Data Mining and Analysis

MapReduce + Spark: 1

CSCE 676 :: Fall 2019
Texas A&M University
Department of Computer Science & Engineering
Prof. James Caverlee

Resources

- MMDS Chapter 2.1, 2.2, 2.4.2, 2.4.3, 2.4.5
- The Google File System by Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung
- MapReduce: Simplified Data Processing on Large Clusters by Jeffrey Dean and Sanjay Ghemawat
- Pregel: A System for Large-Scale Graph Processing
- AWS Guide

The problem of Scalability…

The Three Vs

Disclaimer: Cloudera slides are used with permission from Cloudera Inc.

Velocity

- **We are generating data faster than ever**
  - Processes are increasingly automated
  - Systems are increasingly interconnected
  - People are increasingly interacting online

© Copyright 2010-2014 Cloudera. All rights reserved. Not to be reproduced without prior written consent.

Variety

- **We are producing a wide variety of data**
  - Social network connections
  - Server and application log files
  - Electronic medical records
  - Images, audio, and video
  - RFID and wireless sensor network events
  - Product ratings on shopping and review Web sites
  - And much more…
- **Not all of this maps cleanly to the relational model**

© Copyright 2010-2014 Cloudera. All rights reserved. Not to be reproduced without prior written consent.

Volume

- **Every day…**
  - More than 1.5 billion shares are traded on the New York Stock Exchange
  - Facebook stores 2.7 billion comments and 'Likes'
  - Google processes about 24 petabytes of data
- **Every minute…**
  - Foursquare handles more than 2,000 check-ins
  - TransUnion makes nearly 70,000 updates to credit files
- **And every second…**
  - Banks process more than 10,000 credit card transactions

© Copyright 2010-2014 Cloudera. All rights reserved. Not to be reproduced without prior written consent.

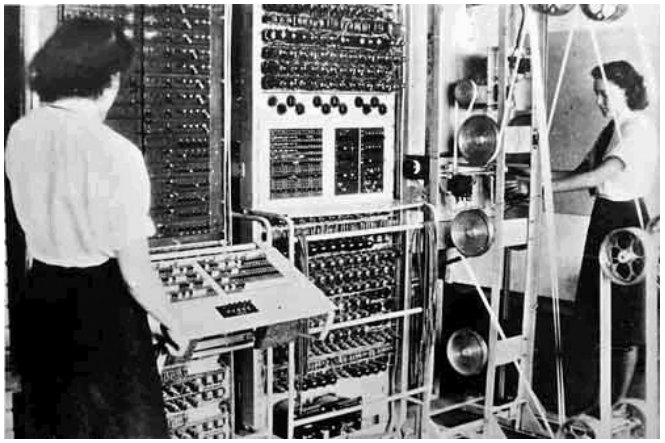
We Need a System that Scales

- **We're generating too much data to process with traditional tools**
- **Two key problems to address**
  - How can we reliably store large amounts of data at a reasonable cost?
  - How can we analyze all the data we have stored?

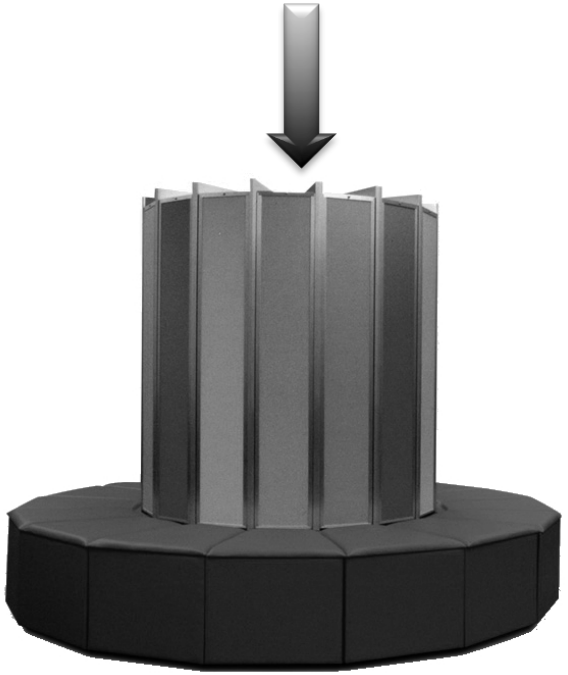
© Copyright 2010-2014 Cloudera. All rights reserved. Not to be reproduced without prior written consent.

Traditional Large-Scale Computation

- **Traditionally, computation has been processor-bound**
  - Relatively small amounts of data
  - Lots of complex processing



- **The early solution: bigger computers**
  - Faster processor, more memory
  - But even this couldn't keep up



cloudera®

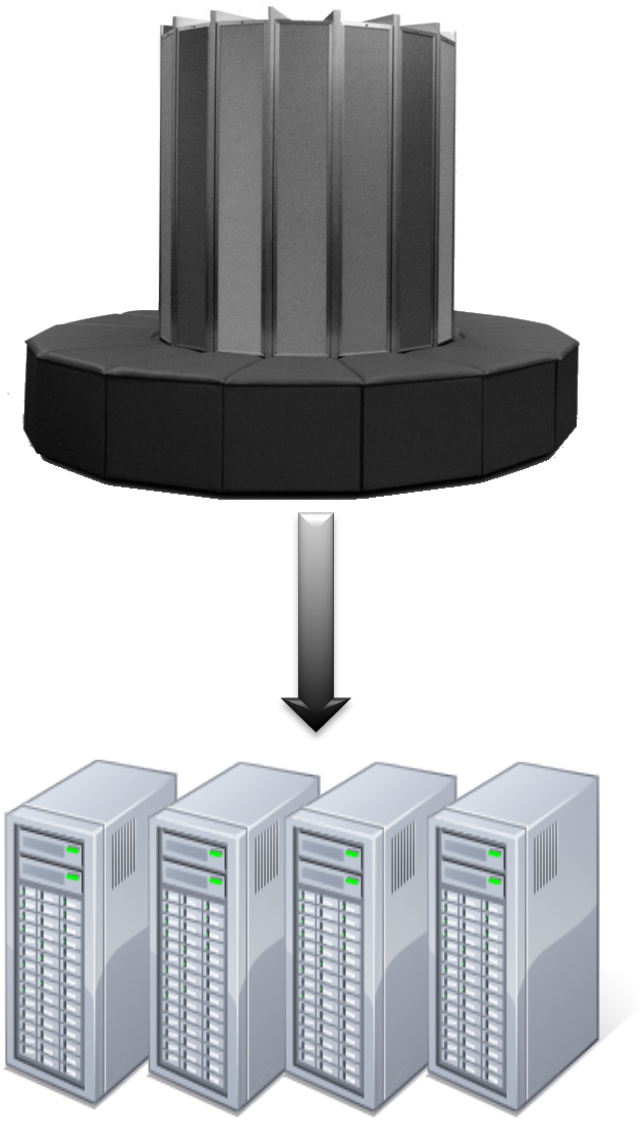
© Copyright 2010-2015 Cloudera. All rights reserved. Not to be reproduced without prior written consent.

Distributed Systems

- **The better solution: more computers**
  - Distributed systems – use multiple machines for a single job

"In pioneer days they used oxen for heavy pulling, and when one ox couldn't budge a log, we didn't try to grow a larger ox. We shouldn't be trying for bigger computers, but for *more systems* of computers."

– Grace Hopper

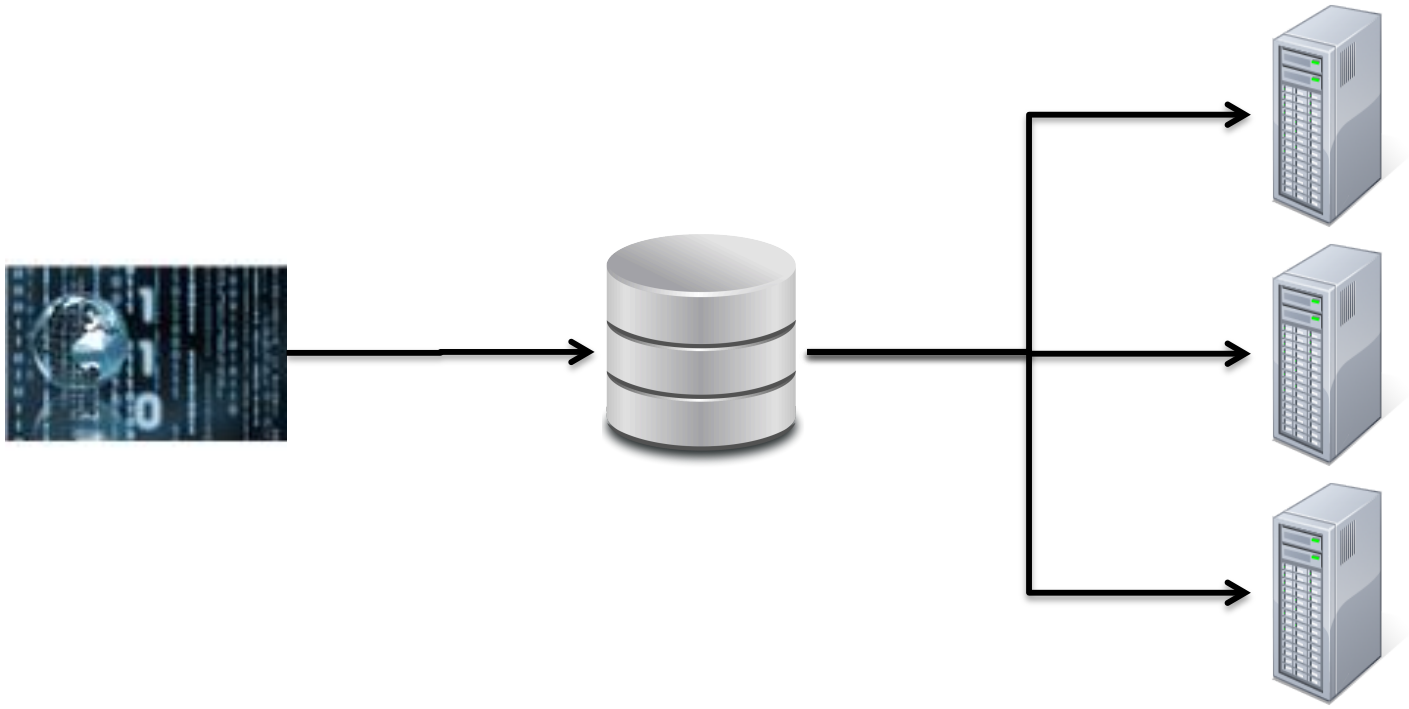


cloudera®

© Copyright 2010-2015 Cloudera. All rights reserved. Not to be reproduced without prior written consent.

Distributed Systems: The Data Bottleneck

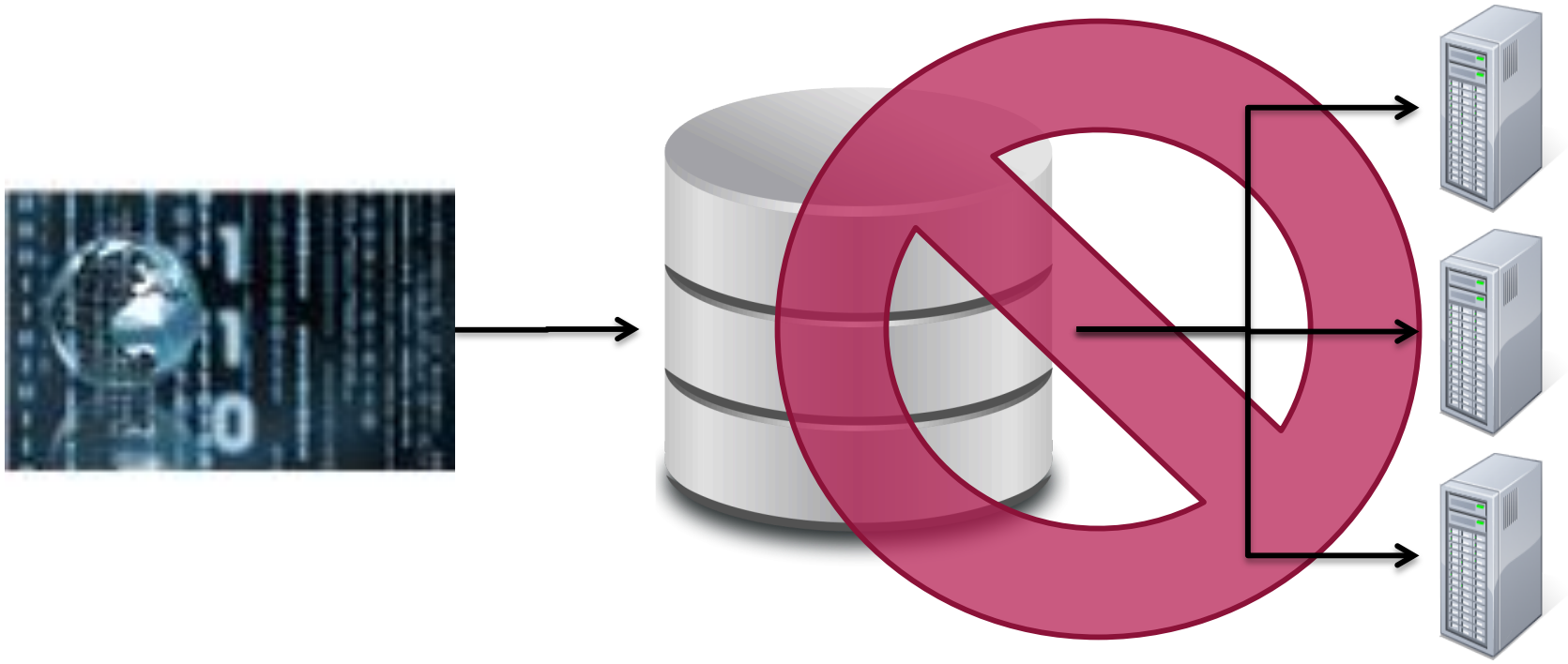
- **Traditionally, data is stored in a central location**
- **Data is copied to processors at runtime**
- **Fine for limited amounts of data**



© Copyright 2010-2015 Cloudera. All rights reserved. Not to be reproduced without prior written consent.

Distributed Systems: The Data Bottleneck (cont'd)

- **Modern systems have much more data**
  - terabytes+ a day
  - petabytes+ total
- **We need a new approach…**



© Copyright 2010-2015 Cloudera. All rights reserved. Not to be reproduced without prior written consent.

Introducing:
Apache Hadoop

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.

Introducing:
Apache Hadoop

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.

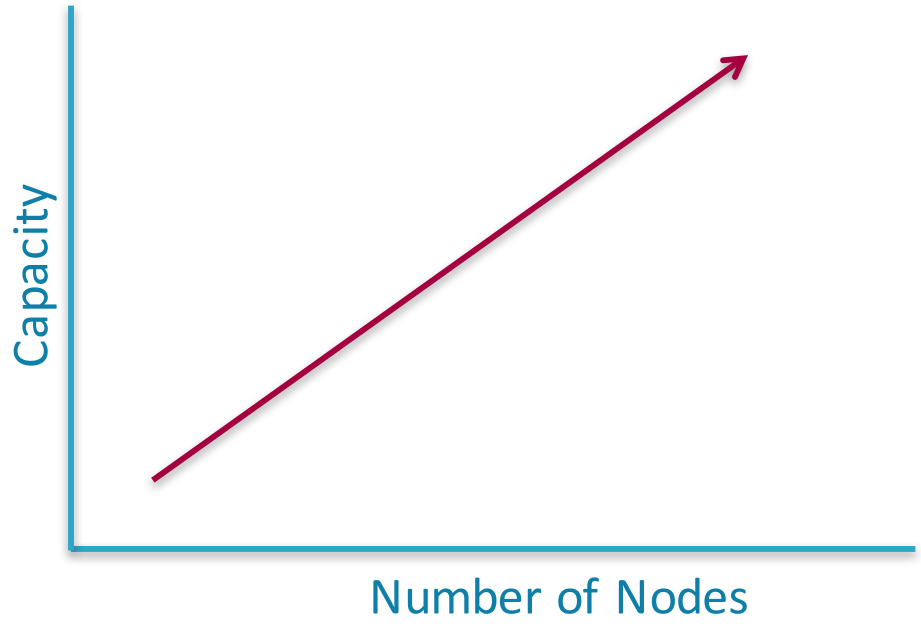
Core Hadoop is a File System and a Processing Framework

- **The Hadoop Distributed File System (HDFS)**
  - Any type of file can be stored in HDFS
  - Data is split into chunks and replicated as it is written
    - Provides resiliency and high availability
    - Handled automatically by Hadoop
- **YARN (Yet Another Resource Negotiator)**
  - Manages the processing resources of the Hadoop cluster
  - Schedules jobs
  - Runs processing frameworks
- **MapReduce**
  - A distributed processing framework

© Copyright 2010-2016 Cloudera. All rights reserved. Not to be reproduced or shared without prior written consent from Cloudera.

Hadoop is Scalable

- **Adding nodes (machines) adds capacity proportionally**
- **Increasing load results in a graceful decline in performance**
  - Not failure of the system



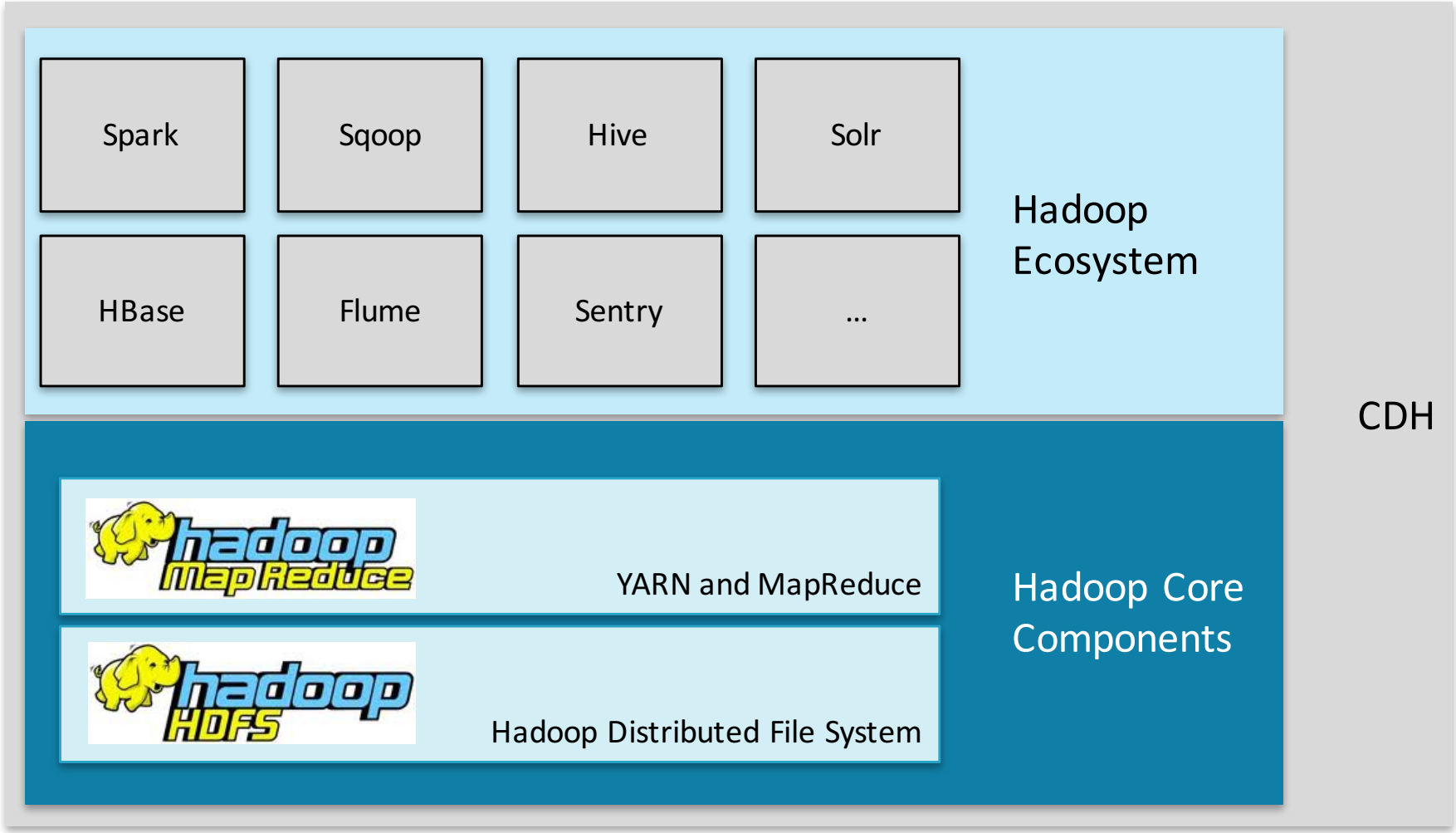
cloudera® © Copyright 2010-2016 Cloudera. All rights reserved. Not to be reproduced or shared without prior written consent from Cloudera.

Hadoop is Fault Tolerant

- **Node failure is inevitable**
- **What happens?**
  - System continues to function
  - Master reassigns work to a different node
  - Data replication means there is no loss of data
  - Nodes which recover rejoin the cluster automatically

© Copyright 2010-2016 Cloudera. All rights reserved. Not to be reproduced or shared without prior written consent from Cloudera.

Hadoop Components



cloudera © Copyright 2010-2016 Cloudera. All rights reserved. Not to be reproduced or shared without prior written consent from Cloudera.

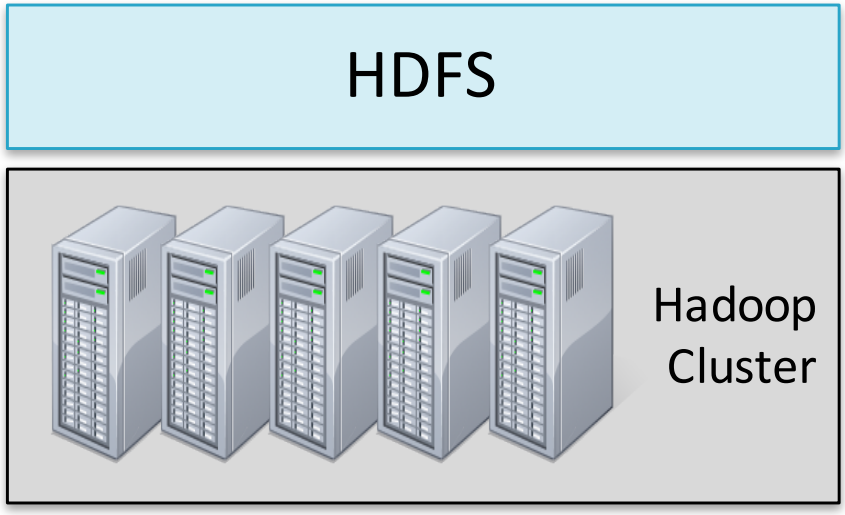
The Hadoop Distributed File System (HDFS)

- **HDFS is the storage layer for Hadoop**
- **A filesystem which can store any type of data**
- **Provides inexpensive and reliable storage for massive amounts of data**
  - Data is replicated across computers
- **HDFS performs best with a "modest" number of large files**
  - Millions, rather than billions, of files
  - Each file typically 100MB or more
- **File in HDFS are "write once"**
  - Appends are permitted
  - No random writes are allowed

© Copyright 2010-2016 Cloudera. All rights reserved. Not to be reproduced or shared without prior written consent from Cloudera.

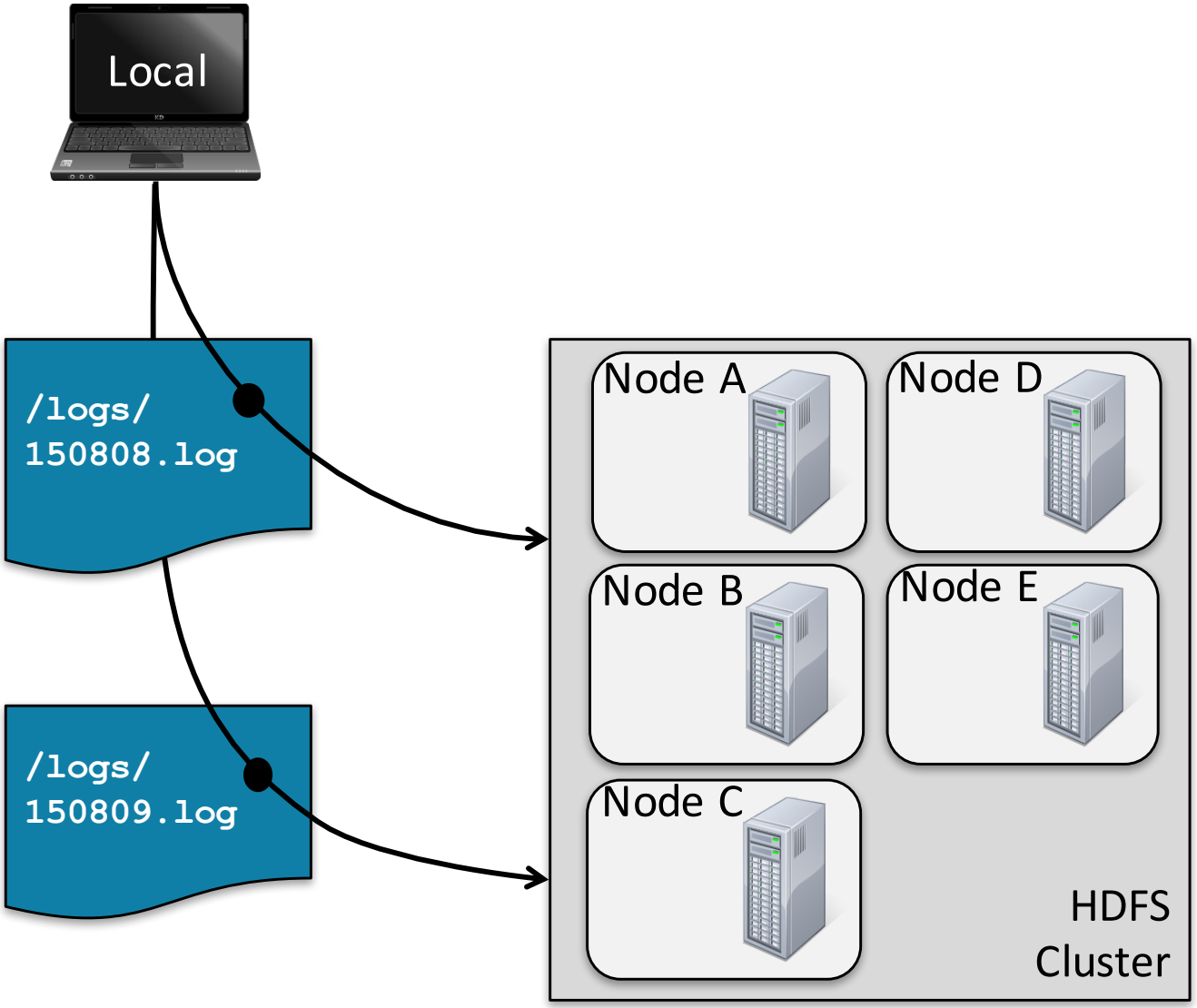
HDFS Basic Concepts

- **HDFS is a filesystem written in Java**
- **Sits on top of a native filesystem**
- **Scalable**
- **Fault tolerant**
- **Supports efficient processing with MapReduce, Spark, and other frameworks**



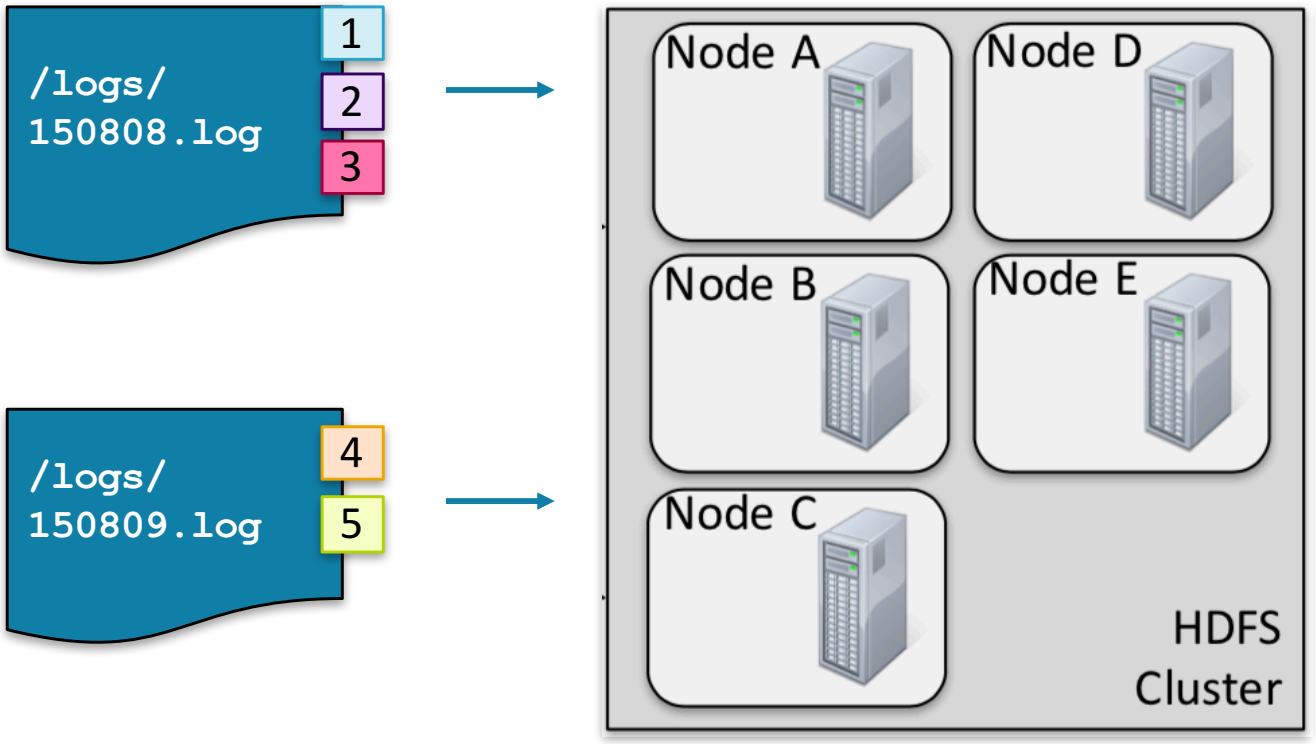
cloudera® © Copyright 2010-2016 Cloudera. All rights reserved. Not to be reproduced or shared without prior written consent from Cloudera.

Example: Storing and Retrieving Files (1)



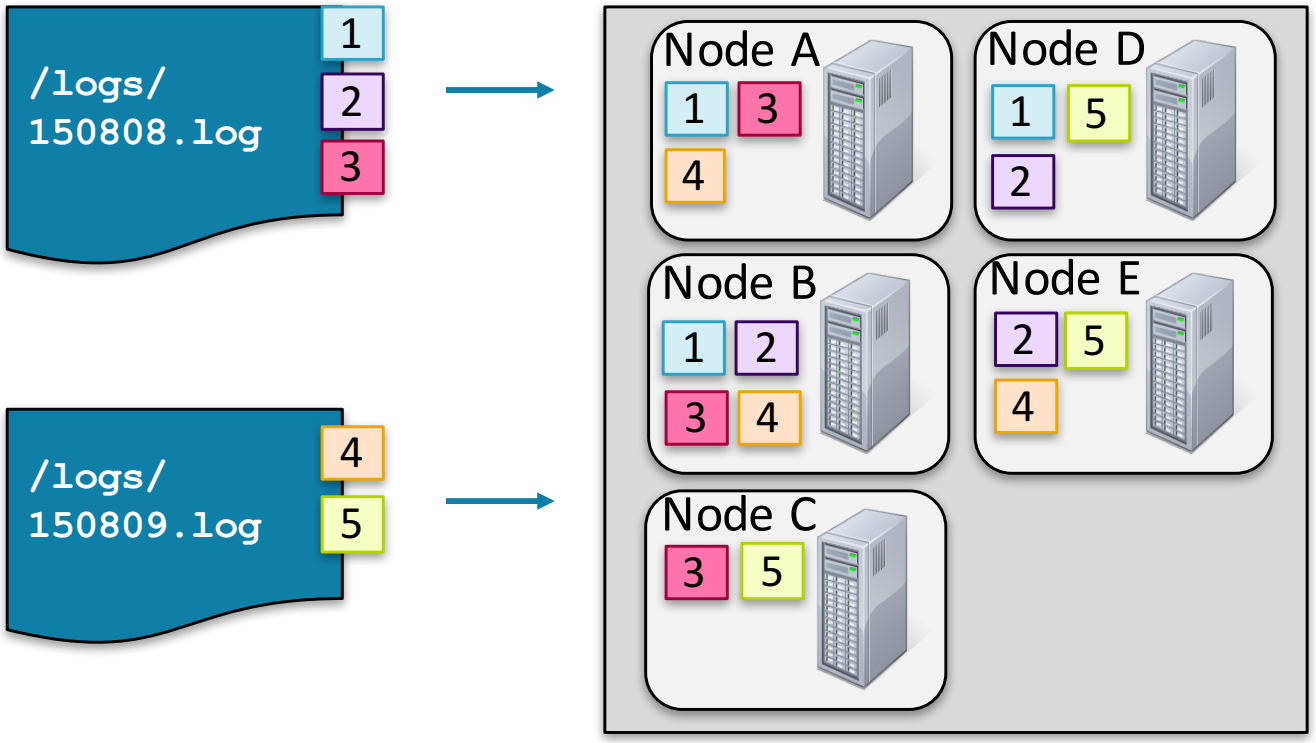
cloudera® © Copyright 2010-2016 Cloudera. All rights reserved. Not to be reproduced or shared without prior written consent from Cloudera.

Example: Storing and Retrieving Files (2)



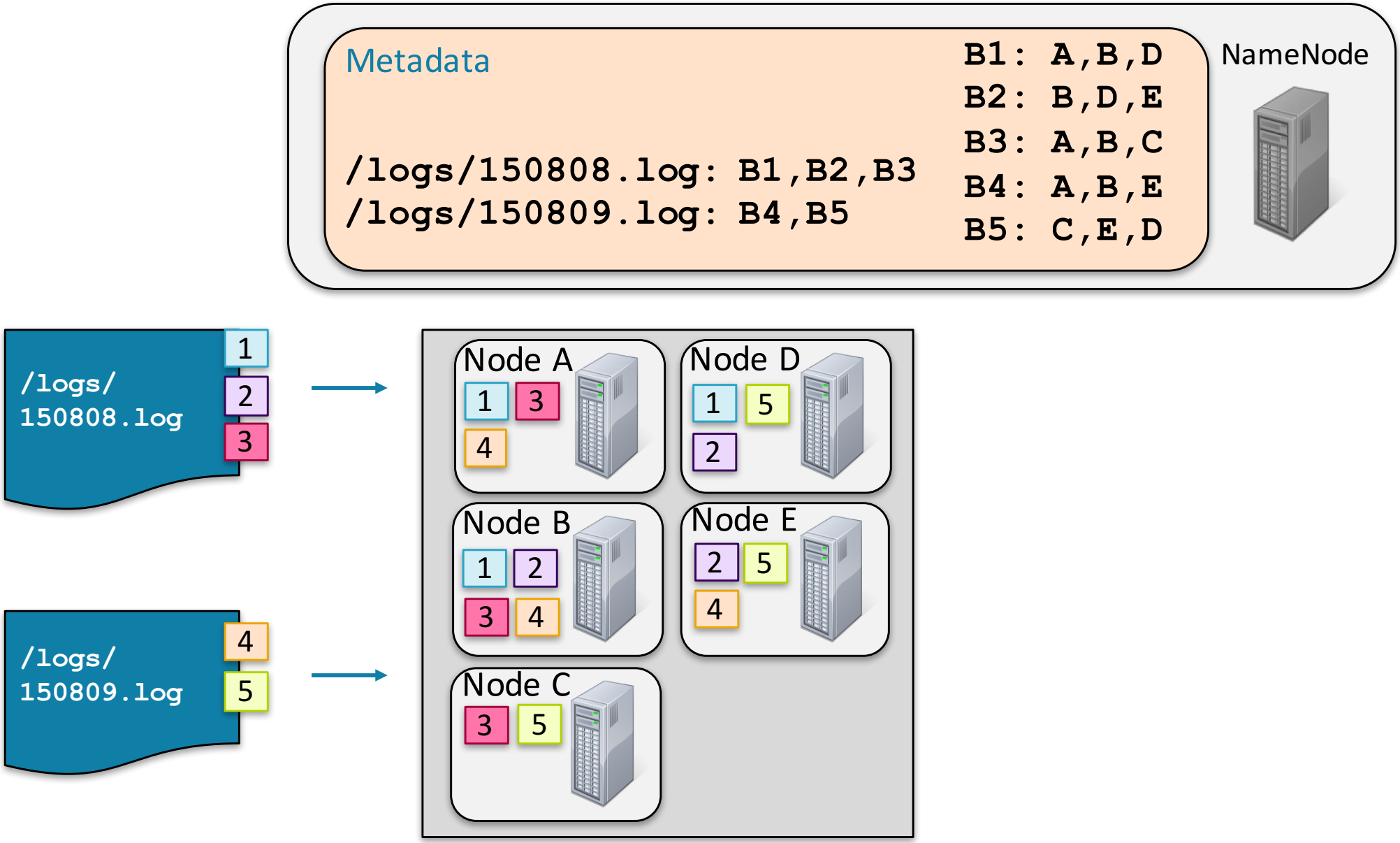
cloudera © Copyright 2010-2016 Cloudera. All rights reserved. Not to be reproduced or shared without prior written consent from Cloudera.

Example: Storing and Retrieving Files (2)



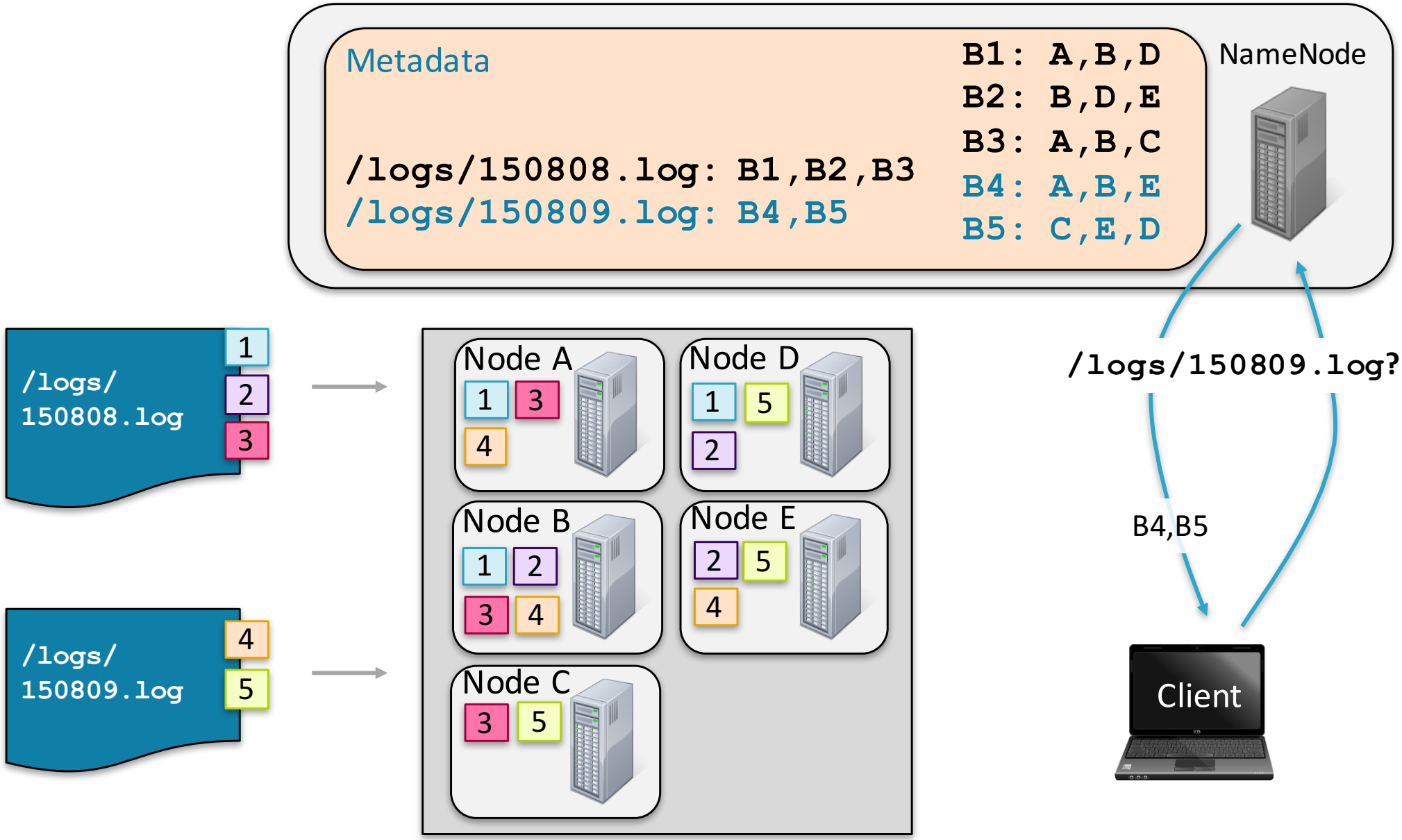
cloudera® © Copyright 2010-2016 Cloudera. All rights reserved. Not to be reproduced or shared without prior written consent from Cloudera.

Example: Storing and Retrieving Files (2)



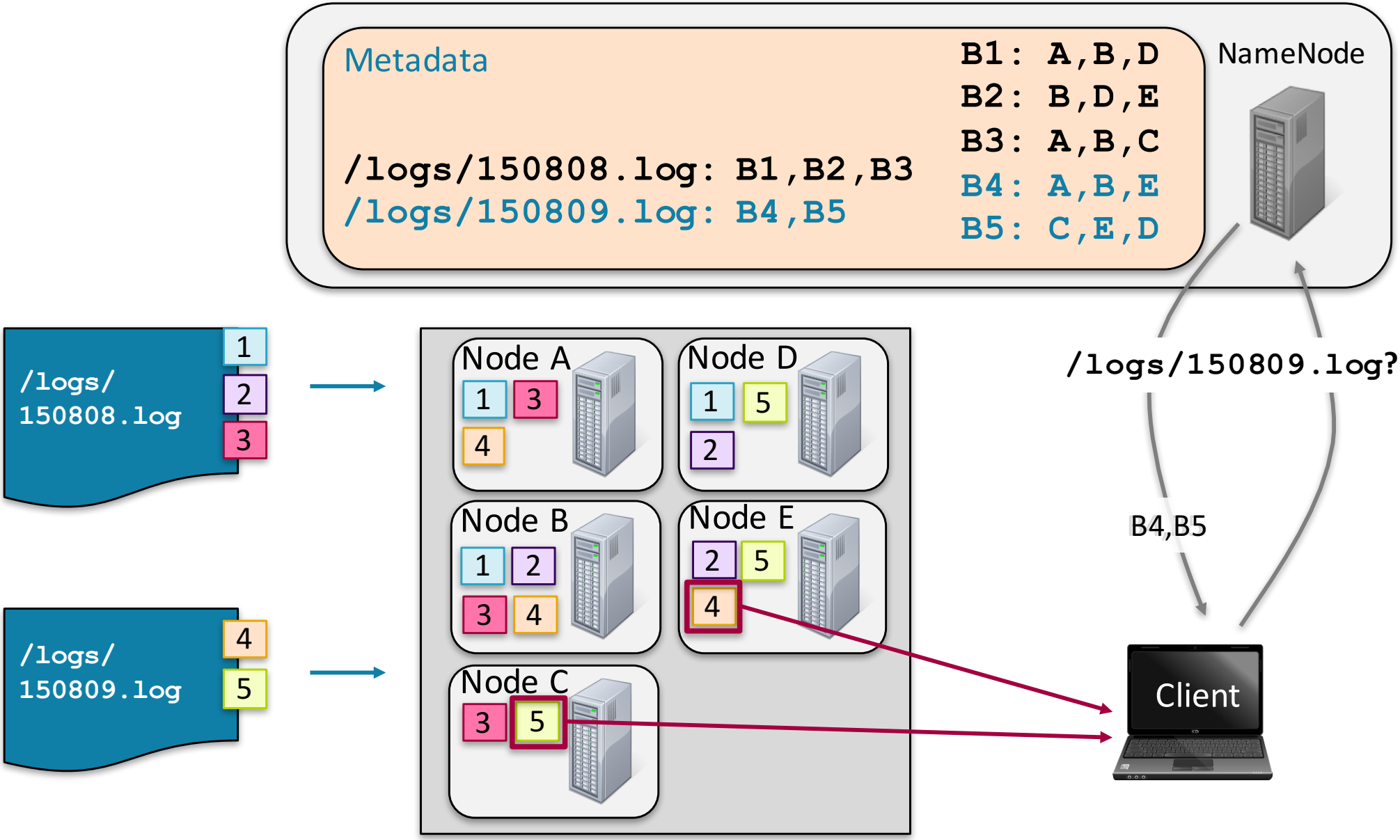
cloudera © Copyright 2010-2016 Cloudera. All rights reserved. Not to be reproduced or shared without prior written consent from Cloudera.

Example: Storing and Retrieving Files (3)



cloudera © Copyright 2010-2016 Cloudera. All rights reserved. Not to be reproduced or shared without prior written consent from Cloudera.

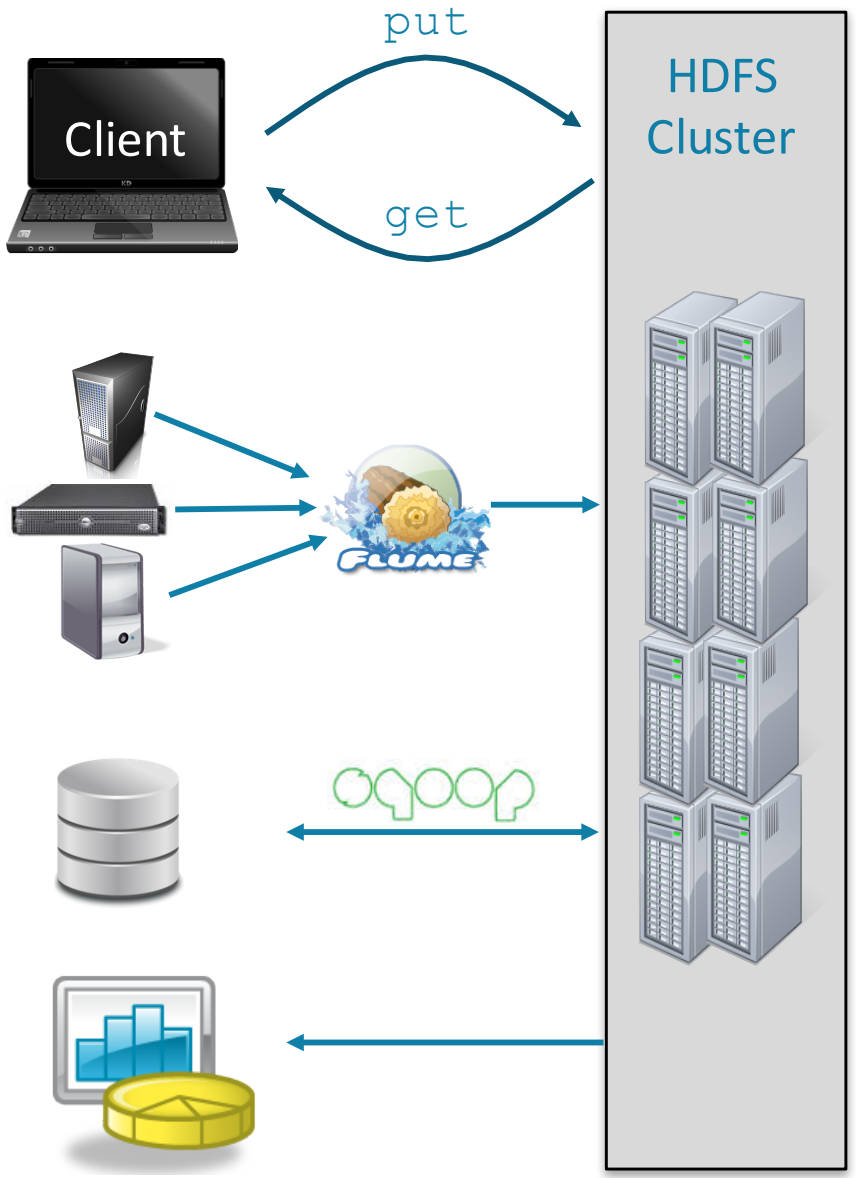
Example: Storing and Retrieving Files (4)



© Copyright 2010-2016 Cloudera. All rights reserved. Not to be reproduced or shared without prior written consent from Cloudera.

Getting Data In and Out of HDFS

- **Hadoop**
  - Copies data between client (local) and HDFS (cluster)
  - API or command line
- **Ecosystem Projects**
  - Flume
    - Collects data from network sources (such as, websites, system logs)
  - Sqoop
    - Transfers data between HDFS and RDBMSs
- **Business Intelligence Tools**



RDBMS: Relational Database Management System

cloudera © Copyright 2010-2016 Cloudera. All rights reserved. Not to be reproduced or shared without prior written consent from Cloudera.

HDFS Quick Demo

MapReduce: Key Features

- **MapReduce is a programming model**
  - Neither platform- nor language-specific
  - Record-oriented data processing (key and value)
  - Facilitates task distribution across multiple nodes
- **MapReduce was the original processing framework available on Hadoop**
  - Still widely used, although other frameworks are replacing it for many types of workload
- **MapReduce code is typically written in Java**

© Copyright 2010-2016 Cloudera. All rights reserved. Not to be reproduced or shared without prior written consent from Cloudera.

Programming Model

- **MapReduce is a style of programming designed for:**
  1. Easy parallel programming
  2. Invisible management of hardware and software failures
  3. Easy management of very-large-scale data

- It has several implementations, including Hadoop, Spark (used in this class), Flink, and the original Google implementation just called "MapReduce"

MapReduce: Overview

3 steps of MapReduce

- **Map:**
  - Apply a user-written *Map function* to each input element
    - *Mapper* applies the Map function to a single element
      - Many mappers grouped in a *Map task* (the unit of parallelism)
  - The output of the Map function is a set of 0, 1, or more *key-value pairs*.
- **Group by key:** Sort and shuffle
  - System sorts all the key-value pairs by key, and outputs key-(list of values) pairs
- **Reduce:**
  - User-written *Reduce function* is applied to each key-(list of values)

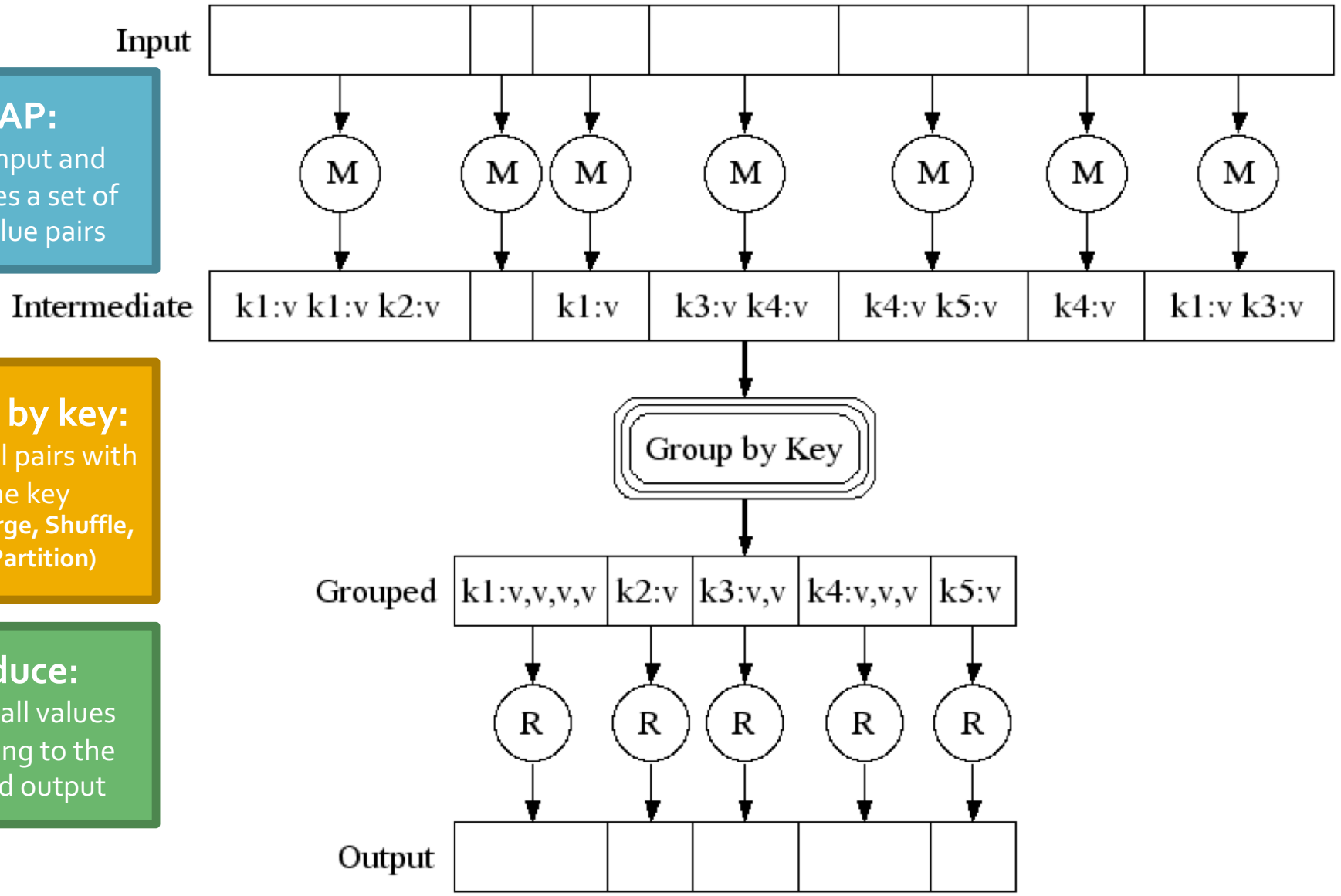
Outline stays the same, **Map** and **Reduce** change to fit the problem

Map-Reduce: A diagram

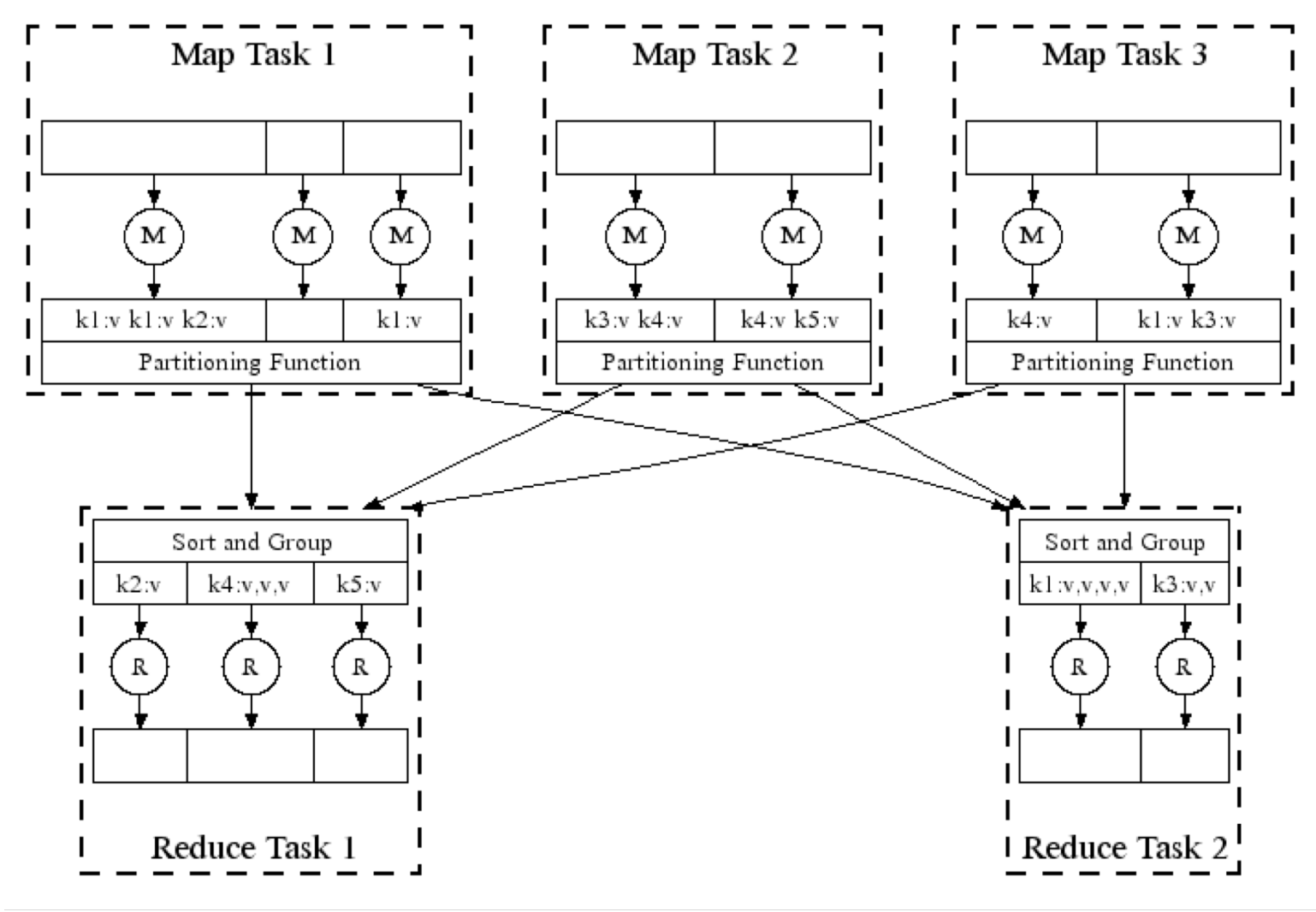
MAP:
Read input and produces a set of key-value pairs

Group by key:
Collect all pairs with same key
(Hash merge, Shuffle, Sort, Partition)

Reduce:
Collect all values belonging to the key and output

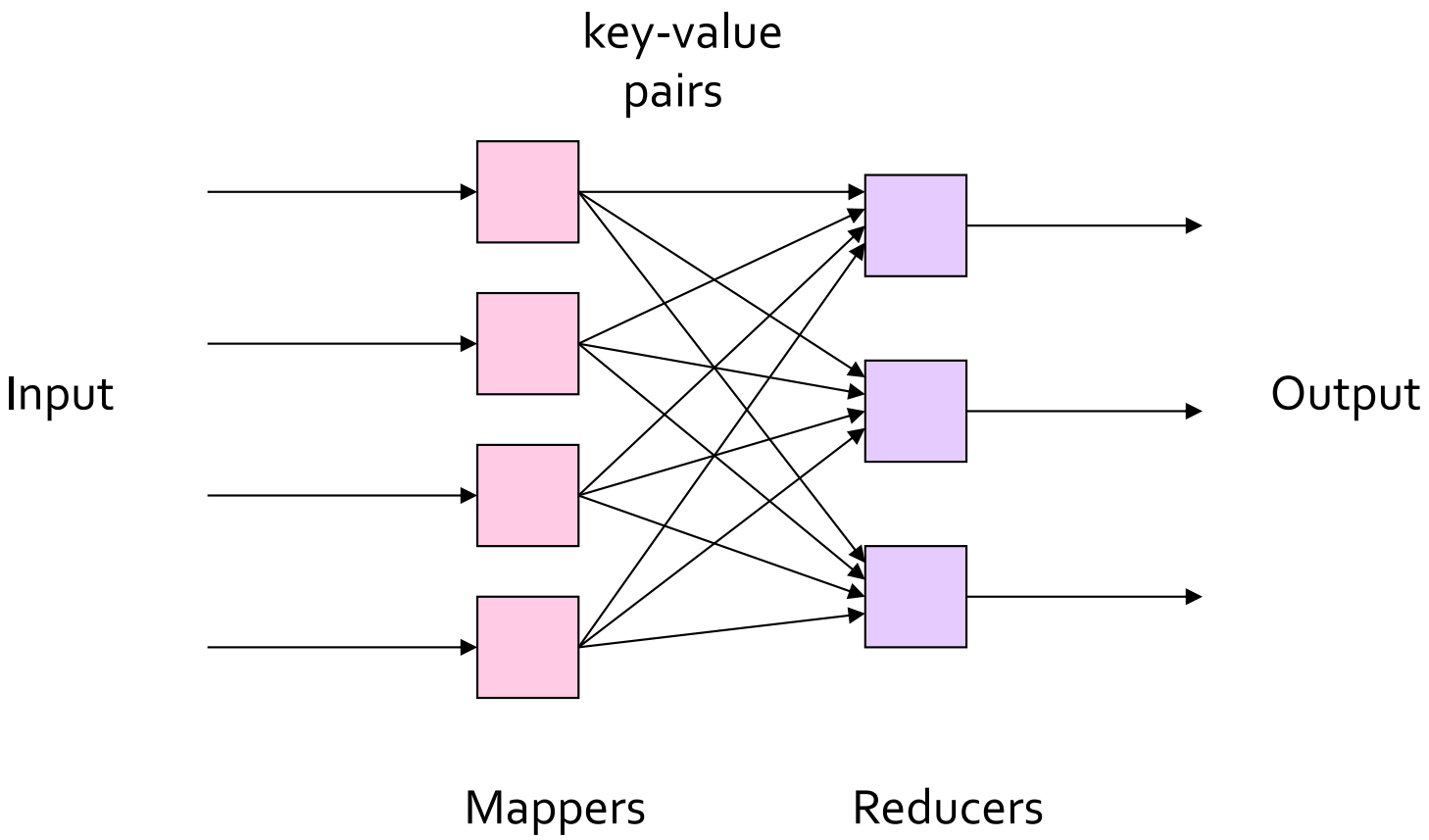


Map-Reduce: In Parallel



All phases are distributed with many tasks doing the work

MapReduce Pattern



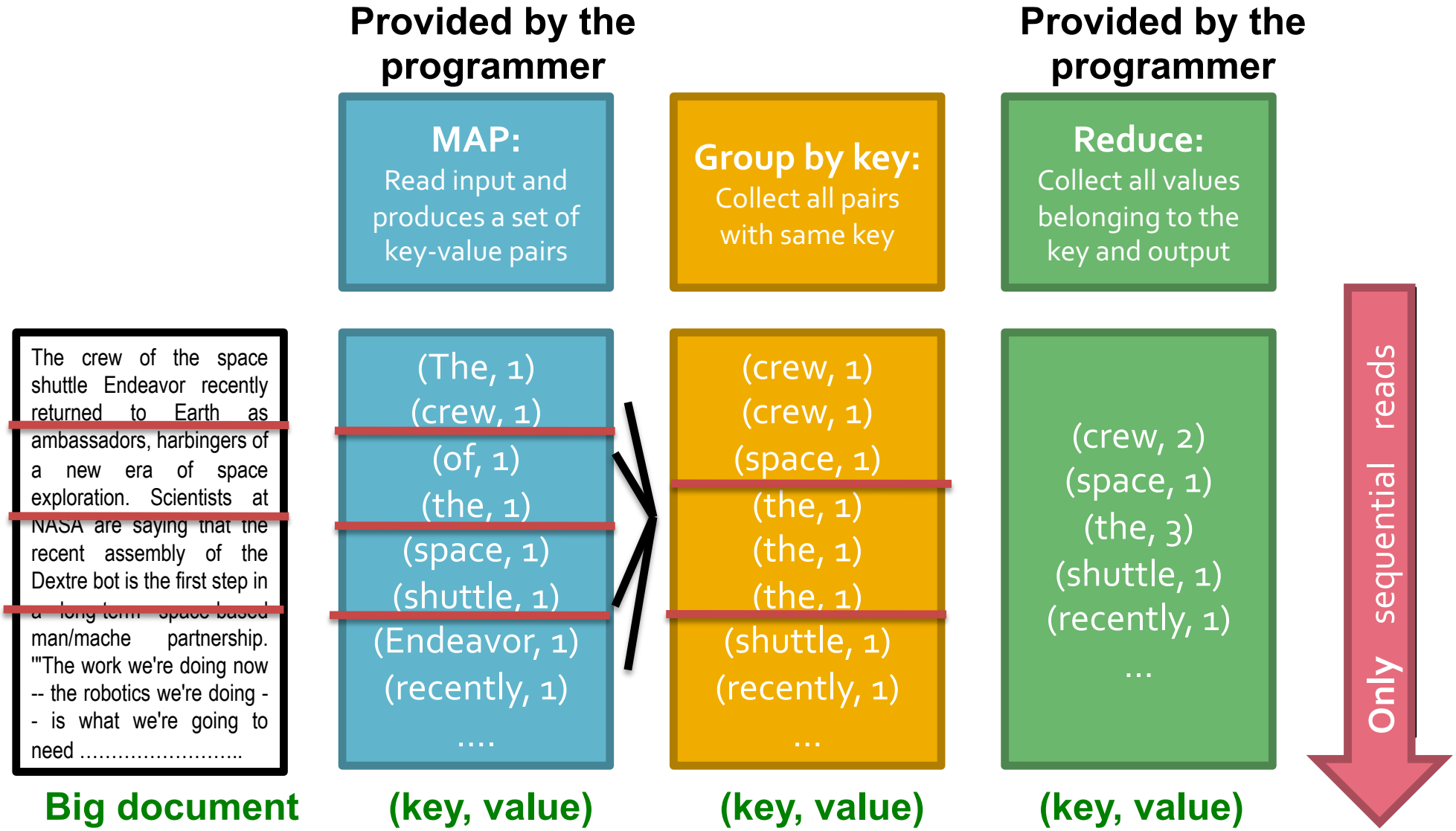
Example: Word Counting

Example MapReduce task:

- We have a huge text document
- Count the number of times each distinct word appears in the file

- **Many applications of this:**
  - Analyze web server logs to find popular URLs
  - Statistical machine translation:
    - Need to count number of times every 5-word sequence occurs in a large corpus of documents

MapReduce: Word Counting



Word Count Using MapReduce

```
map(key, value):
# key: document name; value: text of the document
   for each word w in value:
      emit(w, 1)
```

```
reduce(key, values):
# key: a word; value: an iterator over counts
      result = 0
      for each count v in values:
            result += v
      emit(key, result)
```

MapReduce: Environment

MapReduce environment takes care of:

- **Partitioning** the input data
- **Scheduling** the program's execution across a set of machines
- Performing the **group by key** step
  - In practice this is is the bottleneck
- Handling machine **failures**
- Managing required inter-machine **communication**

Dealing with Failures

- **Map worker failure**
  - Map tasks completed or in-progress at worker are reset to idle and rescheduled
  - Reduce workers are notified when map task is rescheduled on another worker
- **Reduce worker failure**
  - Only in-progress tasks are reset to idle and the reduce task is restarted

MapReduce Quick Demo

DIY: mapper

```
"""mapper.py"""

import sys

# input comes from STDIN (standard input)
for line in sys.stdin:
    # split the line into words
    words = line.split()
    # increase counters
    for word in words:
        # write the results to STDOUT (standard output);
        # what we output here will be the input for the
        # Reduce step, i.e. the input for reducer.py
        #
        # tab-delimited; the trivial word count is 1
        print '%s\t%s' % (word, 1)
```

DIY: reducer

```
"""reducer.py"""

from operator import itemgetter
import sys

current_word = None
current_count = 0
word = None

# input comes from STDIN
for line in sys.stdin:

    # parse the input we got from mapper.py
    word, count = line.split('\t', 1)
    count = int(count)
    except ValueError:

    # this IF-switch only works because Hadoop sorts map output
    # by key (here: word) before it is passed to the reducer
    if current_word == word:
        current_count += count
    else:
        if current_word:
            # write result to STDOUT
            print '%s\t%s' % (current_word, current_count)
        current_count = count
        current_word = word
```

DIY: mapreduce framework

doc.txt:

The Longhorns? The Longhorns? No … Aggies! Aggies! Aggies! Aggies!

```
$ cat doc.txt | python mapper.py | sort -k1 | python reducer.py
...   1
Aggies!  4
Longhorns?  2
No 1
The   2
```

In Hadoop Streaming:

```
hadoop jar hadoop-streaming.jar \
   -files mapper.py,reducer.py \
   -input csce676/doc.txt \
   -output myresult \
   -mapper mapper.py \
   -reducer reducer.py
```

Problems with MapReduce

- **Two major limitations of MapReduce:**
  - Difficulty of programming directly in MR
    - Many problems aren't easily described as map-reduce
  - Performance bottlenecks, or batch not fitting the use cases
    - Persistence to disk typically slower than in-memory work
- **In short, MR doesn't compose well for large applications**
  - Many times one needs to chain multiple map-reduce steps

Data-Flow Systems

- **MapReduce uses two "ranks" of tasks:**
  One for Map the second for Reduce
  - Data flows from the first rank to the second

- **Data-Flow Systems generalize this in two ways:**
  1. Allow any number of tasks/ranks
  2. Allow functions other than Map and Reduce
  - As long as data flow is in one direction only, we can have the blocking property and allow recovery of tasks rather than whole jobs

Spark: Most Popular Data-Flow System

- **Expressive computing system, not limited to the map-reduce model**
- **Additions to MapReduce model:**
  - Fast data sharing
    - Avoids saving intermediate results to disk
    - Caches data for repetitive queries (e.g. for machine learning)
  - General execution graphs (DAGs)
  - Richer functions than just map and reduce
- Compatible with Hadoop

Spark: Overview

- Open source software (Apache Foundation)
- Supports **Java, Scala and Python**

- **Key construct/idea:** Resilient Distributed Dataset (RDD)

- **Higher-level APIs:** DataFrames & DataSets
  - Introduced in more recent versions of Spark
  - Different APIs for aggregate data, which allowed to introduce SQL support

Spark: RDD

Key concept *Resilient Distributed Dataset* (RDD)

- Partitioned collection of records
  - Generalizes (key-value) pairs
- Spread across the cluster, Read-only
- Caching dataset in memory
  - Different storage levels available
  - Fallback to disk possible
- RDDs can be created from Hadoop, or by transforming other RDDs (you can stack RDDs)
- RDDs are best suited for applications that apply the same operation to all elements of a dataset

Spark RDD Operations

- **Transformations** build RDDs through deterministic operations on other RDDs:
  - Transformations include *map*, *filter*, *join, union, intersection, distinct*
  - **Lazy evaluation:** Nothing computed until an action requires it
- **Actions** to return value or export data
  - Actions include *count*, *collect*, *reduce, save*
  - Actions can be applied to RDDs; actions force calculations and return values

DataFrame & Dataset

- DataFrame:
  - Unlike an RDD, data organized into named columns, e.g. a **table in a relational database**.
  - Imposes a structure onto a distributed collection of data, allowing higher-level abstraction
- Dataset:
  - Extension of DataFrame API which provides **type-safe, object-oriented programming interface** (compile-time error detection)

Both built on Spark SQL engine. both can be converted back to an RDD

Useful Libraries for Spark

- Spark SQL

- Spark Streaming – **stream processing of live datastreams**
- MLlib – **scalable machine learning**
- GraphX – **graph manipulation**
  - extends Spark RDD with Graph abstraction: a directed multigraph with properties attached to each vertex and edge

Spark vs. Hadoop MapReduce

- Performance: Spark normally faster but with caveats
  - Spark can process data in-memory; Hadoop MapReduce persists back to the disk after a map or reduce action
  - Spark generally outperforms MapReduce, but it **often needs lots of memory to perform well**; if there are other resource-demanding services or can't fit in memory, Spark degrades
  - MapReduce easily runs alongside other services with minor performance differences, & works well with the 1-pass jobs it was designed for
- Ease of use: **Spark is easier to program** (higher-level APIs)
- Data processing: **Spark more general**

Spark Quick Demo