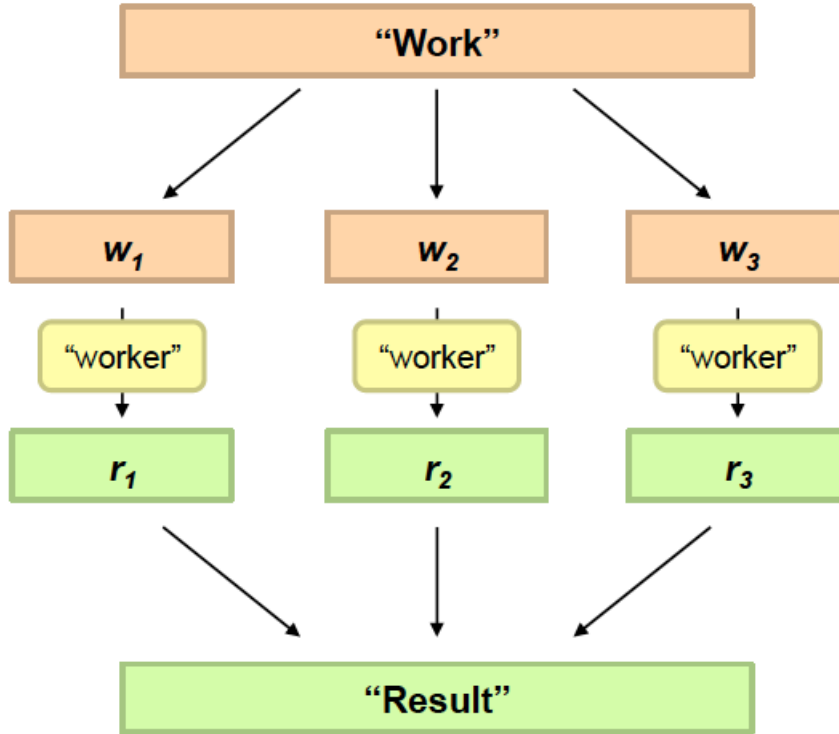
13

mylist

MapReduce explained in 41 words

- **Goal:** count the number of books in the library.
- **Map:** You count up shelf #1, I count up shelf #2.
 - (The more people we get, the faster this part goes.)
- **Reduce:** We all get together and add up our individual counts.

Divide and Conquer



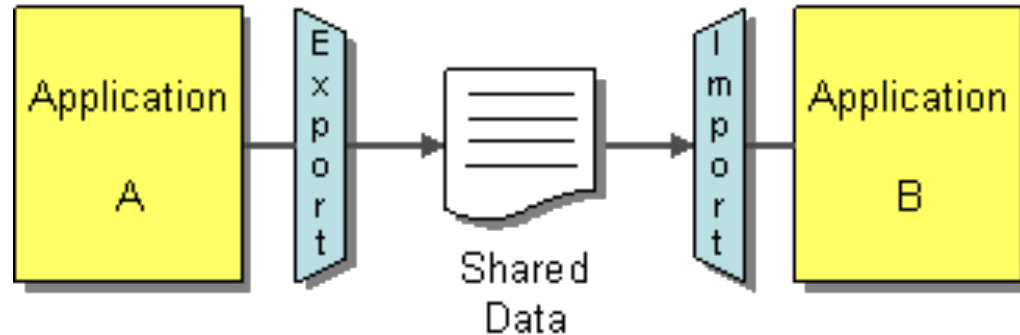
Divide Work



Combine Results

MapReduce uses File Transfer Integration

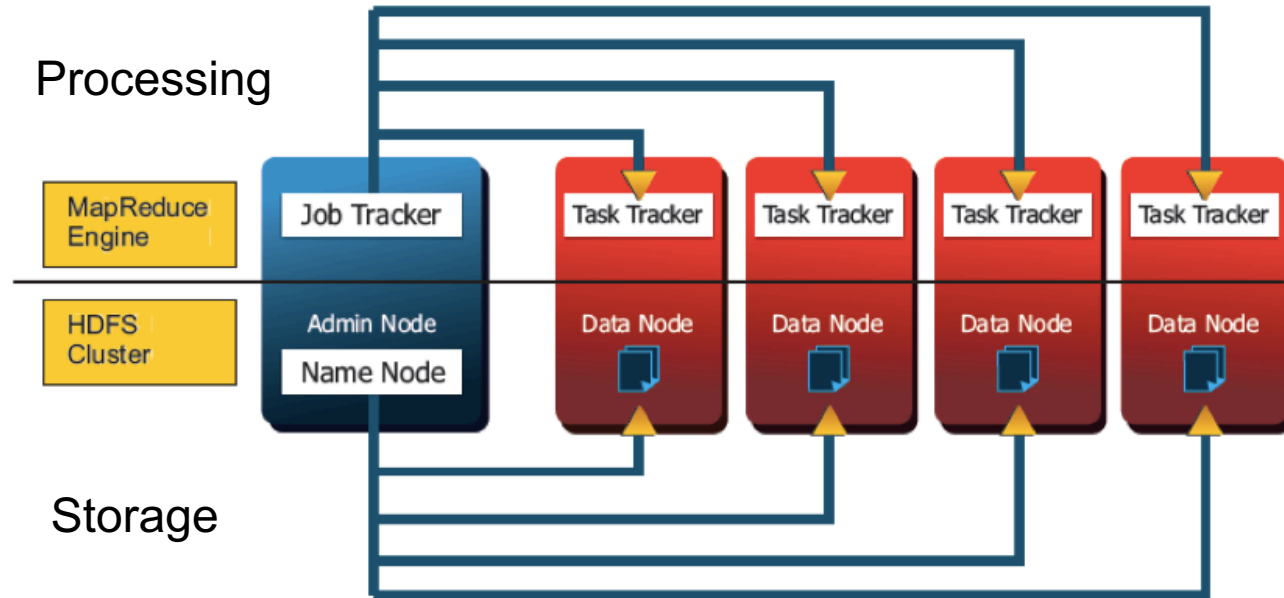
- **Problem:** How can I integrate multiple distributed applications or services so that they work together and can exchange information?



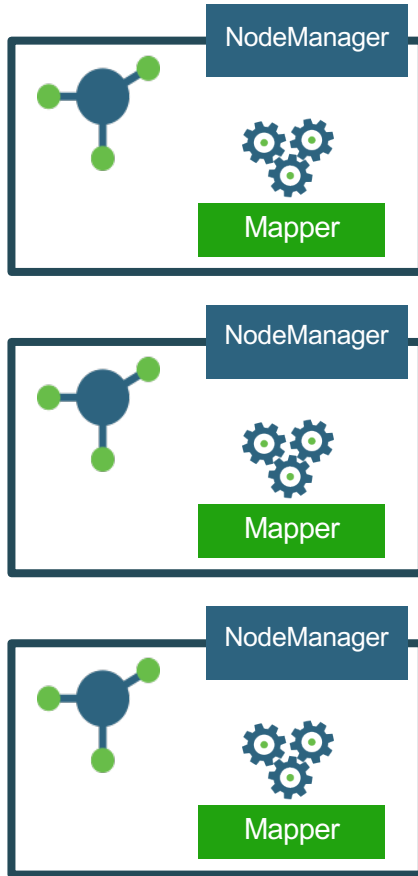
- **Solution:** Have each application produce **files** containing information that other applications need to consume.

Data Locality and MapReduce + HDFS

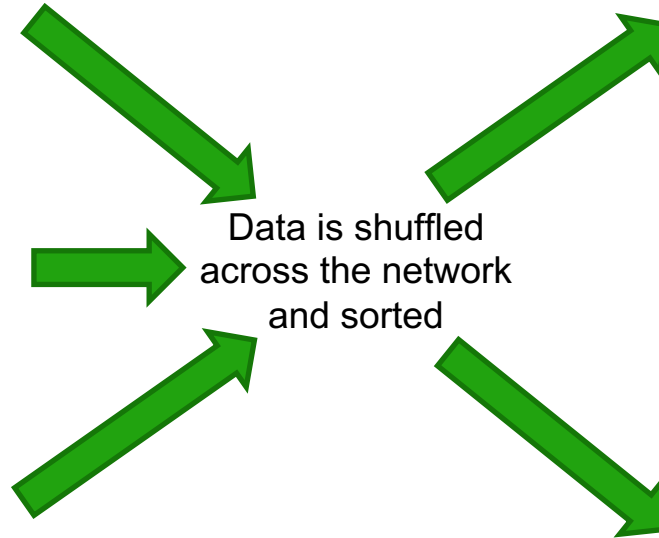
- Move the computation to where the data is
- Minimize the movement of the data (data locality)



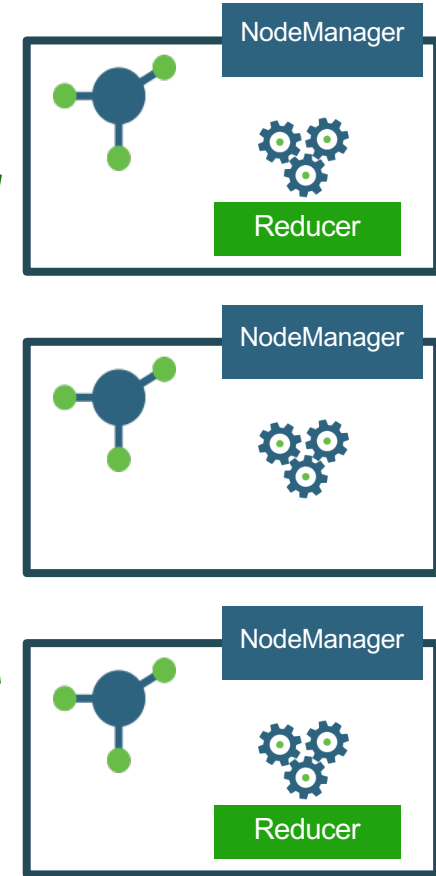
Map Phase



Shuffle/Sort (Combine)



Reduce Phase



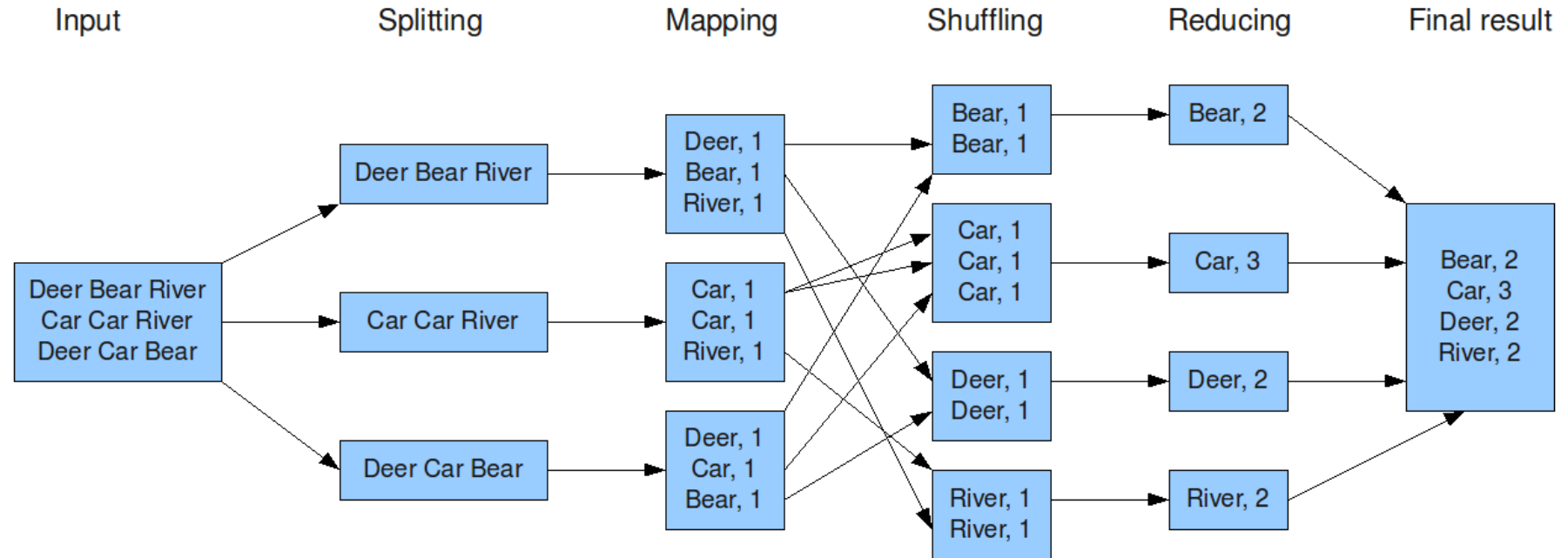
The Mapper

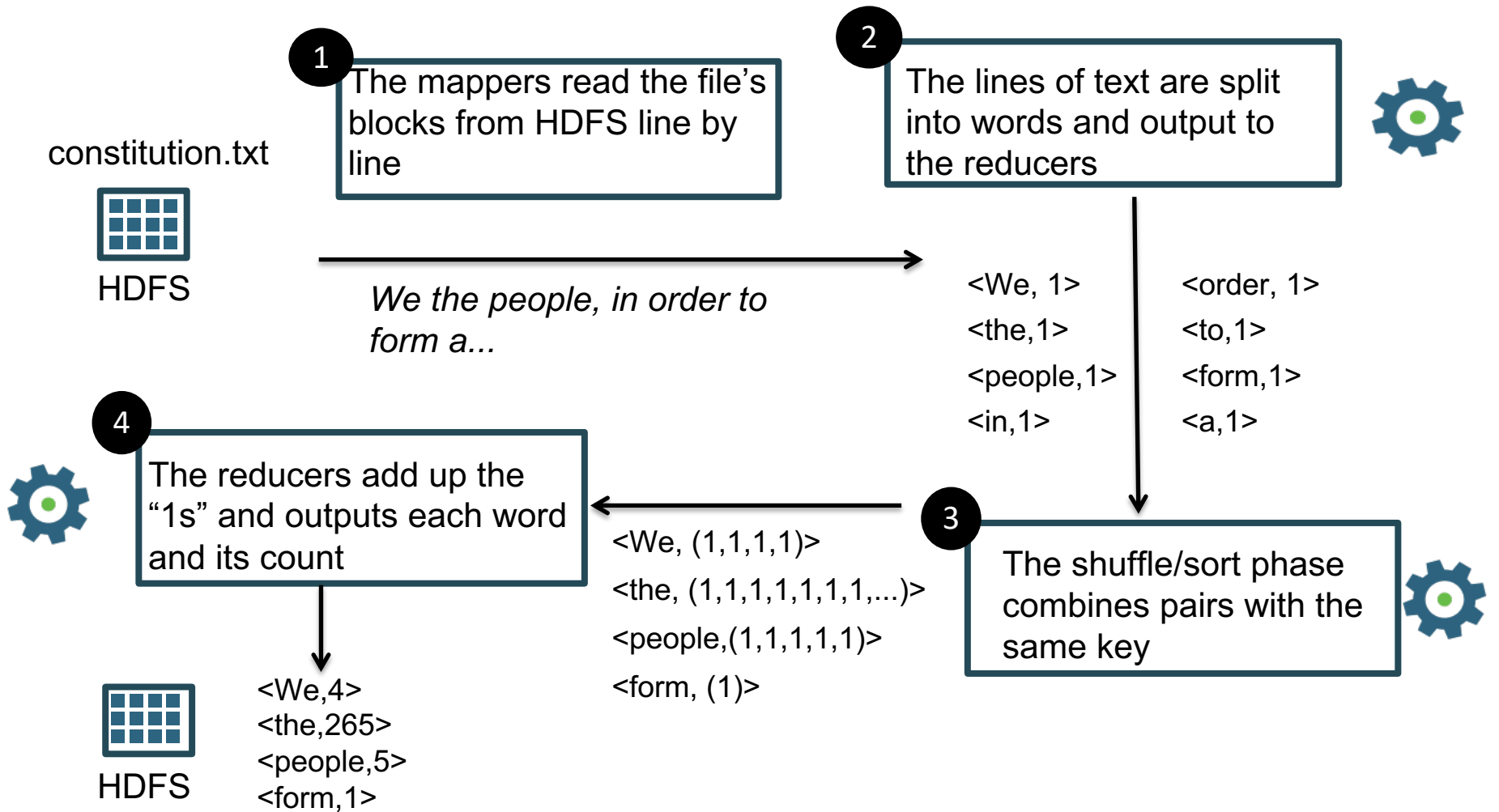
- The Mapper reads data in the form of key/value pairs
- It outputs zero or more key/value pairs
- The Mapper may use or completely ignore the input key
 - A standard pattern is to read a line of a file at a time
- If the Mapper writes anything out, it must in the form of key/value pairs

The Reducer

- After the Map phase is over, all the intermediate values for a given intermediate key are combined together into a list
- This list is given to a Reducer
 - There may be a single Reducer, or multiple Reducers
 - All values associated with a particular intermediate key are guaranteed to go to the same Reducer
 - The intermediate keys, and their value lists, are passed in sorted order
- The Reducer outputs zero or more Key-Value-Pairs
 - These are written to HDFS

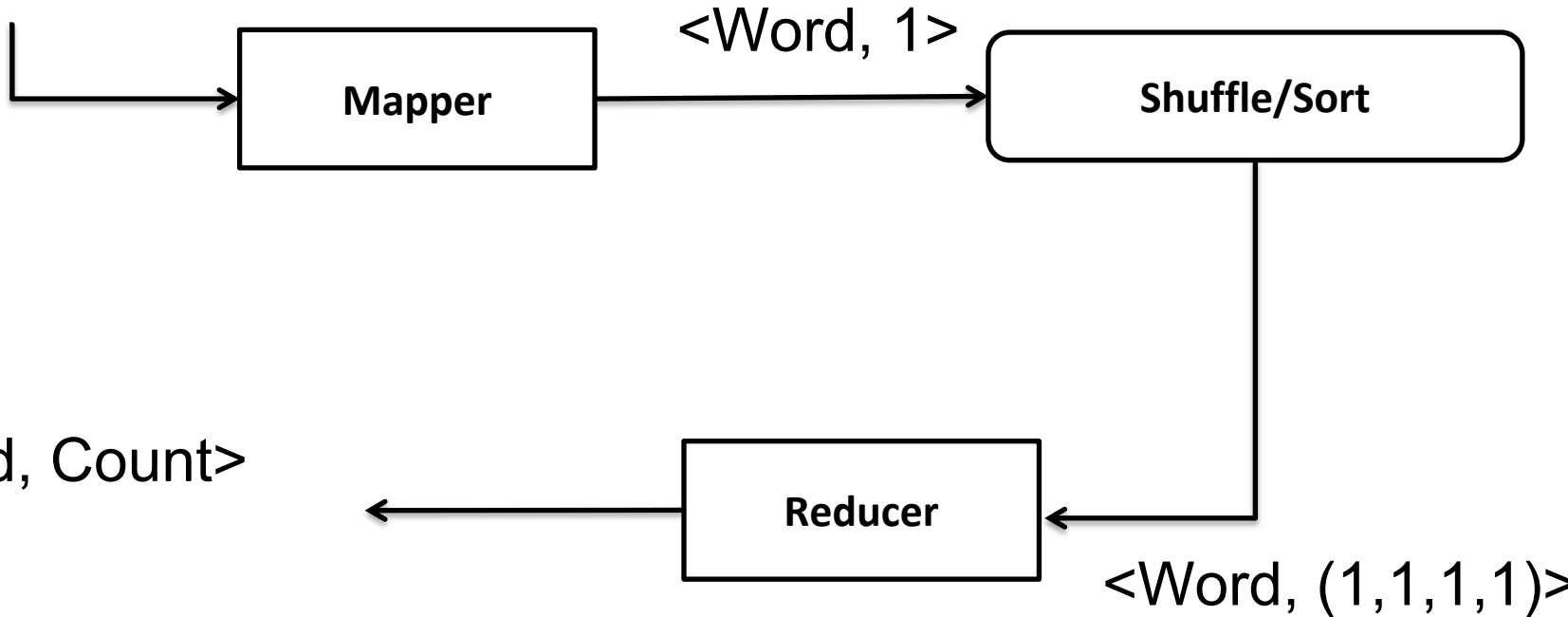
MapReduce Word Count Example





WordCount with MapReduce

<Doc-ID, Content>



MapReduce Example – Word Count

- Count the number of occurrences of each word in a large amount of input data

```
map(key, value):  
    for word in value.split():  
        emit(word,1)
```

```
reduce(key, values):  
    count = 0  
    for value in values:  
        count += value  
    emit(key, count)
```

MapReduce Example – Map Phase

Input to Mapper

- Ignoring the key

```
(8675, 'I will not eat  
green eggs and ham')
```

```
(8709, 'I will not eat  
them Sam I am')
```

Output from Mapper

- No attempt is made to optimize within a record in this example

```
('I', 1), ('will', 1),  
( 'not', 1), ('eat', 1),  
( 'green', 1),  
( 'eggs', 1), ('and', 1),  
( 'ham', 1), ('I', 1),  
( 'will', 1),  
( 'not', 1), ('eat', 1),  
( 'them', 1), ('Sam', 1),  
( 'I', 1), ('am', 1)
```

MapReduce Example – Reduce Phase

Input to Reducer

- Keys are sorted
- Values for same key are in a single list
- Shuffle & Sort did this for us

```
('I', [1, 1, 1])  
(`Sam`, [1])  
(`am`, [1])  
(`and`, [1])  
(`eat`, [1, 1])  
(`eggs`, [1])  
(`green`, [1])  
(`ham`, [1])  
(`not`, [1, 1])  
(`them`, [1])  
(`will`, [1, 1])
```

Output from Reducer

- All done!

```
('I', 3)  
(`Sam`, 1)  
(`am`, 1)  
(`and`, 1)  
(`eat`, 2)  
(`eggs`, 1)  
(`green`, 1)  
(`ham`, 1)  
(`not`, 2)  
(`them`, 1)  
(`will`, 2)
```

Hadoop MapReduce

Advantages

- Scales to petabytes
- Fault tolerant
- Easy programming model
- Usable for unstructured, semi-structured and structured data
- Data locality (computation where the data is). Minimize movement of the data (only to an extend possible)
- Uses commodity hardware
- Open Source. No costs for expensive commercial parallel databases

Disadvantages

- Very limiting programming model
- After each step storage to disk
- Still a lot of data movements needed over the network (despite data locality)
- Slow response time
- Mainly for Batch Processing
- No Indexing (except of the keys). Always full table scan
- Joins of data are slow
- A MapReduce job can only start when all tasks in the preceding jobs have completed
- Data is stored in verbose formats – expensive parsing
- Immutable Data (see HDFS)