cloudera
Ask Bigger Questions

Apache Hadoop – A course for undergraduates

Lecture 2

The Hadoop Ecosystem

Chapter 2.1

The Hadoop Ecosystem

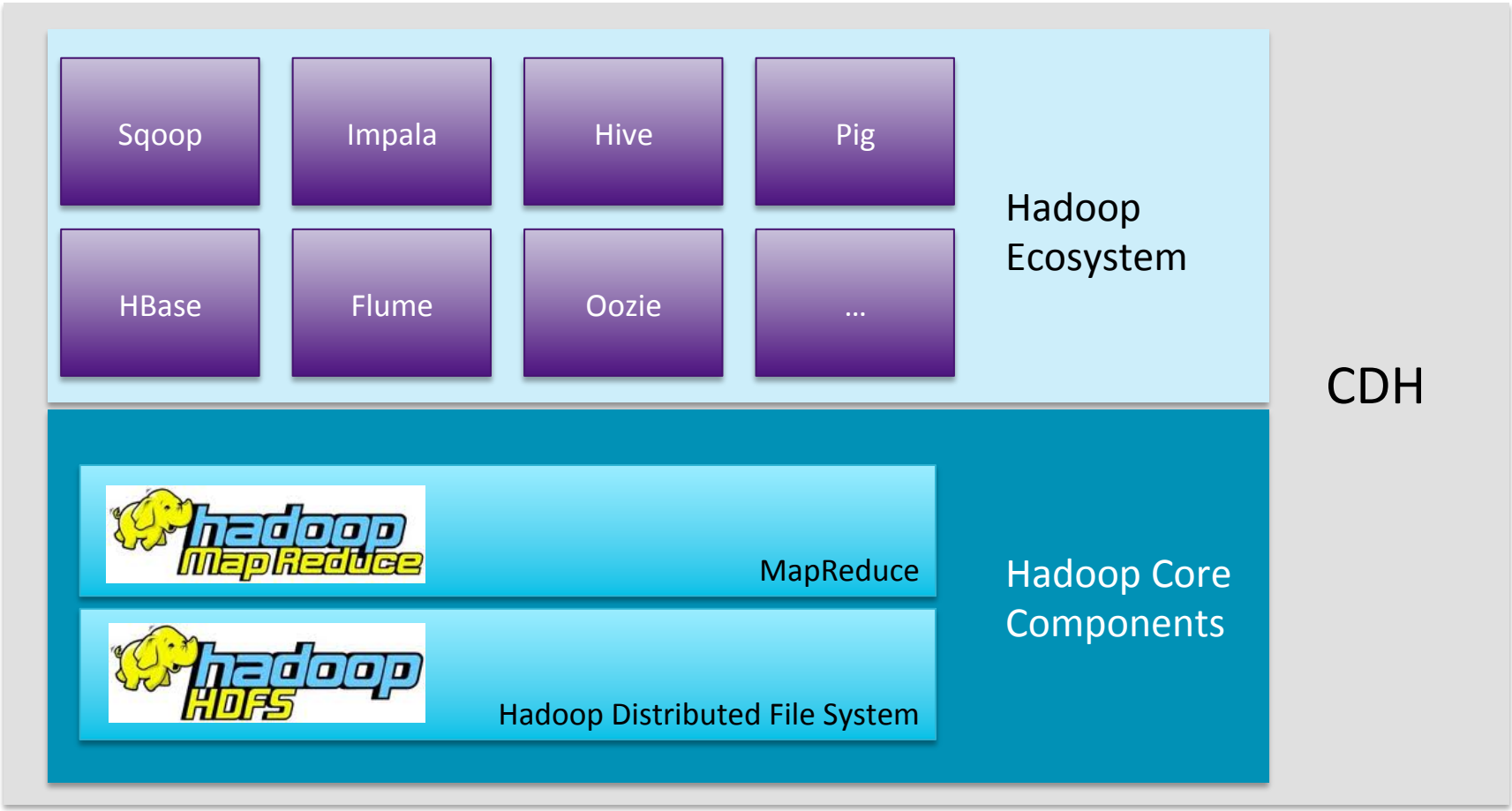
- **What other projects exist around core Hadoop?**
- **When to use HBase?**
- **How does Spark compare to MapReduce?**
- **What is the differences between Hive, Pig, and Impala?**
- **How is Flume typically deployed?**

Chapter Topics

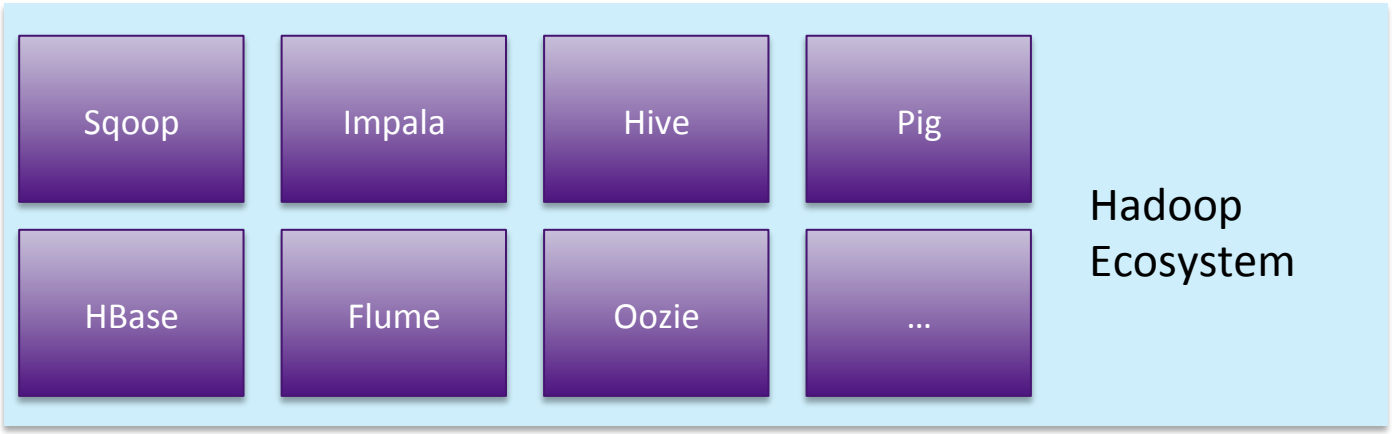
The Hadoop Ecosystem

- **Introduction**
- Data Storage: HBase
- Data Integration: Flume and Sqoop
- Data Processing: Spark
- Data Analysis: Hive, Pig, and Impala
- Workflow Engine: Oozie
- Machine Learning: Mahout

The Hadoop Ecosystem (1)



The Hadoop Ecosystem (2)



- **Next, a discussion of the key Hadoop ecosystem components**

The Hadoop Ecosystem (3)

- **Ecosystem projects may be**
  - Built on HDFS and MapReduce
  - Built on just HDFS
  - Designed to integrate with or support Hadoop
- **Most are Apache projects or Apache Incubator projects**
  - Some others are not managed by the Apache Software Foundation
    - These are often hosted on GitHub or a similar repository
- **Following is an introduction to some of the most significant projects**

Chapter Topics

The Hadoop Ecosystem

- Introduction
- **Data Storage: HBase**
- Data Integration: Flume and Sqoop
- Data Processing: Spark
- Data Analysis: Hive, Pig, and Impala
- Workflow Engine: Oozie
- Machine Learning: Mahout

HBase

- **HBase is the Hadoop database**
- **A 'NoSQL' datastore**
- **Can store massive amounts of data**
  - Petabytes+
- **High write throughput**
  - Scales to hundreds of thousands of inserts per second
- **Handles sparse data well**
  - No wasted spaces for empty columns in a row
- **Limited access model**
  - Optimized for lookup of a row by key rather than full queries
  - No transactions: single row operations only
  - Only one column (the 'row key') is indexed

HBase vs Traditional RDBMSs

	RDBMS	HBase
Data layout	Row-oriented	Column-oriented
Transactions	Yes	Single row only
Query language	SQL	get/put/scan (or use Hive or Impala)
Security	Authentication/Authorization	Kerberos
Indexes	Any column	Row-key only
Max data size	TBs	PB+
Read/write throughput (queries per second)	Thousands	Millions

When To Use HBase

- **Use plain HDFS if…**
  - You only append to your dataset (no random write)
  - You usually read the whole dataset (no random read)

- **Use HBase if…**
  - You need random write and/or read
  - You do thousands of operations per second on TB+ of data

- **Use an RDBMS if…**
  - Your data fits on one big node
  - You need full transaction support
  - You need real-time query capabilities

Chapter Topics

The Hadoop Ecosystem

- Introduction
- Data Storage: HBase
- **Data Integration: Flume and Sqoop**
- Data Processing: Spark
- Data Analysis: Hive, Pig, and Impala
- Workflow Engine: Oozie
- Machine Learning: Mahout

Flume: Real-time Data Import

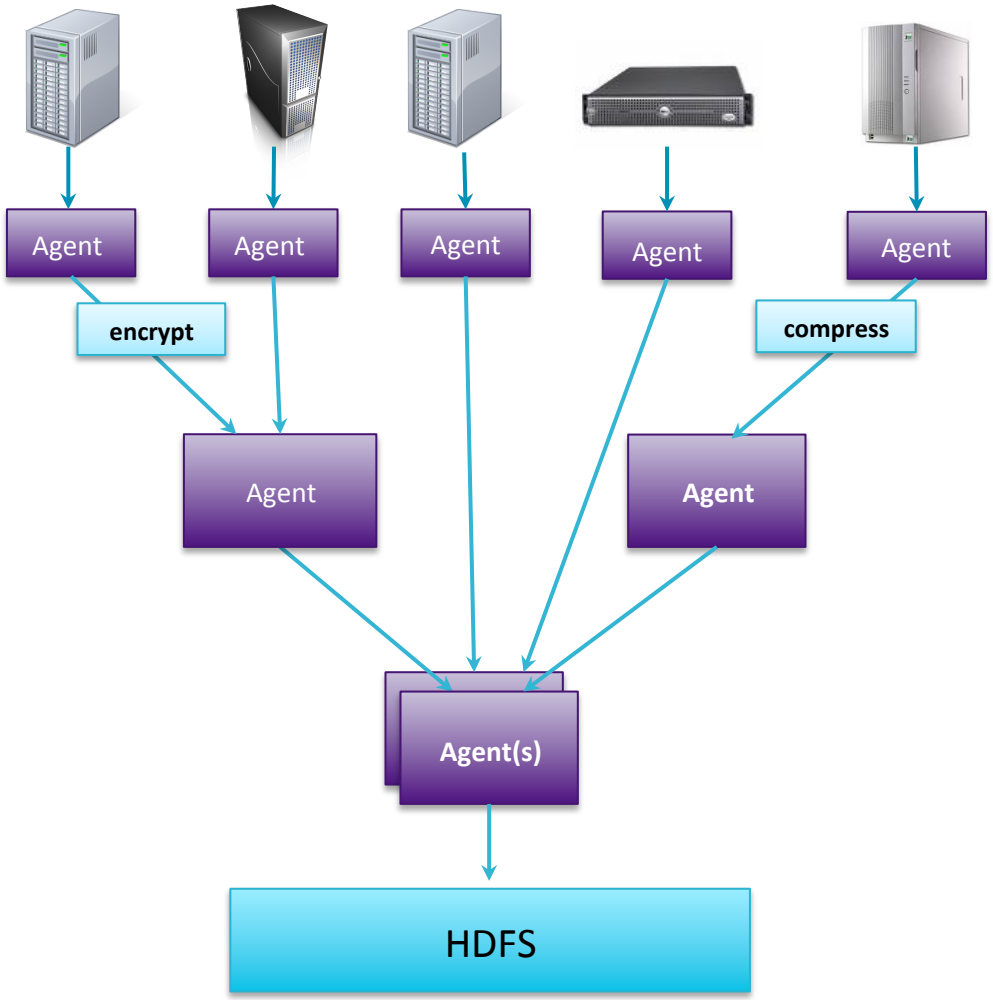
- **What is Flume?**
  - A service to move large amounts of data in real time
  - Example: storing log files in HDFS

- **Flume imports data into HDFS as it is generated**
  - Instead of batch-processing it later
  - For example, log files from a Web server

- **Flume is**
  - Distributed
  - Reliable and available
  - Horizontally scalable
  - Extensible

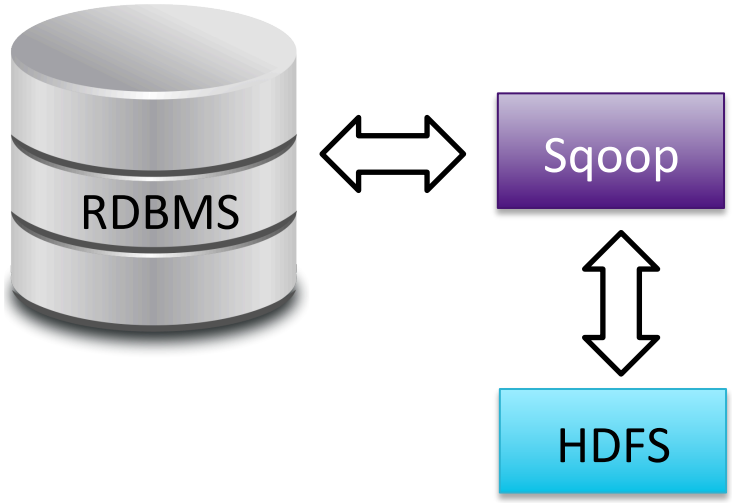
Flume: High-Level Overview

- Collect data as it is produced
  - Files, syslogs, stdout or custom source
- Process in place
  - e.g., encrypt, compress
- Pre-process data before storing
  - e.g., transform, scrub, enrich
- Write in parallel
  - Scalable throughput
- Store in any format
  - Text, compressed, binary, or custom sink



Sqoop: Exchanging Data With RDBMSs

- **Sqoop transfers data between RDBMSs and HDFS**
  - Does this very efficiently via a Map-only MapReduc
  - Supports JDBC, ODBC, and several specific databases
  - "Sqoop" = "SQL to Hadoop"



Sqoop Custom Connectors

- **Custom connectors for**
  - MySQL
  - Postgres
  - Netezza
  - Teradata
  - Oracle (partnered with Quest Software)
- **Not open source, but free to use**

Chapter Topics

The Hadoop Ecosystem

- Introduction
- Data Storage: HBase
- Data Integration: Flume and Sqoop
- **Data Processing: Spark**
- Data Analysis: Hive, Pig, and Impala
- Workflow Engine: Oozie
- Machine Learning: Mahout

Apache Spark

- **Apache Spark is a fast, general engine for large-scale data processing on a cluster**
- **Originally developed UC Berkeley's AMPLab**
- **Open source Apache project**
- **Provides several benefits over MapReduce**
  - Faster
  - Better suited for iterative algorithms
    - Can hold intermediate data in RAM, resulting in much better performance
  - Easier API
    - Supports Python, Scala, Java
  - Supports real-time streaming data processing

Spark vs Hadoop MapReduce

- **MapReduce**
  - Widely used, huge investment already made
  - Supports and supported by many complementary tools
  - Mature, well-tested
- **Spark**
  - Flexible
  - Elegant
  - Fast
  - Supports real-time streaming data processing
- **Over time, Spark is expected to supplant MapReduce as the general processing framework used by most organizations**

Chapter Topics

The Hadoop Ecosystem

- Introduction
- Data Storage: HBase
- Data Integration: Flume, Sqoop
- Data Processing: Spark
- **Data Analysis: Hive, Pig, and Impala**
- Workflow Engine: Oozie
- Machine Learning: Mahout

Hive and Pig: High Level Data Languages

- **The motivation: MapReduce is powerful but hard to master**
- **The solution: Hive and Pig**
  - Languages for querying and manipulating data
  - Leverage existing skillsets
    - Data analysts who use SQL
    - Programmers who use scripting languages
  - Open source Apache projects
    - Hive initially developed at Facebook
    - Pig Initially developed at Yahoo!
- **Interpreter runs on a client machine**
  - Turns queries into MapReduce jobs
  - Submits jobs to the cluster

Hive

- **What is Hive?**
  - **HiveQL**: An SQL-like interface to Hadoop


```
SELECT * FROM purchases WHERE price > 10000 ORDER BY
storeid
```

Pig

- **What is Pig?**
  - **Pig Latin**: A dataflow language for transforming large data sets

```
purchases = LOAD "/user/dave/purchases" AS (itemID,
                    price, storeID, purchaserID);
bigticket = FILTER purchases BY price > 10000;
...
```

Hive vs. Pig

	Hive	Pig
Language	HiveQL (SQL-like)	Pig Latin (dataflow language)
Schema	Table definitions stored in a metastore	Schema optionally defined at runtime
Programmatic access	JDBC, ODBC	PigServer (Java API)

JDBC: Java Database Connectivity
ODBC: Open Database Connectivity

Impala: High Performance Queries

- **High-performance SQL engine for vast amounts of data**
  - Similar query language to HiveQL
  - 10 to 50+ times faster than Hive, Pig, or MapReduce
- **Impala runs on Hadoop clusters**
  - Data stored in HDFS
  - Does not use MapReduce
- **Developed by Cloudera**
  - 100% open source, released under the Apache software license

Which to Choose?

- **Use Impala when…**
  - You need near real-time responses to ad hoc queries
  - You have structured data with a defined schema
- **Use Hive or Pig when…**
  - You need support for custom file types, complex data types, or external functions
- **Use Pig when…**
  - You have developers experienced with writing scripts
  - Your data is unstructured/multi-structured
- **Use Hive When…**
  - You have analysts familiar with SQL
  - You are integrating with BI or reporting tools via ODBC/JDBC

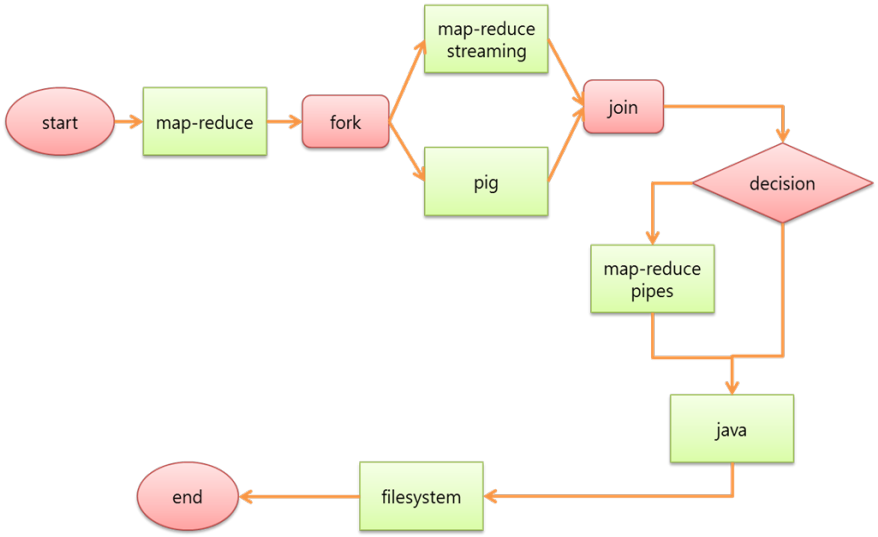
Chapter Topics

The Hadoop Ecosystem

- Introduction
- Data Storage: HBase
- Data Integration: Flume, Sqoop
- Data Processing: Spark
- Data Analysis: Hive, Pig, and Impala
- **Workflow Engine: Oozie**
- Machine Learning: Mahout

Oozie

- **Oozie**
  - Workflow engine for MapReduce jobs
  - Defines dependencies between jobs
- **The Oozie server submits the jobs to the server in the correct sequence**
- **We will investigate Oozie later in the course**



Chapter Topics

The Hadoop Ecosystem

- Introduction
- Data Storage: HBase
- Data Integration: Flume, Sqoop
- Data Processing: Spark
- Data Analysis: Hive, Pig, and Impala
- Workflow Engine: Oozie
- **Machine Learning: Mahout**

Mahout

- **Mahout is a Machine Learning library written in Java**
- **Used for**
  - Collaborative filtering (recommendations)
  - Clustering (finding naturally occurring "groupings" in data)
  - Classification (determining whether new data fits a category)
- **Why use Hadoop for Machine Learning?**
  - "It's not who has the best algorithms that wins. It's who has the most data."

Key Points

- **Hadoop Ecosystem**
  - Many projects built on, and supporting, Hadoop
  - Several will be covered in detail later in the course

Bibliography

The following offer more information on topics discussed in this chapter

- **HBase was inspired by Google's "BigTable" paper presented at OSDI in 2006**
  - http://research.google.com/archive/bigtable-osdi06.pdf
- **An interesting link comparing NoSQL databases**
  - http://www.networkworld.com/news/tech/2012/102212-nosql-263595.html
- **AMPLab**
  - https://amplab.cs.berkeley.edu
- **Databricks**
  - http://databricks.com

Bibliography (cont'd)

The following offer more information on topics discussed in this chapter

- **Spark**
  - http://databricks.com/blog/2013/10/28/databricks-and-cloudera-partner-to-support-spark.html
- **Dremel is a distributed system for interactive ad-hoc queries that was created by Google**
  - http://research.google.com/pubs/archive/36632.pdf
- **Impala, developed by Cloudera, supports the same inner, outer, and semi-joins that Hive does**
  - http://tiny.cloudera.com/dac15b

Managing Your Hadoop Solution

Chapter 2.2

Managing Your Hadoop Solution

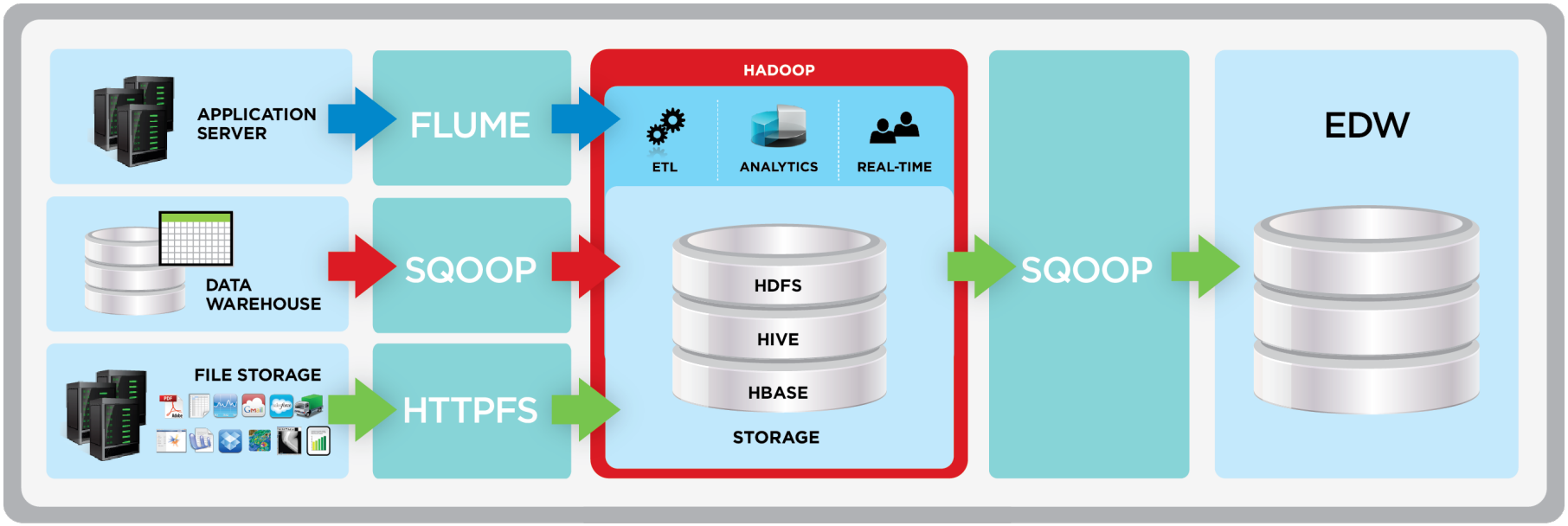
- **What is the typical architecture of a data center with Hadoop?**
- **What are typical hardware requirements for a Hadoop cluster?**

Chapter Topics

Managing Your Hadoop Solution

- **Hadoop in the Data Center**
- Cluster Hardware

A Typical Data Center With Hadoop

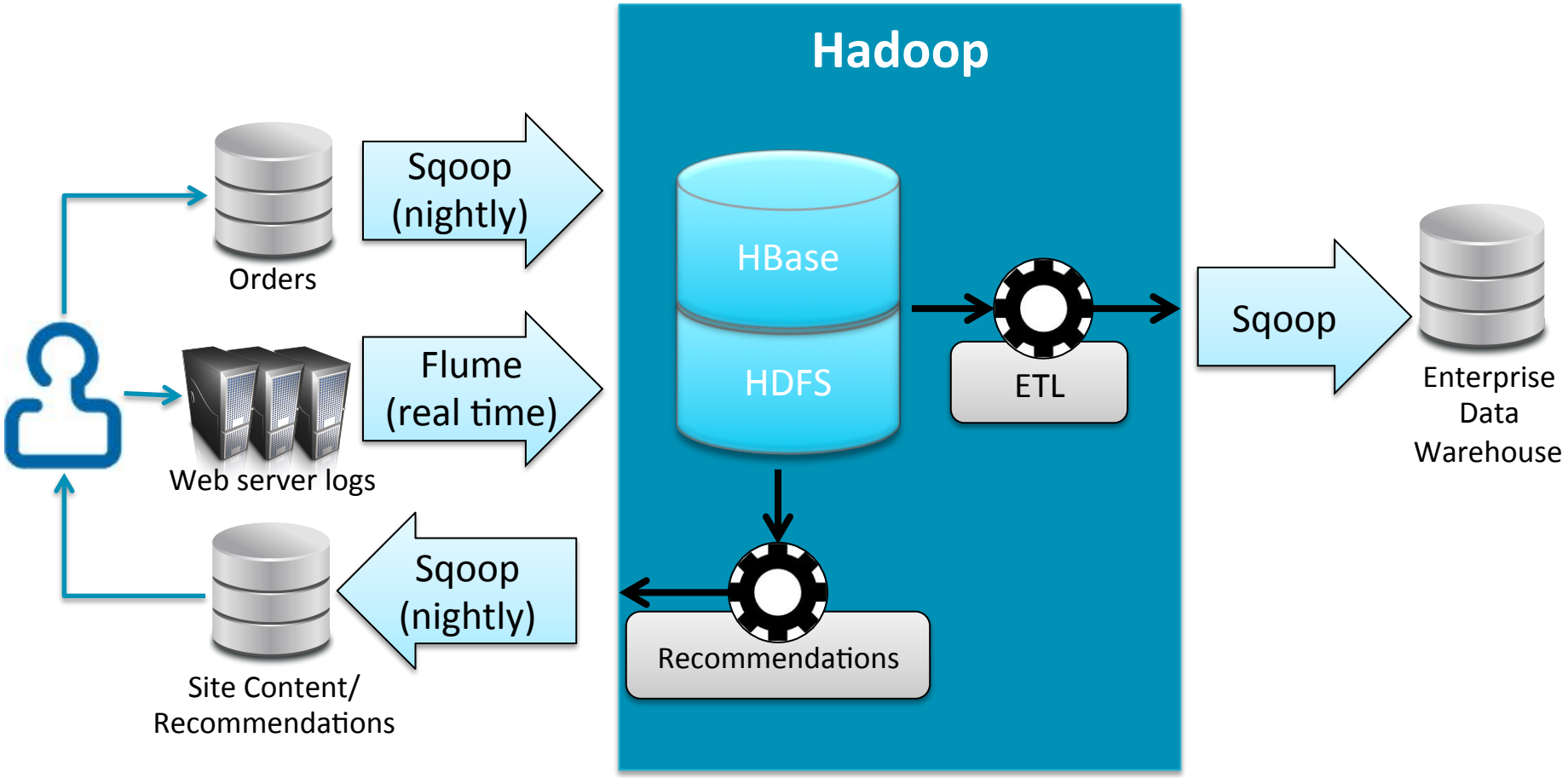


Technology Strengths and Weaknesses

Technology	Strengths	Drawbacks
Hadoop	• Scales to petabytes • Import/export tools • Flexible structure • Commodity hardware	• Query speed • No transaction support
Relational Databases	• Complex transactions • 1000s of queries/second • SQL	• Data must fit into rows and columns • Schema is costly to change • Full table scans are slow
Data warehouses	• Reporting capabilities • Up to 100s of terabytes	• Dimensions require pre-materialization
File servers (SAN/ NAS)	• Serving individual files • Write caches	• Cost • Reading large portions of the data saturates the network
Backup systems (tape)	• Cheap	• Expensive to retrieve the data

SAN: Storage Area Network
NAS: Network-Attached Storage

Example Data Flow

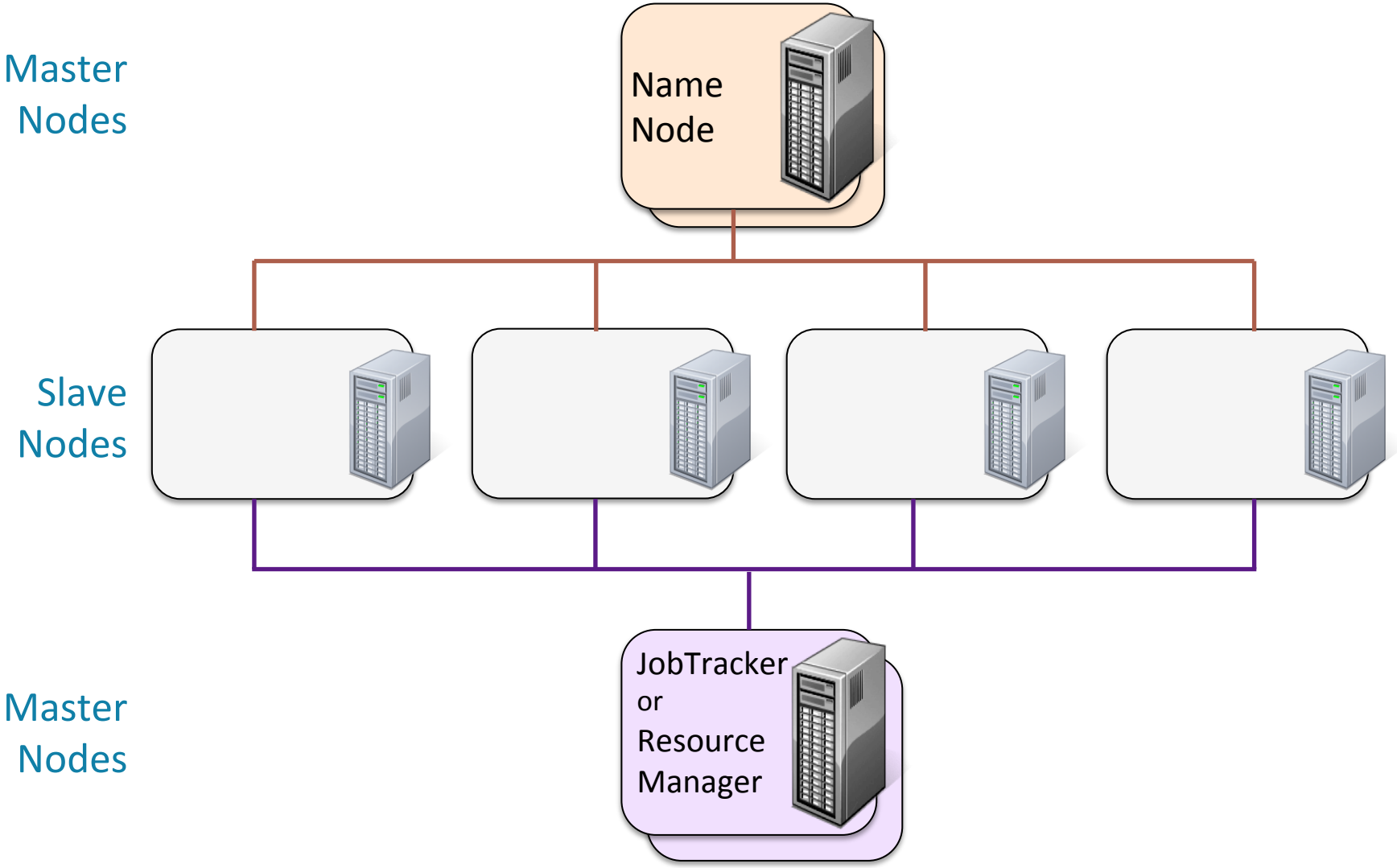


Chapter Topics

Managing Your Hadoop Solution

- Hadoop in the Data Center
- **Cluster Hardware**

Cluster Hardware



Slave Nodes: Recommended Configurations (1)

- **Processors**
  - Mid-grade processors (e.g., 2 x 6-core 2.9 GHz)
- **Memory**
  - 48-96GB RAM
- **Network**
  - 1Gb Ethernet (mid-range)
  - 10Gb Ethernet (high-end)
- **Disk Drives**
  - 6 x 2TB drives per machine (mid-range)
  - 12 x 3TB drives per machine (high-end)
  - Non-RAID

RAID: Redundant Array of Inexpensive Disks

Slave Nodes: Recommended Configurations (2)

- **Switch**
  - Dedicated switching infrastructure required because Hadoop can saturate the network
    - "All nodes talking to all nodes"
- **Cost**
  - Per slave node cost should be around $4,000 to $10,000 (2014 estimate)

Master Nodes Are More Important

- **Four master nodes**
  - NameNode (active and standby)
  - JobTracker (MRv1) or ResourceManager (MRv2) (active and standby)
    - Some installations just use a single JobTracker or Resource Manager
- **Each typically runs on a separate machine**
- **Master node machines are high-quality servers**
  - Carrier-class hardware
  - Dual power supplies, Ethernet cards
  - RAIDed hard drives
  - 24GB RAM for clusters of 20 nodes or less
  - 48GB RAM for clusters of up to 300 nodes
  - 96GB RAM for larger clusters

Capacity Planning (1)

- **Basing your cluster growth on storage capacity is often a good method to use**
- **Example:**
  - Data grows by approximately 3TB per week/40TB per quarter
  - Hadoop replicates 3 times = 120TB
  - Extra space required for temporary data while running jobs (~30%) = 160TB
  - Assuming machines with 12 x 3TB hard drives
    - 4-5 new machines per quarter
    - Two years of data = 1.3PB
      - Requires approximately 36 machines

Capacity Planning (2)

- **New nodes are automatically used by Hadoop**
- **Many clusters start small (less than 10 nodes) and scale up as data and processing demands grow**
- **Hadoop clusters can grow to thousands of nodes**

Bibliography

The following offer more information on topics discussed in this chapter

- **Slave and master node configuration recommendations: Hadoop Operations, first edition (1e), by Eric Sammer, p. 46 and 50.**

Introduction to MapReduce

Chapter 2.3

Introduction to MapReduce

- **Concepts behind MapReduce**
- **How does data flow through MapReduce stages**
- **Typical uses of Mappers**
- **Typical uses of Reducers**

Chapter Topics

Introduction to MapReduce

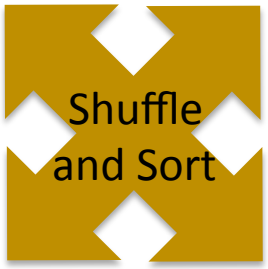
- **MapReduce Overview**
- Example: WordCount
- Mappers
- Reducers

Review - Features of MapReduce

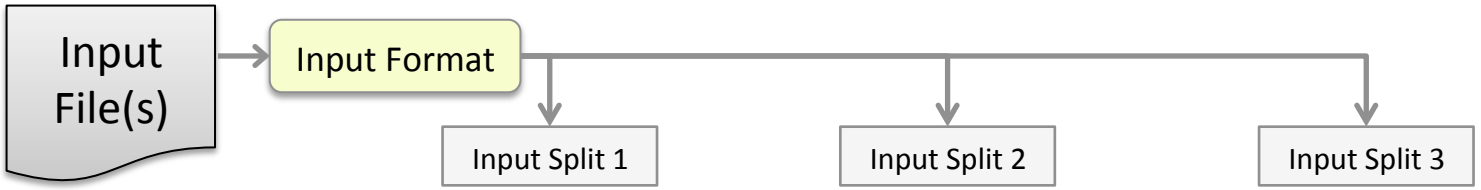
- **Automatic parallelization and distribution**
- **Fault-tolerance**
- **A clean abstraction for programmers**
  - MapReduce programs are usually written in Java
    - Can be written in any language using *Hadoop Streaming*
    - All of Hadoop is written in Java
- **MapReduce abstracts all the 'housekeeping' away from the developer**
  - Developer can simply concentrate on writing the Map and Reduce functions

Review – Key MapReduce Stages

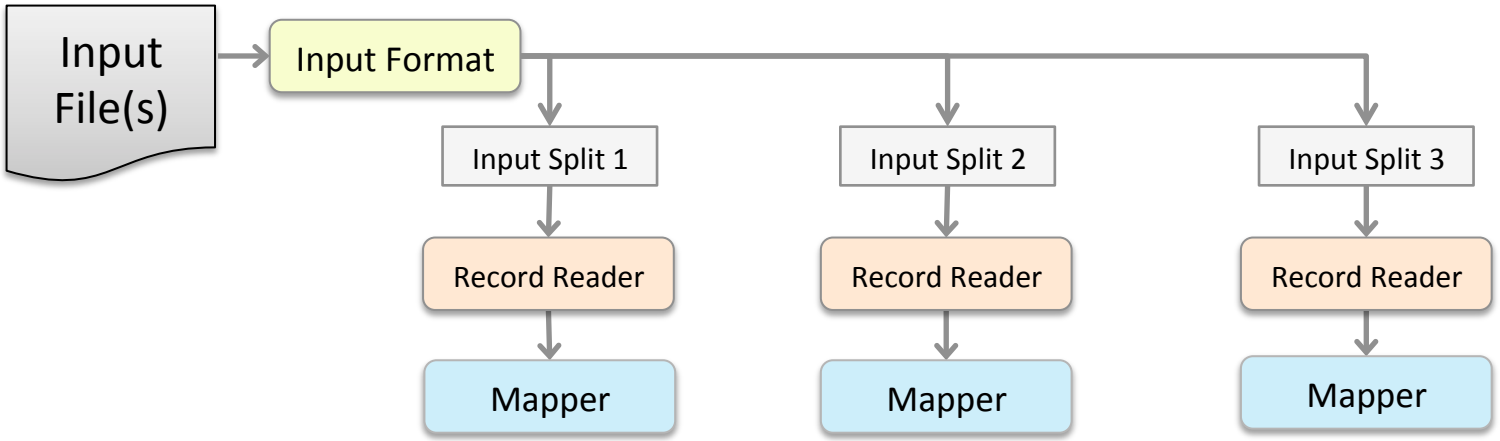
- **The Mapper**
  - Each Map task (typically) operates on a single HDFS block
  - Map tasks (usually) run on the node where the block is stored
- **Shuffle and Sort**
  - Sorts and consolidates intermediate data from all mappers
  - Happens after all Map tasks are complete and before Reduce tasks start
- **The Reducer**
  - Operates on shuffled/sorted intermediate data (Map task output)
  - Produces final output



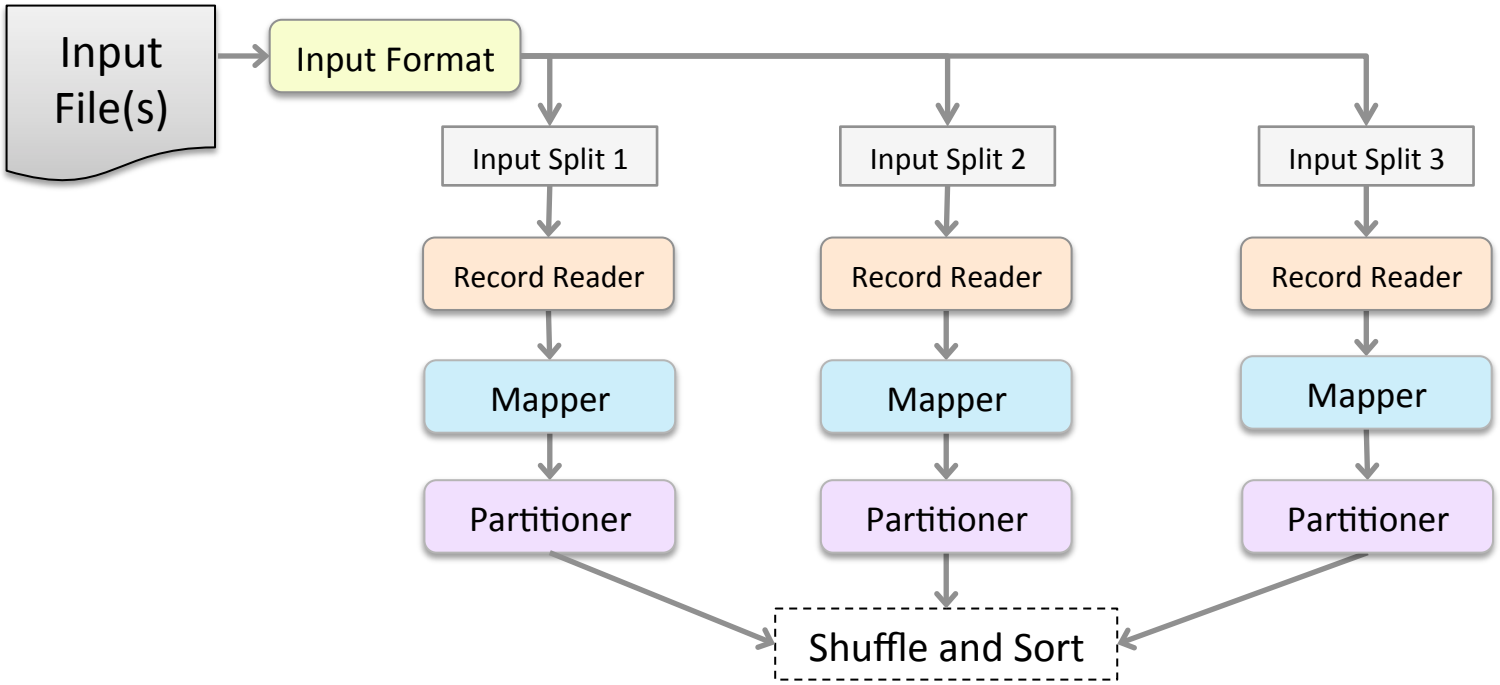
The MapReduce Flow



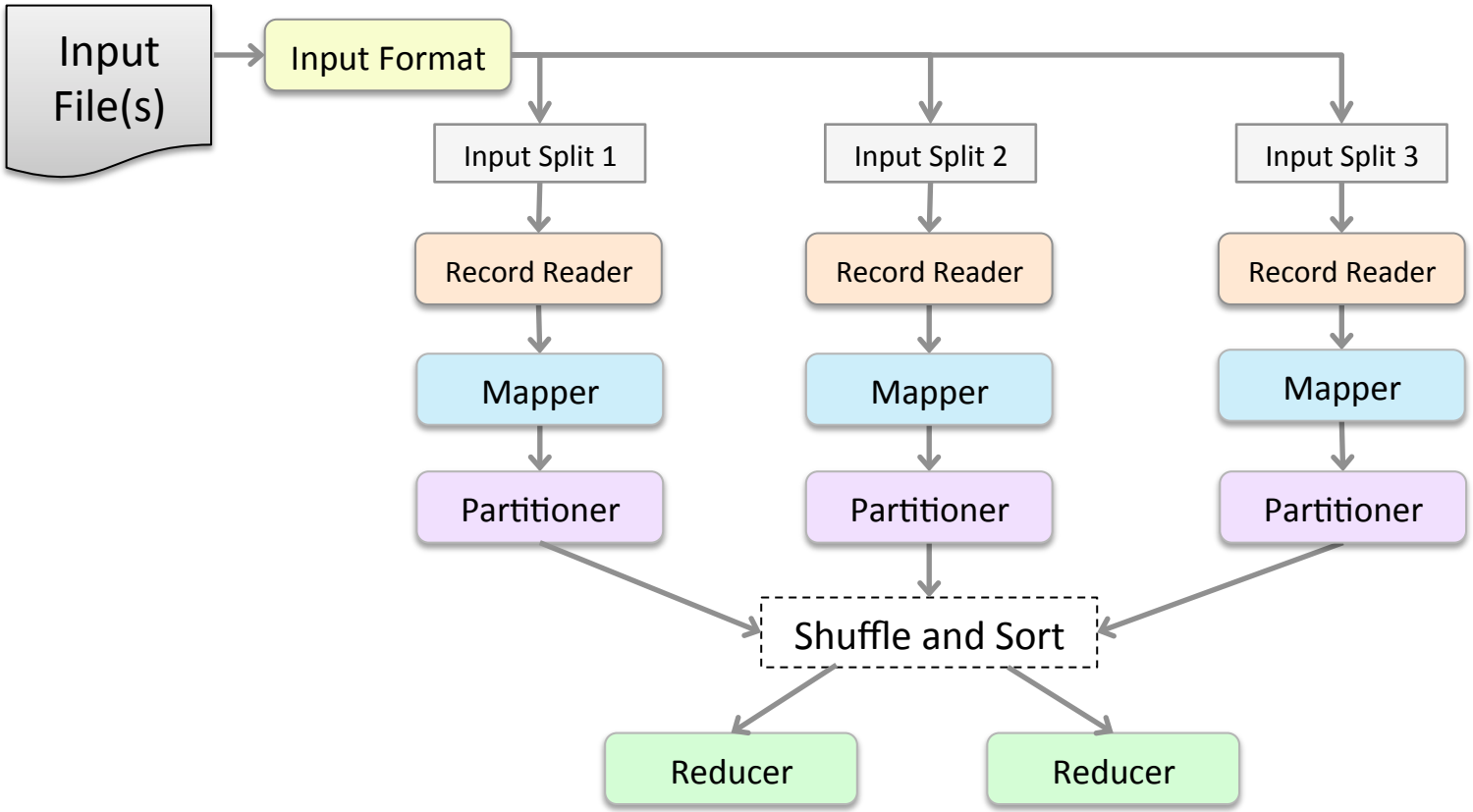
The MapReduce Flow



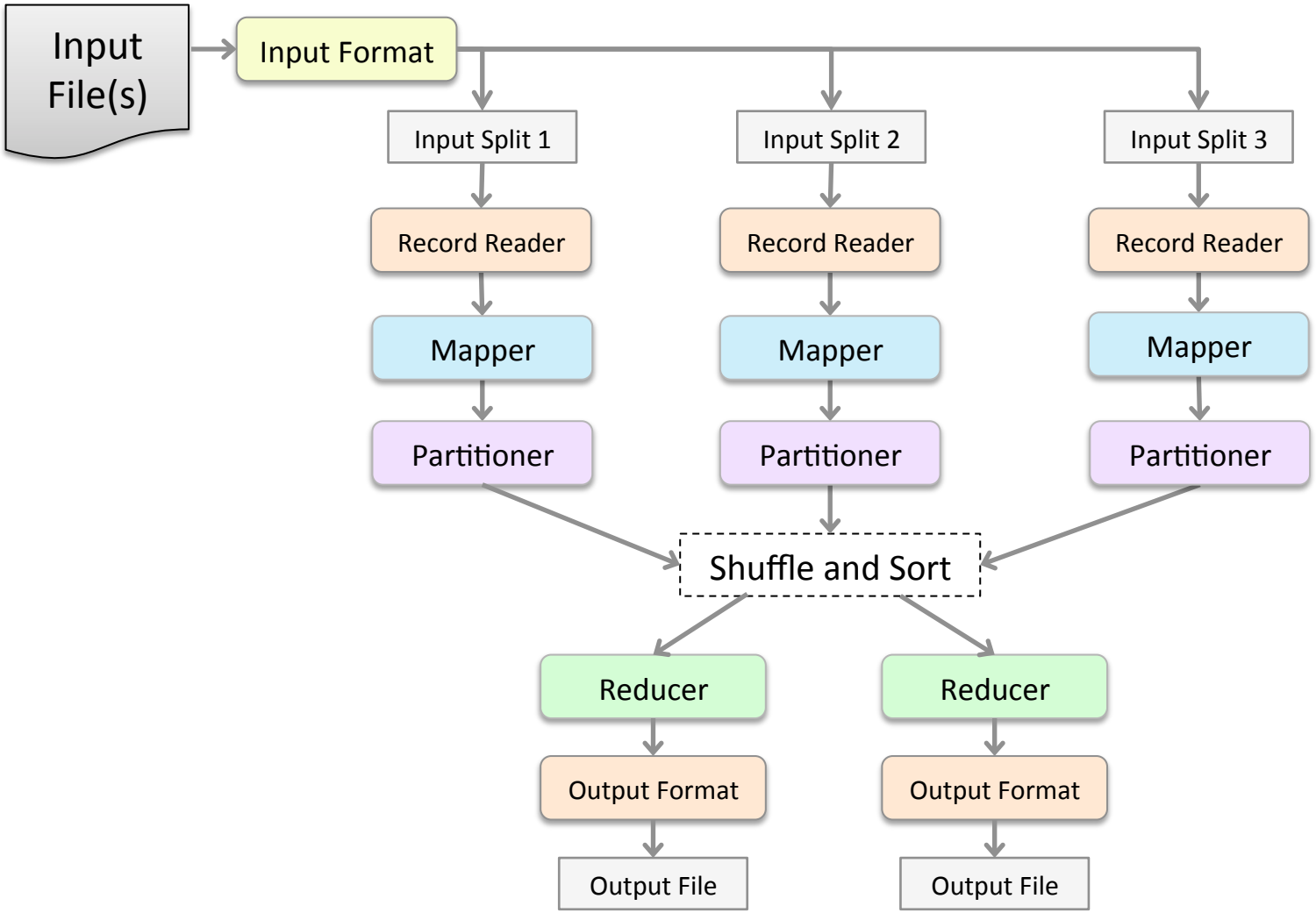
The MapReduce Flow



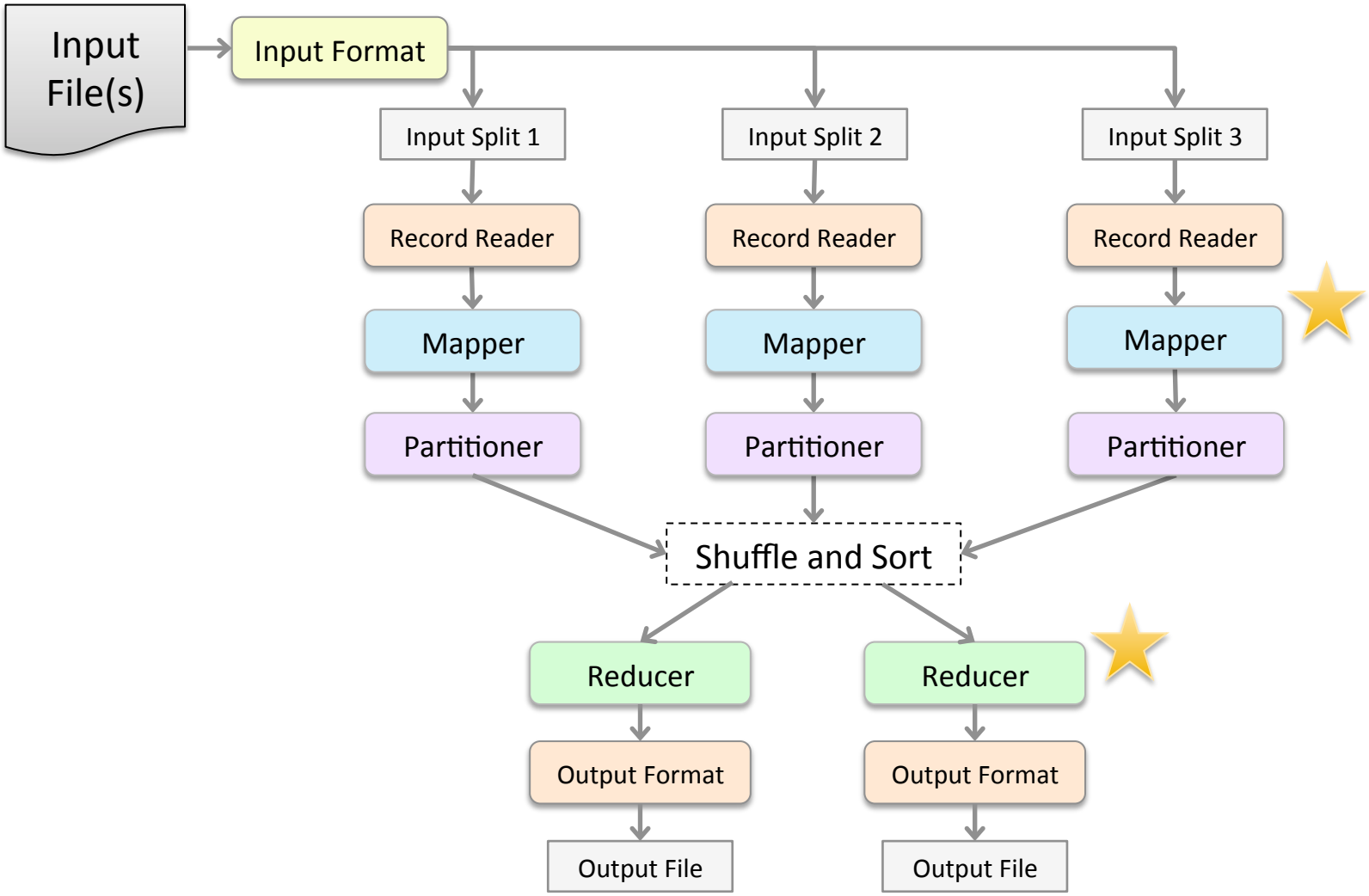
The MapReduce Flow



The MapReduce Flow



The MapReduce Flow

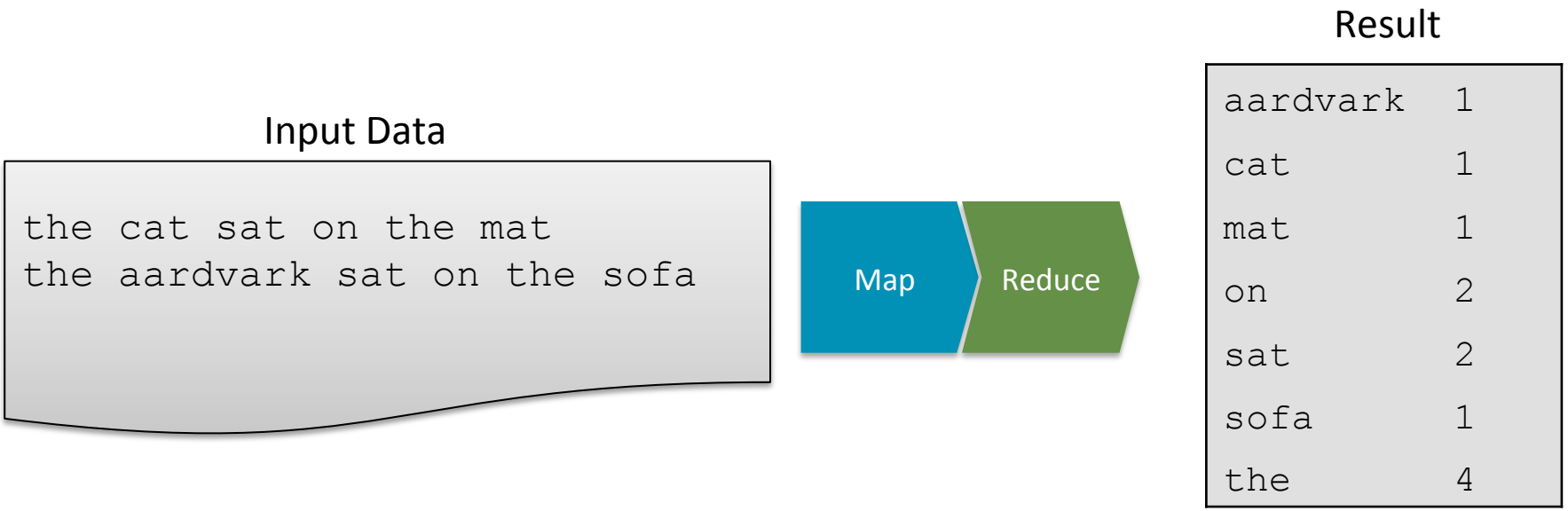


Chapter Topics

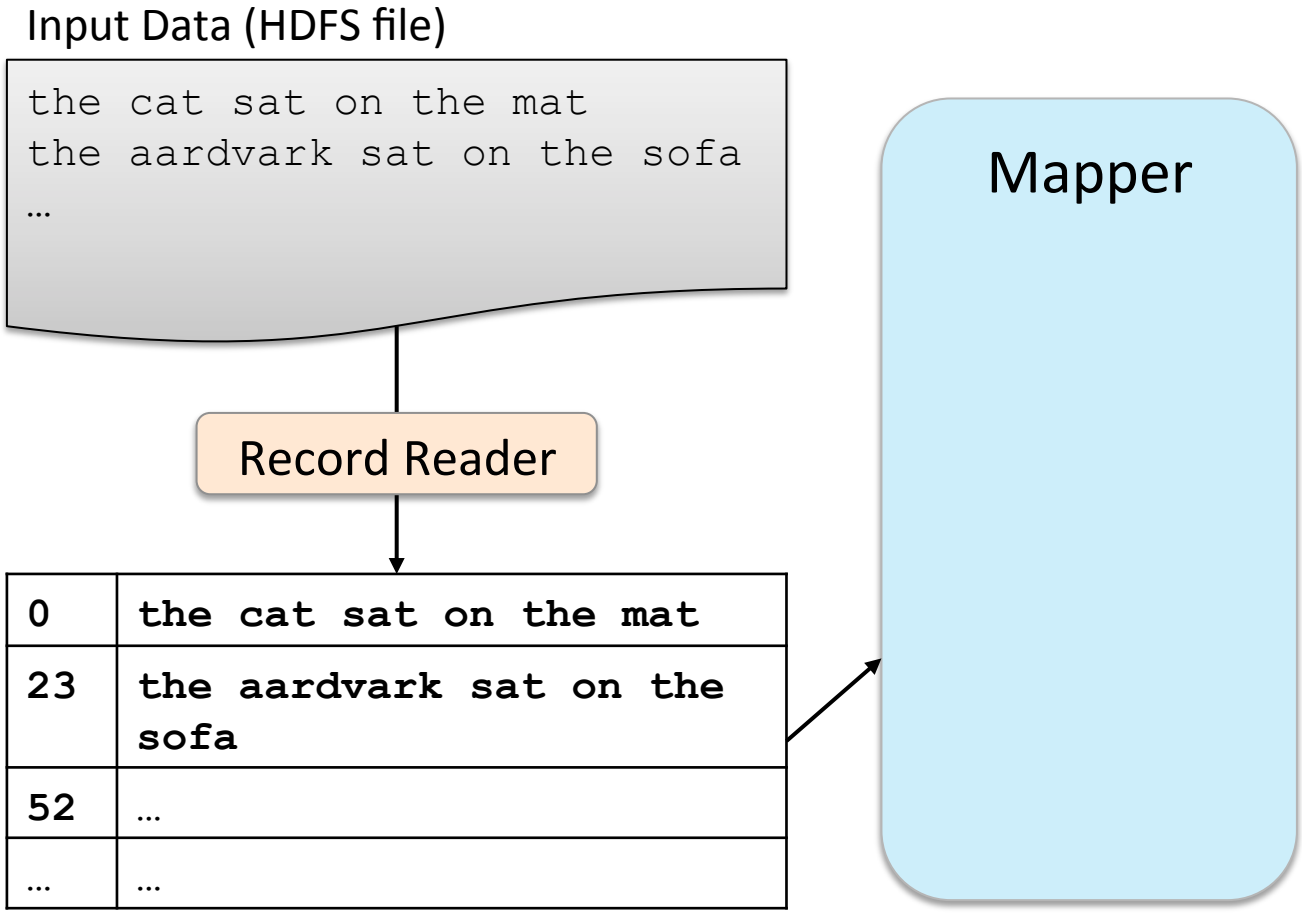
Introduction to MapReduce

- MapReduce Overview
- **Review: WordCount Example**
- Mappers
- Reducers

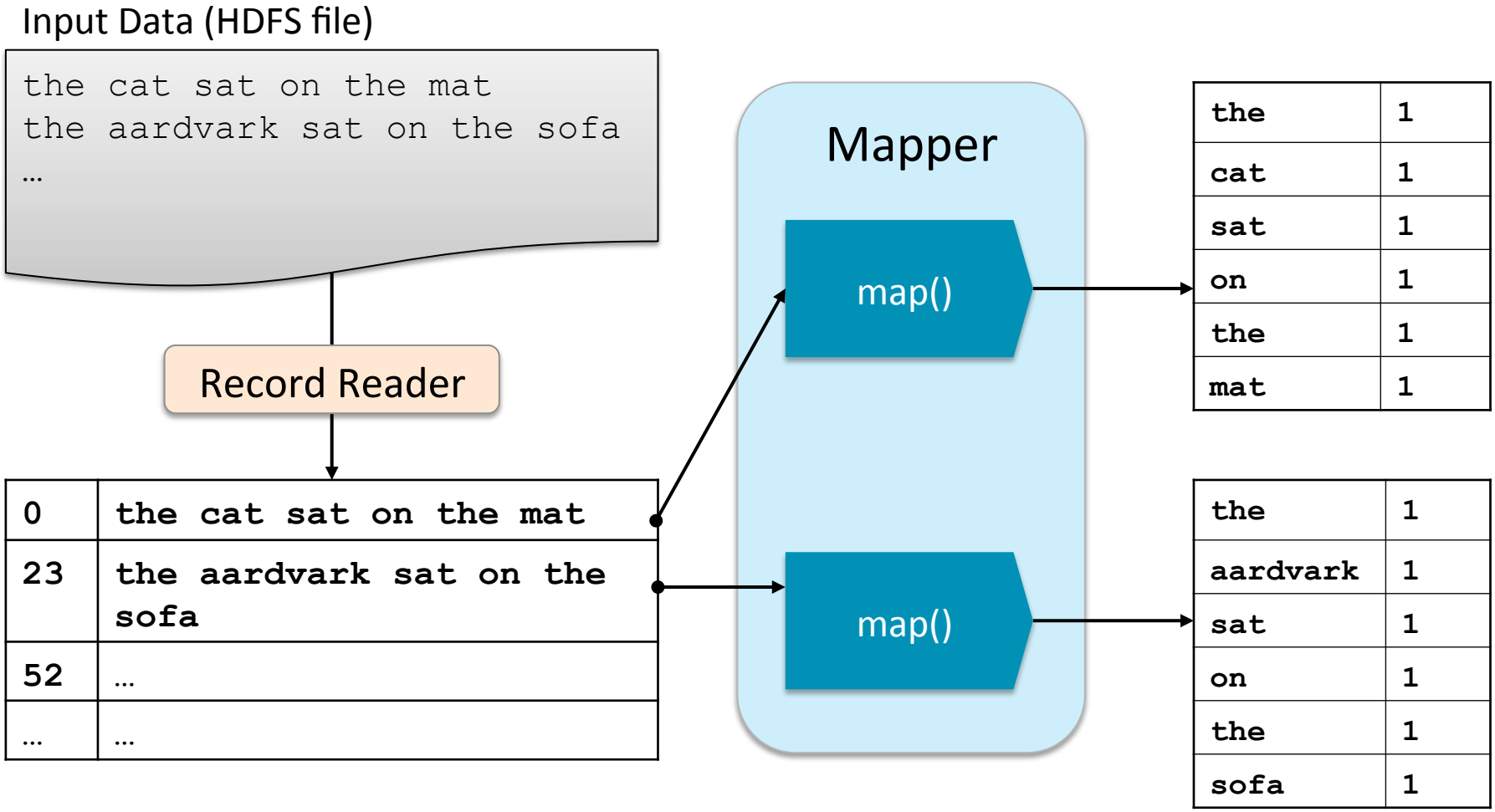
Example: Word Count



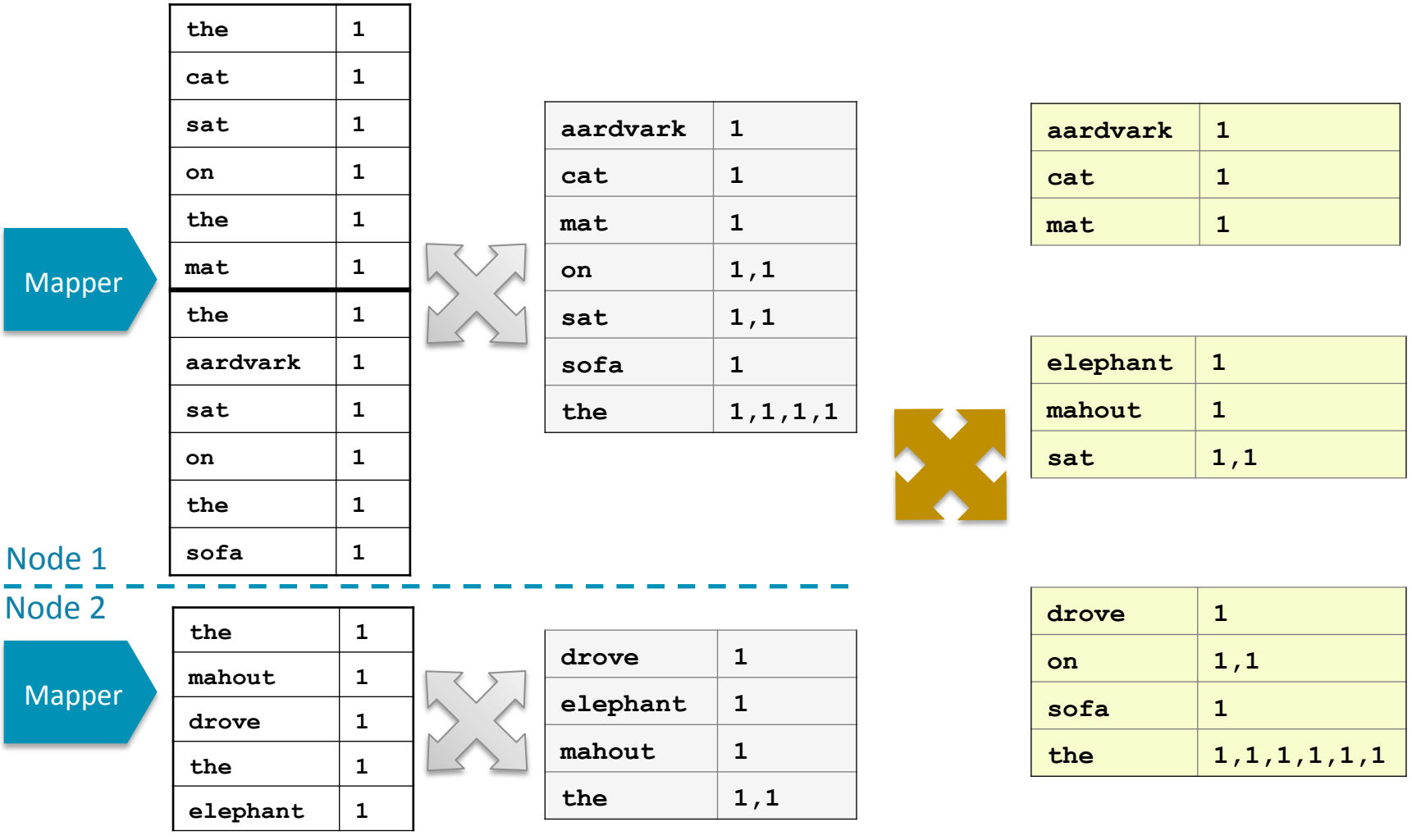
Example: The WordCount Mapper (1)



Example: The WordCount Mapper (2)



Example: WordCount Shuffle and Sort



the	1
cat	1
sat	1
on	1
the	1
mat	1

the	1
aardvark	1
sat	1
on	1
the	1
sofa	1

the	1
mahout	1
drove	1
the	1
elephant	1

aardvark	1
cat	1
mat	1
on	1,1
sat	1,1
sofa	1
the	1,1,1,1

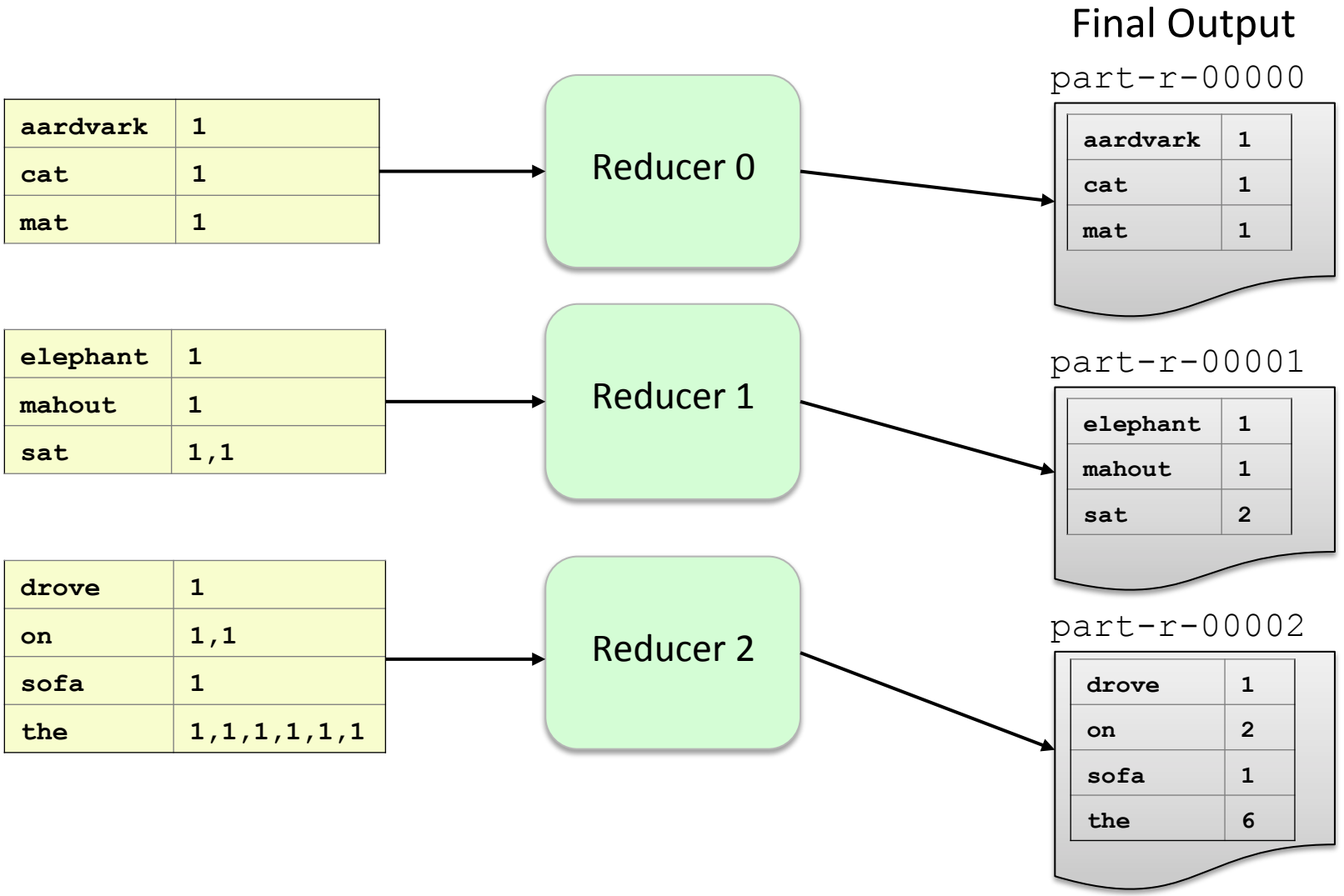
drove	1
elephant	1
mahout	1
the	1,1

aardvark	1
cat	1
mat	1

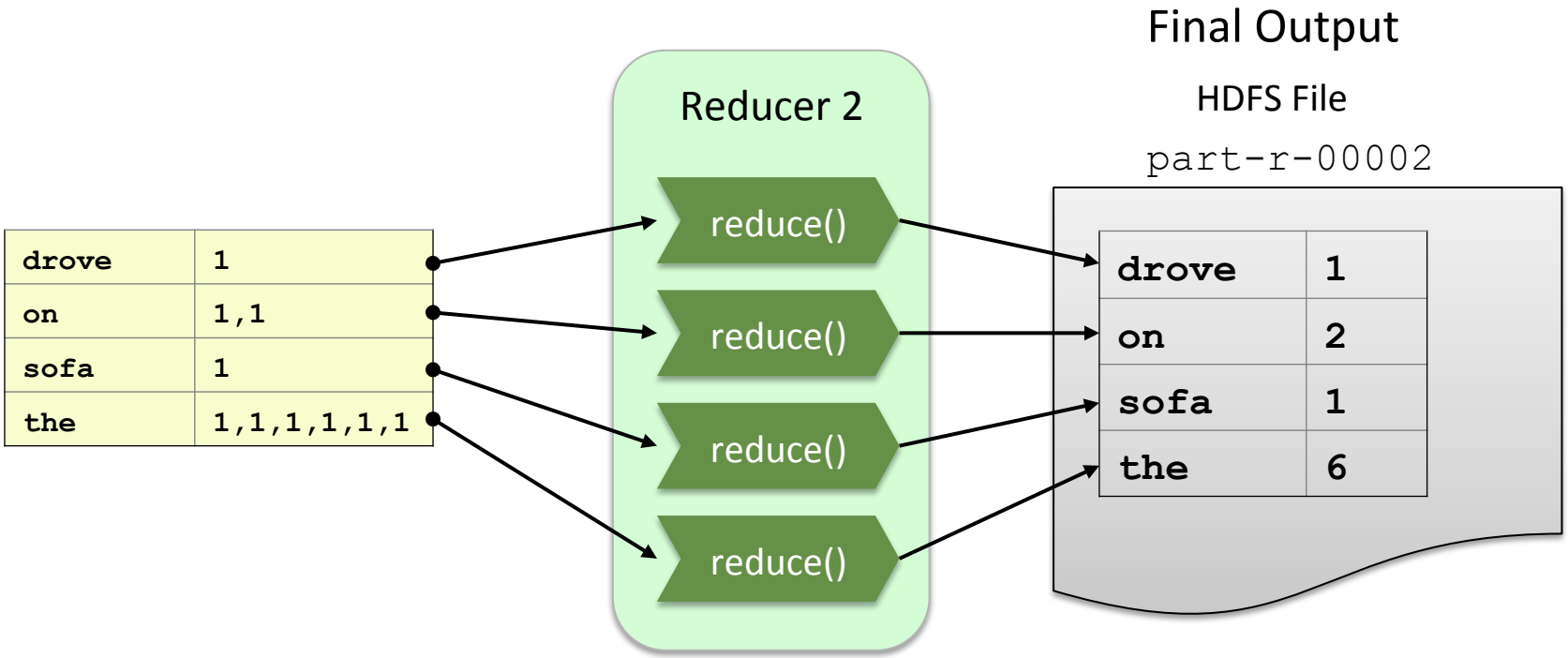
elephant	1
mahout	1
sat	1,1

drove	1
on	1,1
sofa	1
the	1,1,1,1,1,1

Example: SumReducer (1)



Example: SumReducer (2)



Chapter Topics

Introduction to MapReduce

- MapReduce Overview
- Example: WordCount
- **Mappers**
- Reducers

MapReduce: The Mapper (1)

- **The Mapper**
  - Input: key/value pair
  - Output: A list of zero or more key value pairs

```
map(in_key, in_value) →
                        (inter_key, inter_value) list
```

input key	input value

map

intermediate key 1	value 1
intermediate key 2	value 2
intermediate key 3	value 3
…	…

MapReduce: The Mapper (2)

- **The Mapper may use or completely ignore the input key**
  - For example, a standard pattern is to read one line of a file at a time
    - The key is the byte offset into the file at which the line starts
    - The value is the contents of the line itself
    - Typically the key is considered irrelevant
- **If the Mapper writes anything out, the output must be in the form of key/value pairs**

23	the aardvark sat on the sofa

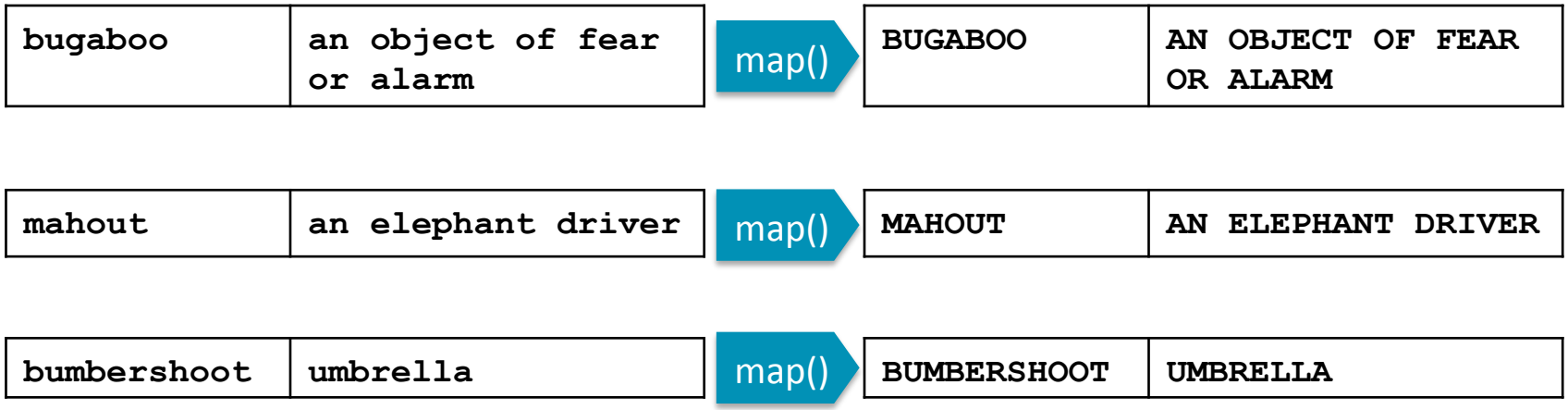
map

the	1
aardvark	1
sat	1
on	1
the	1
sofa	1

Example Mapper: Upper Case Mapper

- **Turn input into upper case (pseudo-code):**

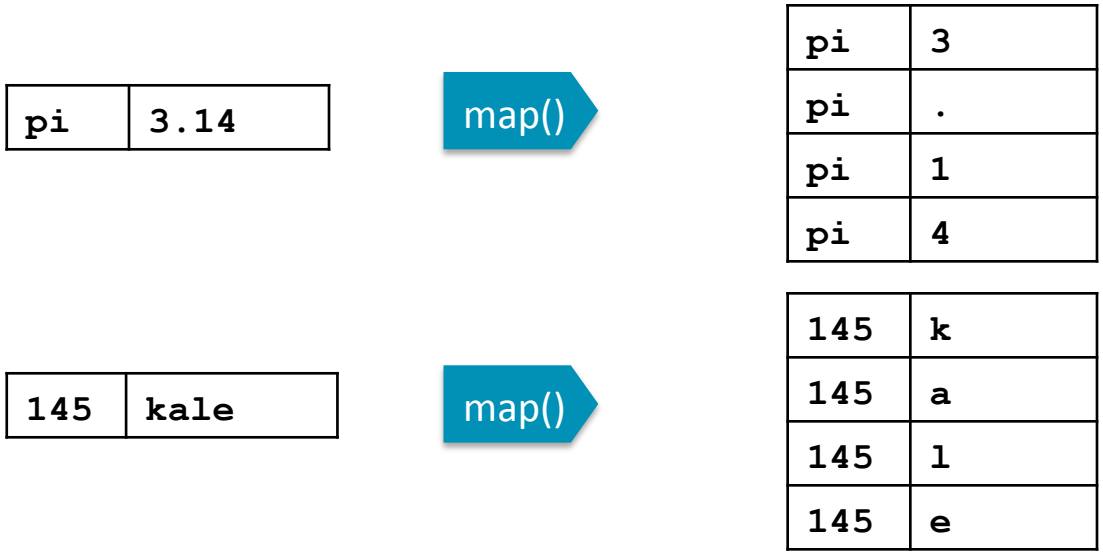
```
let map(k, v) =
    emit(k.toUpper(), v.toUpper())
```



Example Mapper: 'Explode' Mapper

- **Output each input character separately (pseudo-code):**

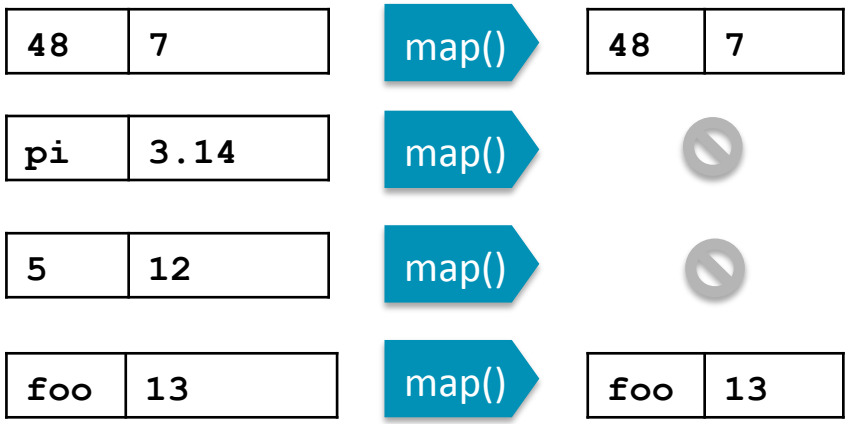
```
let map(k, v) =
    foreach char c in v:
        emit (k, c)
```



Example Mapper: 'Filter' Mapper

- **Only output key/value pairs where the input value is a prime number (pseudo-code):**

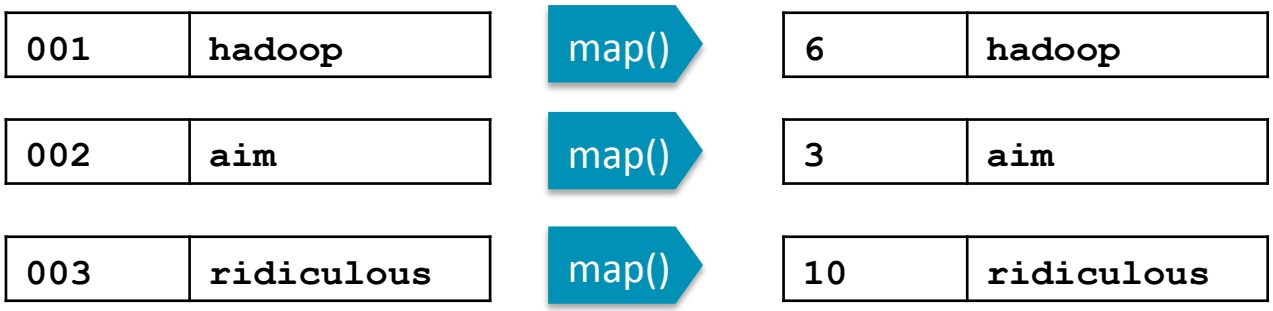
```
let map(k, v) =
    if (isPrime(v)) then emit(k, v)
```



Example Mapper: Changing Keyspaces

- **The key output by the Mapper does not need to be identical to the input key**
- **Example: output the word length as the key (pseudo-code):**

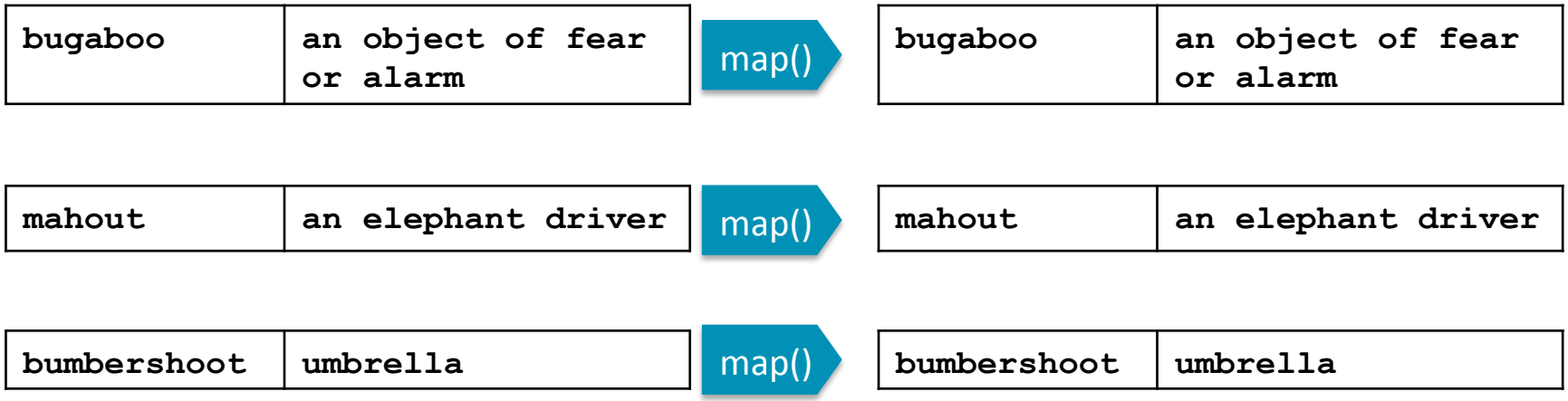
```
let map(k, v) =
    emit(v.length(), v)
```



Example Mapper: Identity Mapper

- **Emit the key,value pair (pseudo-code):**

```
let map(k, v) =
    emit(k,v)
```



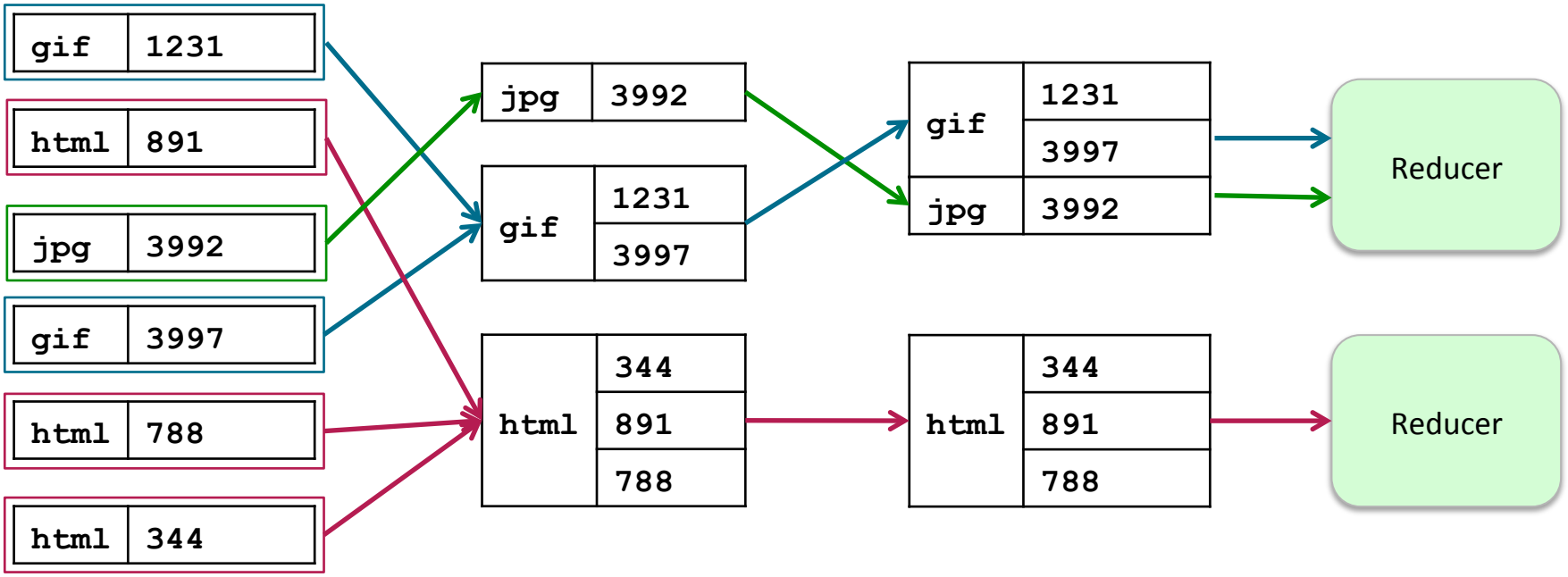
Chapter Topics

Introduction to MapReduce

- MapReduce Overview
- Example: WordCount
- Mappers
- **Reducers**

Shuffle and Sort

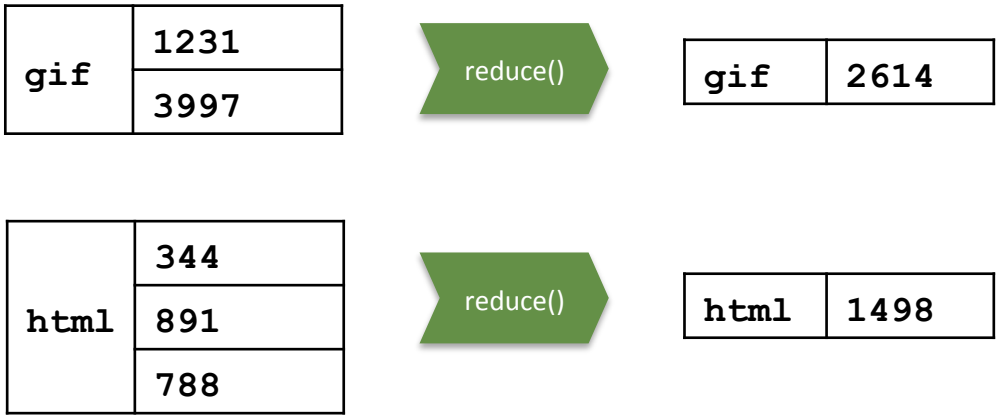
- **After the Map phase is over, all intermediate values for a given intermediate key are grouped together**
- **Each key and value list is passed to a Reducer**
  - All values for a particular intermediate key go to the same Reducer
  - The intermediate keys/value lists are passed in sorted key order



The Reducer

- **The Reducer outputs zero or more final key/value pairs**
  - In practice, usually emits a single key/value pair for each input key
  - These are written to HDFS

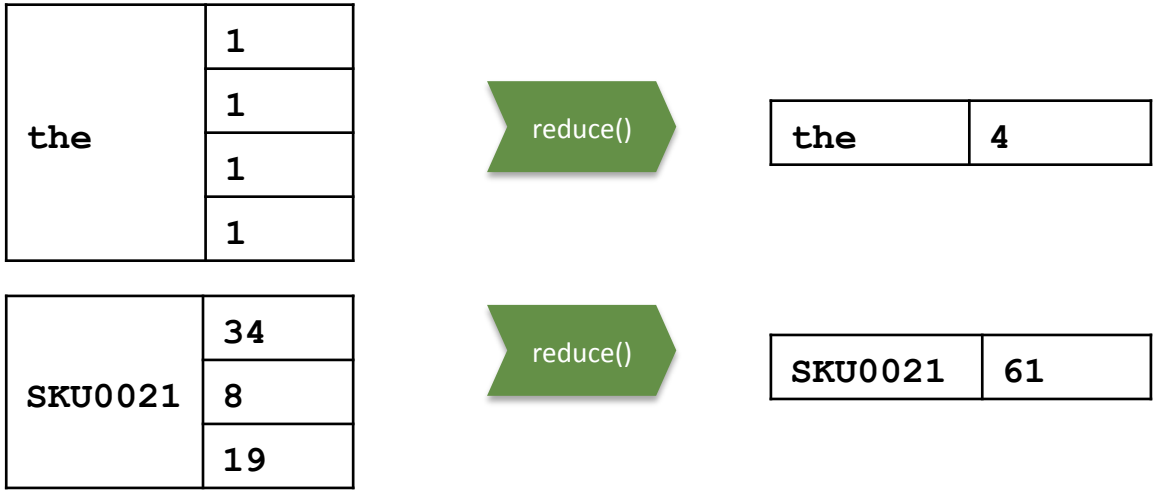
```
reduce(inter_key, [v1, v2, …]) →
                        (result_key, result_value)
```



Example Reducer: Sum Reducer

- **Add up all the values associated with each intermediate key (pseudo-code):**

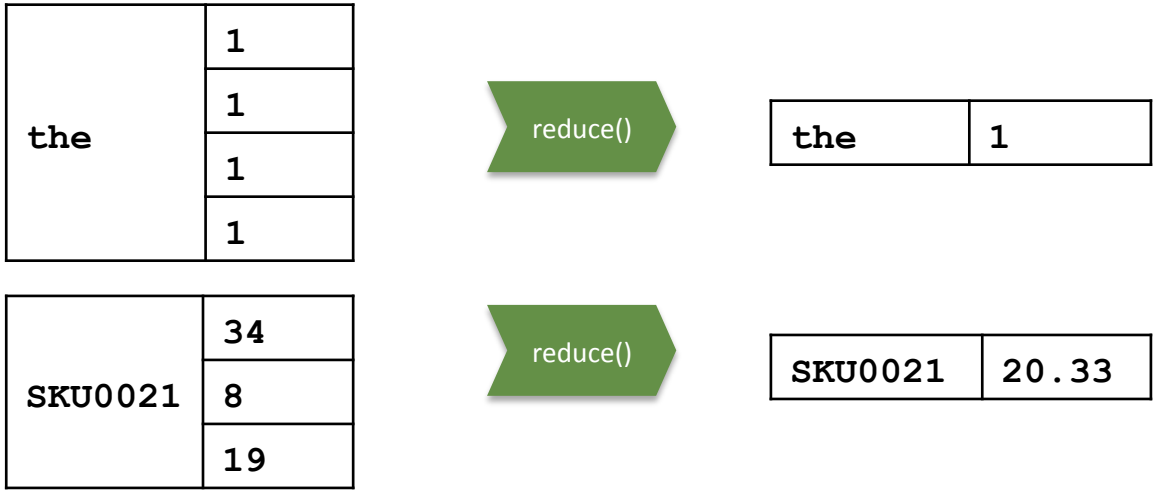
```
let reduce(k, vals) =
    sum = 0
    foreach int i in vals:
        sum += i
    emit(k, sum)
```



Example Reducer: Average Reducer

- **Find the mean of all the values associated with each intermediate key (pseudo-code):**

```
let reduce(k, vals) =
    sum = 0; counter = 0;
    foreach int i in vals:
        sum += i; counter += 1;
    emit(k, sum/counter)
```



Example Reducer: Identity Reducer

- **The Identity Reducer is very common (pseudo-code):**

```
let reduce(k, vals) =
    foreach v in vals:
        emit(k, v)
```

bow	a knot with two loops and two loose ends
	a weapon for shooting arrows
	a bending of the head or body in respect

reduce()

bow	a knot with two loops and two loose ends
bow	a weapon for shooting arrows
bow	a bending of the head or body in respect

28	2
	2
	7

reduce()

28	2
28	2
28	7

Key Points

- **A MapReduce program has two major developer-created components: a Mapper and a Reducer**
- **Mappers map input data to intermediate key/value pairs**
  - Often parse, filter, or transform the data
- **Reducers process Mapper output into final key/value pairs**
  - Often aggregate data using statistical functions

Hadoop Clusters

Chapter 2.4

Hadoop Clusters

- **Components of a Hadoop cluster**
- **How do Hadoop jobs and tasks run on a cluster**
- **How does a job's data flow in a Hadoop cluster**

Chapter Topics

Hadoop Clusters

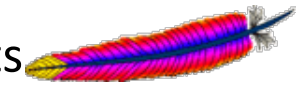
- **Hadoop Cluster Overview**
- Hadoop Jobs and Tasks

Installing A Hadoop Cluster (1)

- **After testing a Hadoop solution on a small sample dataset, the solution can be run on a Hadoop cluster to process all data**
- **Clusters are typically installed and maintained by a system administrator**
- **Developers benefit by understanding how the components of a Hadoop cluster work together**

Installing A Hadoop Cluster (2)

- **Difficult**
  - Download, install, and integrate individual Hadoop components directly from Apache

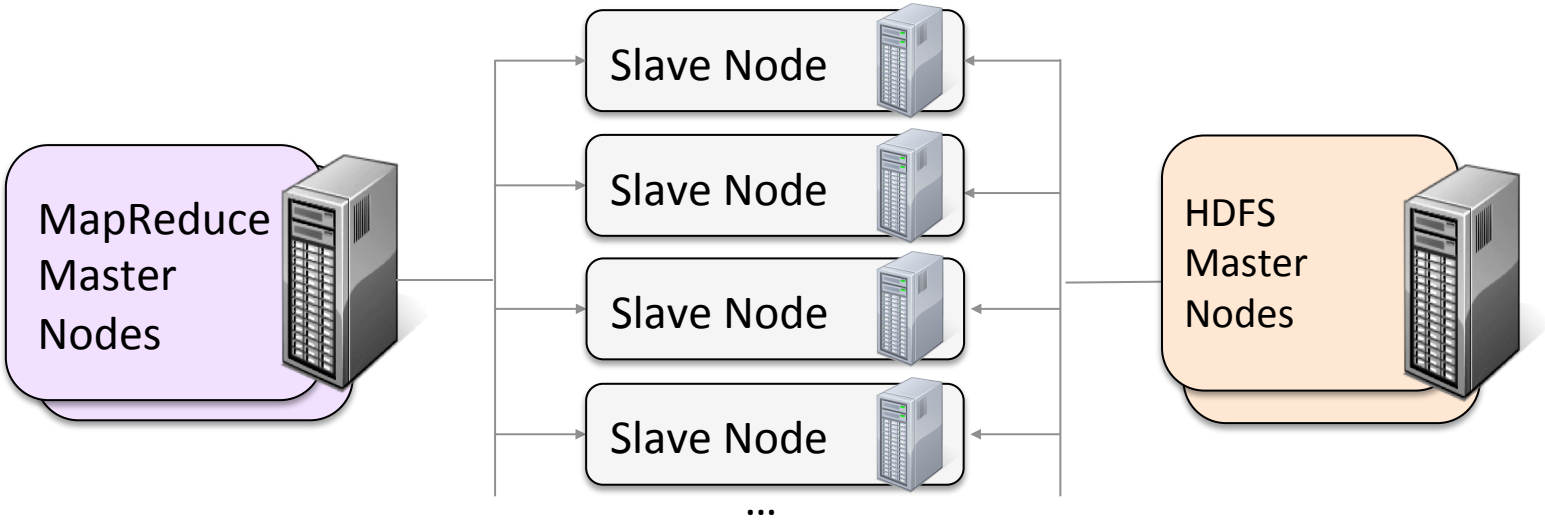


- **Easier: CDH**
  - Cloudera's Distribution for Apache Hadoop
  - Vanilla Hadoop plus many patches, backports, bug fixes
  - Includes many other components from the Hadoop ecosystem

- **Easiest: Cloudera Manager**
  - Wizard-based UI to install, configure and manage a Hadoop cluster
  - Included with Cloudera Standard (free) or Cloudera Enterprise

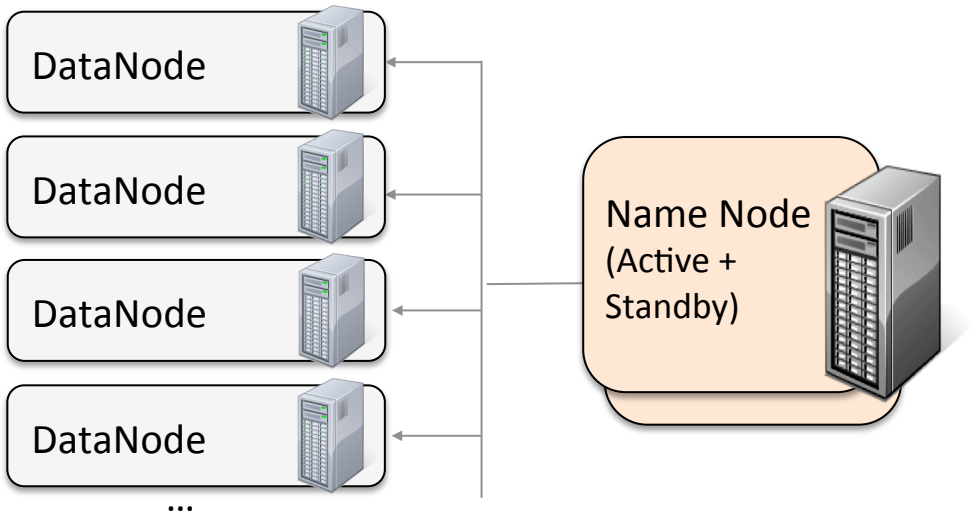
Hadoop Cluster Terminology

- **A Hadoop *cluster* is a group of computers working together**
  - Usually runs HDFS and MapReduce
- **A *node* is an individual computer in the cluster**
  - Master nodes manage distribution of work and data to slave nodes
- **A *daemon* is a program running on a node**
  - Each performs different functions in the cluster



Hadoop Daemons: HDFS

- **HDFS daemons**
  - NameNode – holds the metadata for HDFS
    - Typically two on a production cluster: one active, one standby
  - DataNode – holds the actual HDFS data
    - One per slave node

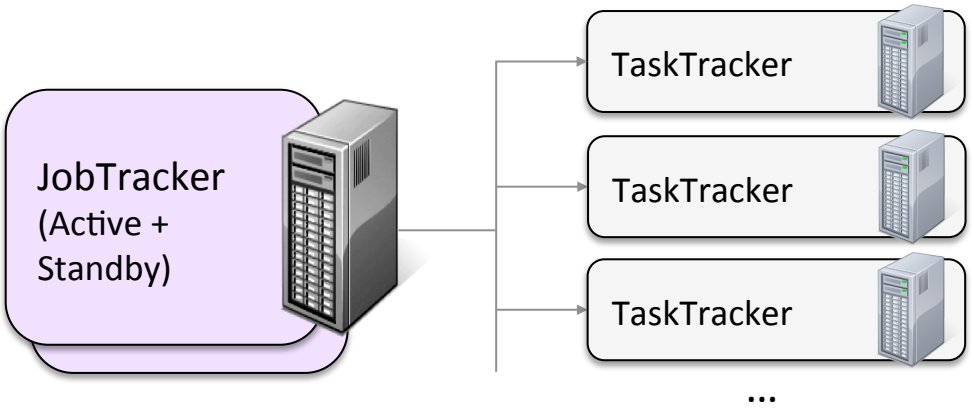


MapReduce v1 and v2 (1)

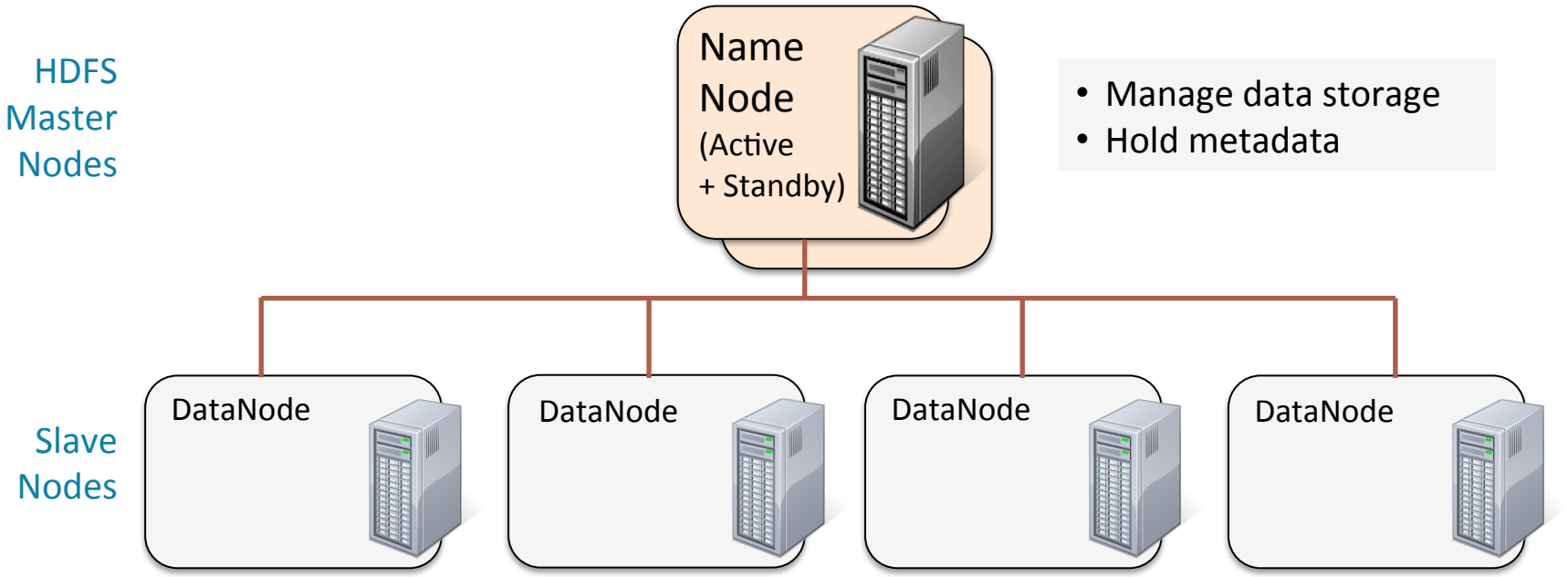
- **MapReduce v1 ("MRv1" or "Classic MapReduce")**
  - Uses a JobTracker/TaskTracker architecture
  - One JobTracker per cluster – limits cluster size to about 4000 nodes
  - *Slots* on slave nodes designated for Map or Reduce tasks
- **MapReduce v2 ("MRv2")**
  - Built on top of YARN (Yet Another Resource Negotiator)
  - Uses ResourceManager/NodeManager architecture
    - Increases scalability of cluster
  - Node resources can be used for any type of task
    - Improves cluster utilization
    - Support for non-MR jobs

Hadoop Daemons: MapReduce v1

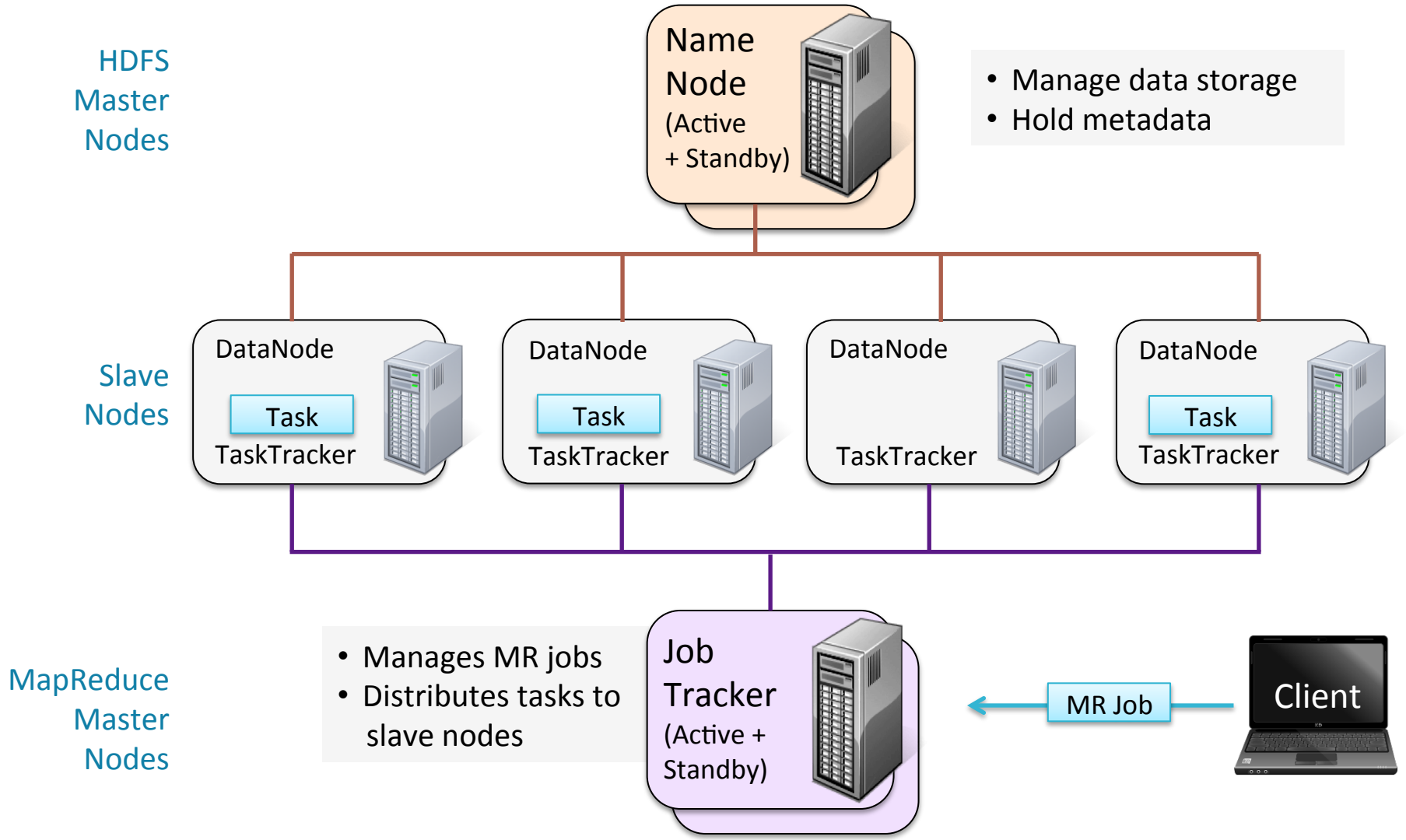
- **MRv1 daemons**
  - JobTracker – one per cluster
    - Manages MapReduce jobs, distributes individual tasks to TaskTrackers
  - TaskTracker – one per slave node
    - Starts and monitors individual Map and Reduce tasks



Basic Cluster Configuration: HDFS

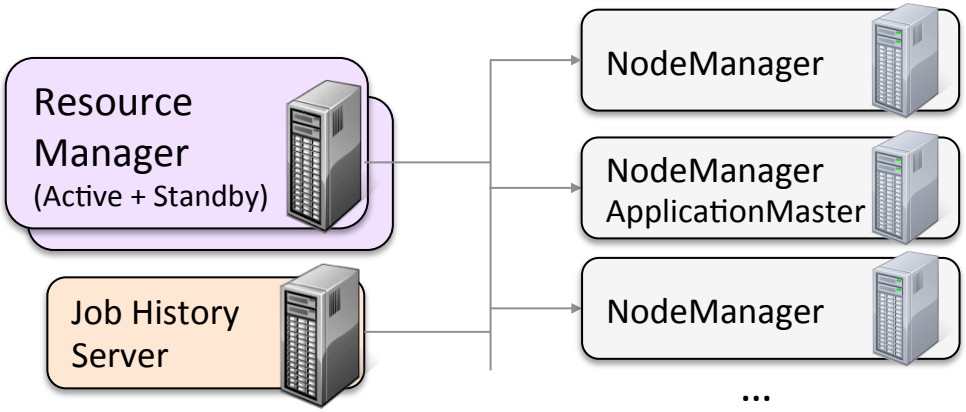


Basic Cluster Configuration: HDFS + MapReduce v1

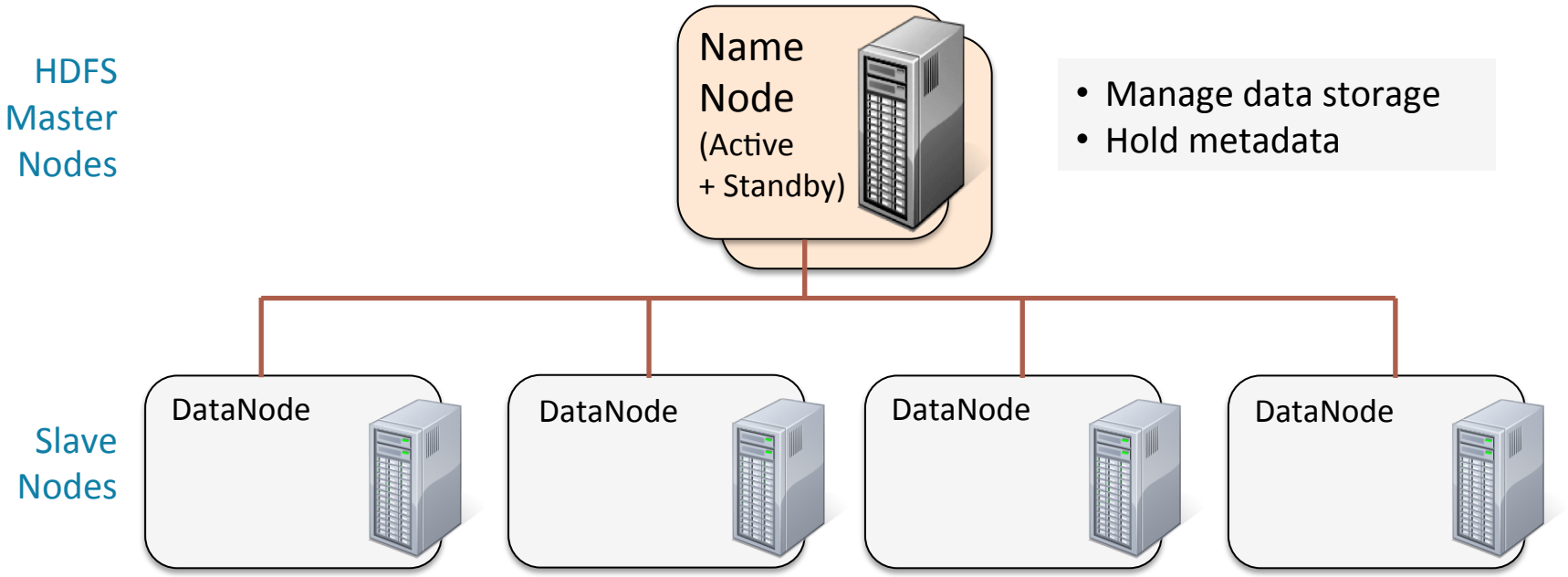


Hadoop Daemons: MapReduce v2

- **MRv2 daemons**
  - ResourceManager – one per cluster
    - Starts ApplicationMasters, allocates resources on slave nodes
  - ApplicationMaster – one per job
    - Requests resources, manages individual Map and Reduce tasks
  - NodeManager – one per slave node
    - Manages resources on individual slave nodes
  - JobHistory – one per cluster
    - Archives jobs' metrics and metadata

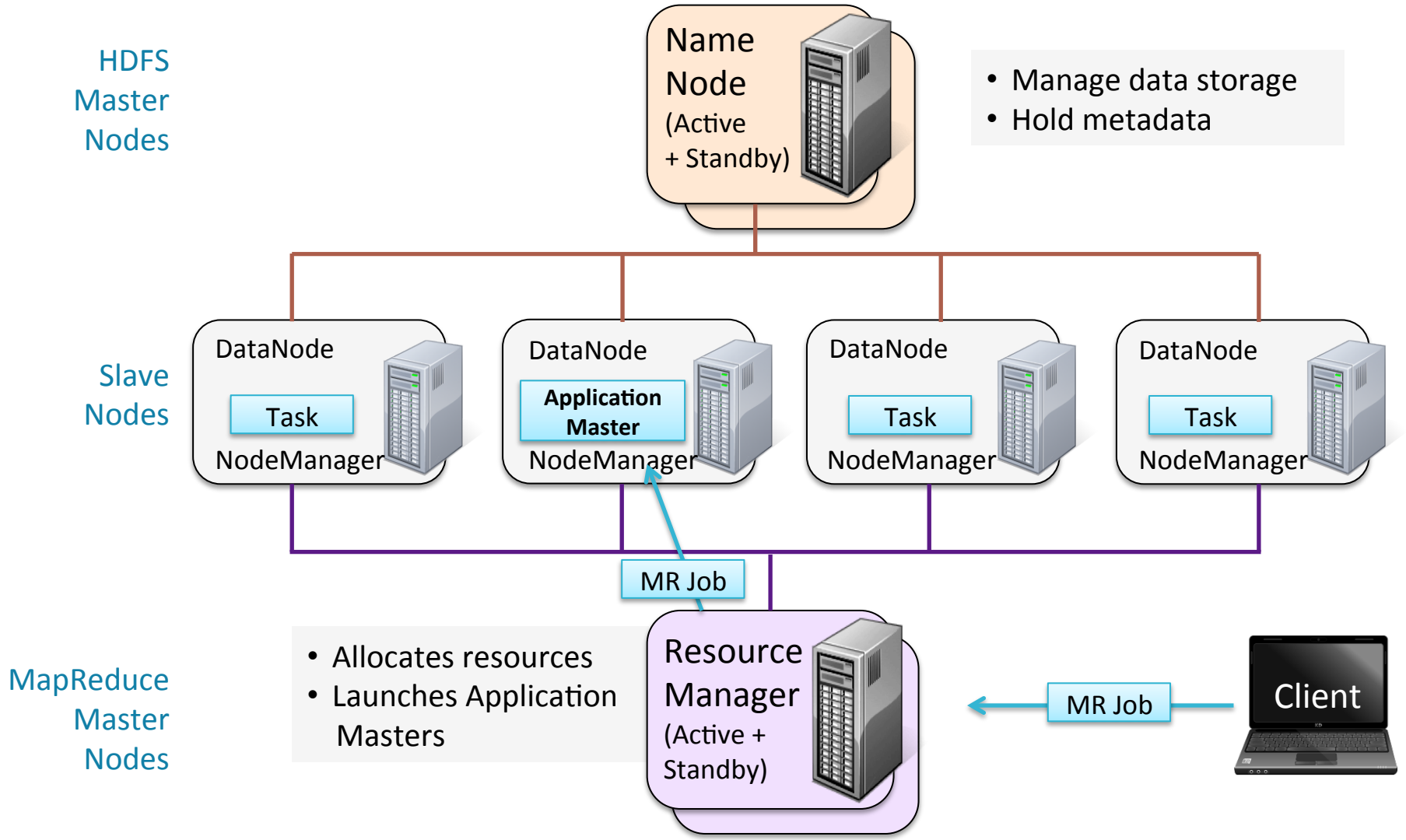


Basic Cluster Configuration: HDFS



Note: This slide is the same as the earlier HDFS slide; there is no change in the HDFS design for YARN/MapReduce v2 because HDFS is the storage side of Hadoop. YARN/MapReduce v2 implement the compute side of Hadoop.

Basic Cluster Configuration: HDFS + MapReduce v2



Chapter Topics

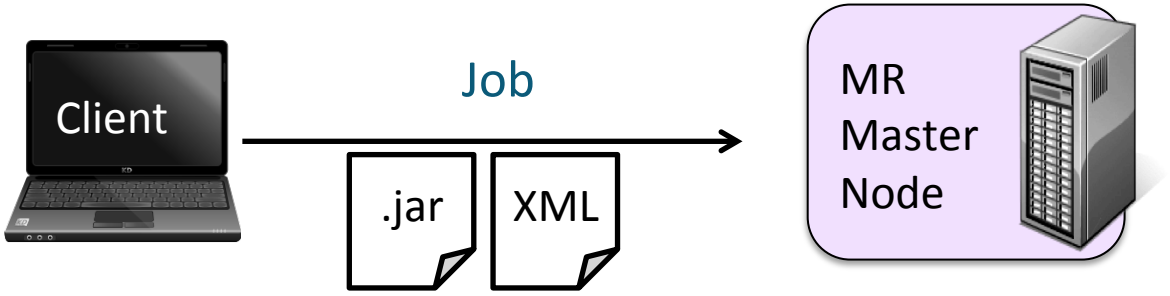
Hadoop Clusters

- Hadoop Cluster Overview
- **Hadoop Jobs and Tasks**

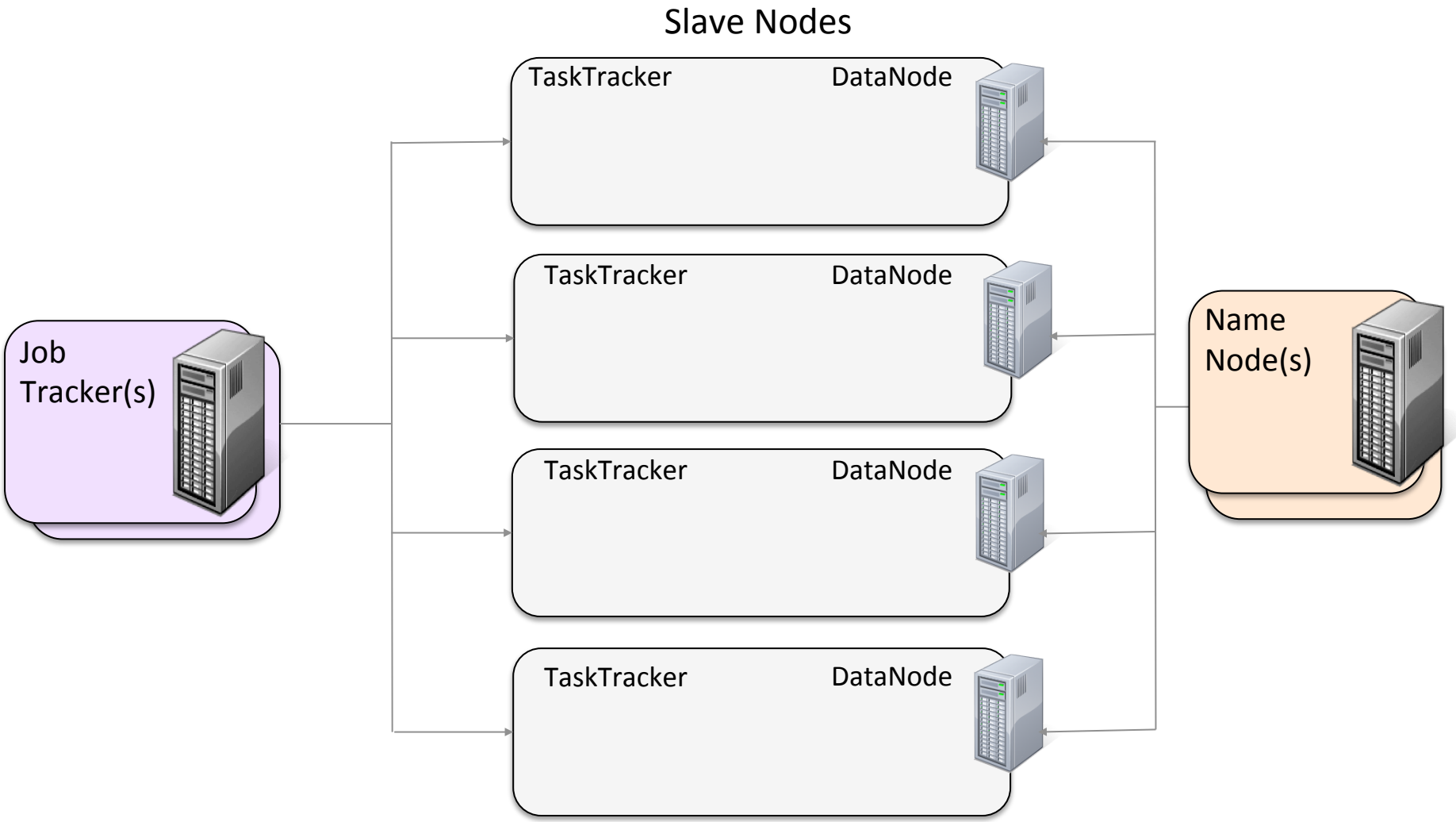
Review - MapReduce Terminology

- **A *job* is a 'full program'**
  - A complete execution of Mappers and Reducers over a dataset
- **A *task* is the execution of a single Mapper or Reducer over a slice of data**
- **A *task attempt* is a particular instance of an attempt to execute a task**
  - There will be at least as many task attempts as there are tasks
  - If a task attempt fails, another will be started by the JobTracker or ApplicationMaster
  - *Speculative execution* (covered later) can also result in more task attempts than completed tasks

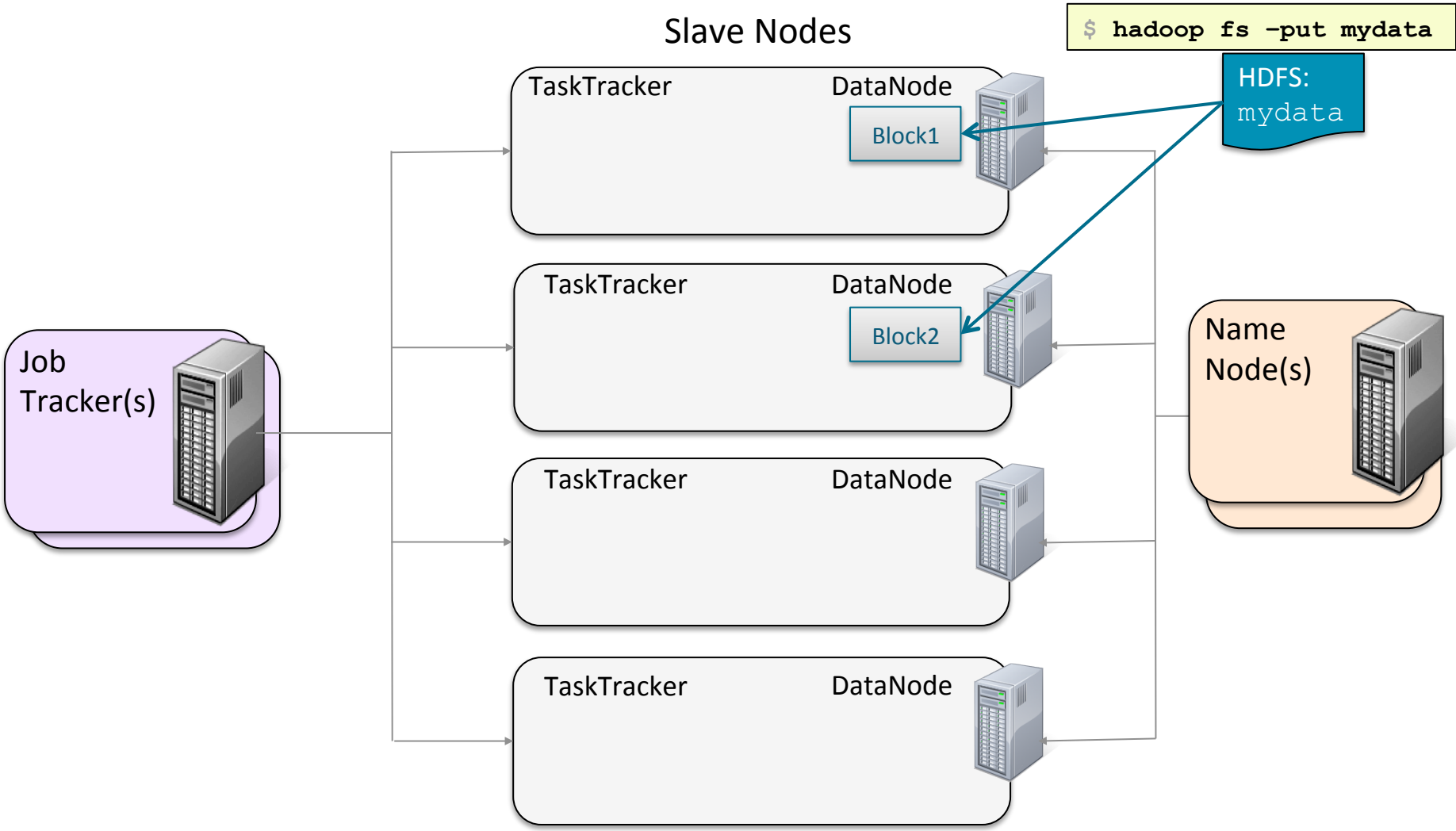
Submitting A Job



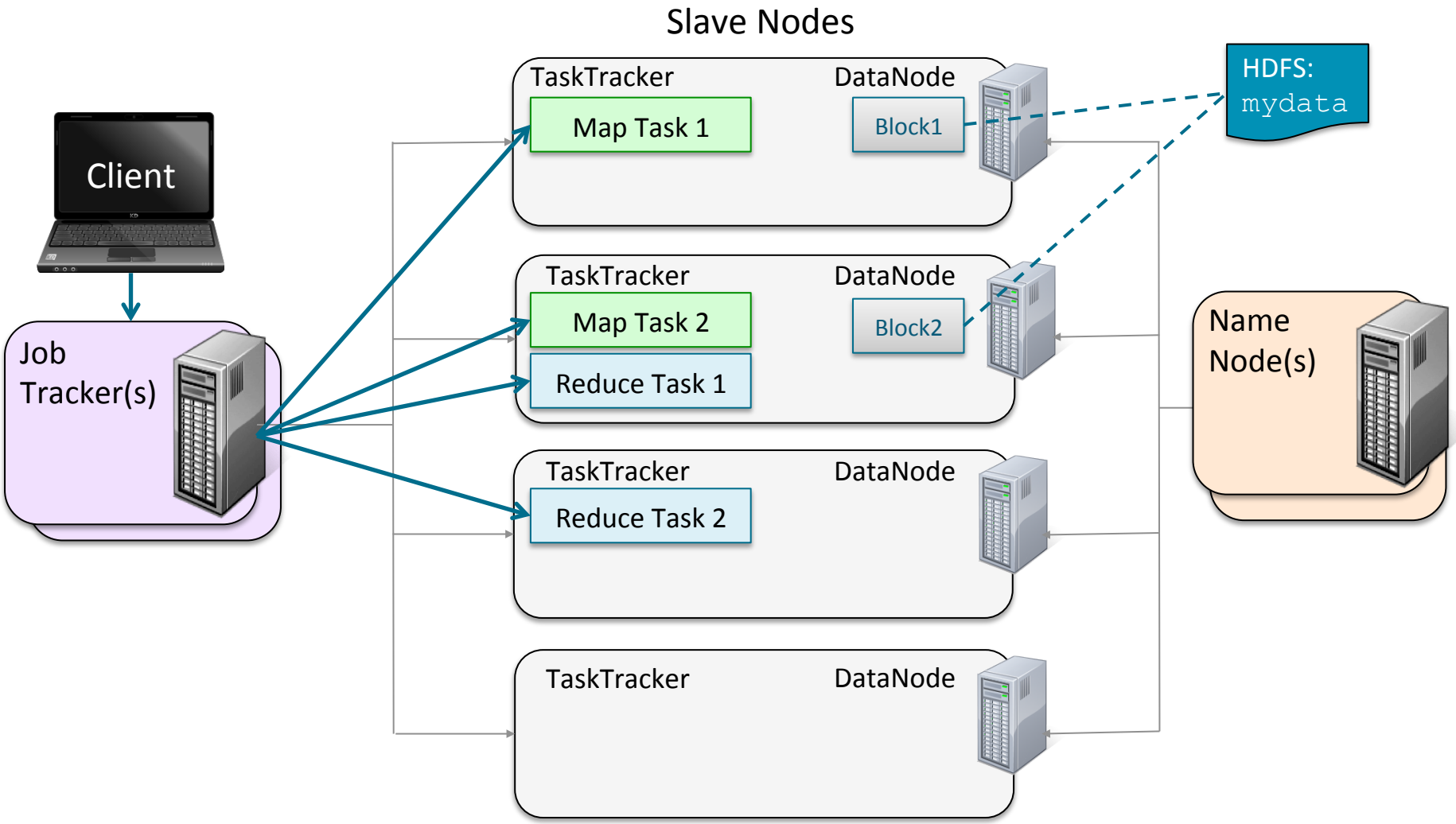
A MapReduce v1 Cluster



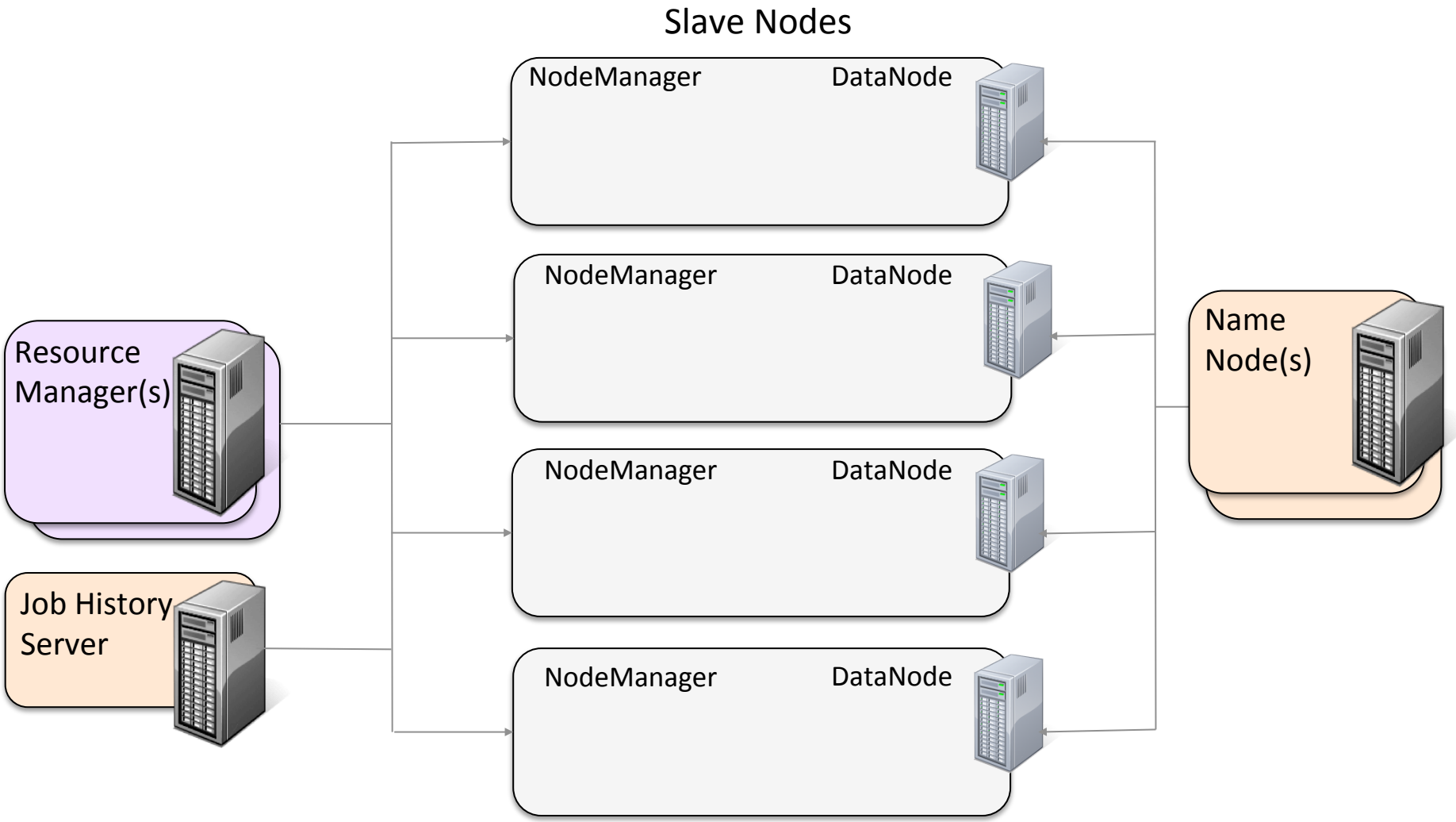
Running a Job on a MapReduce v1 Cluster (1)



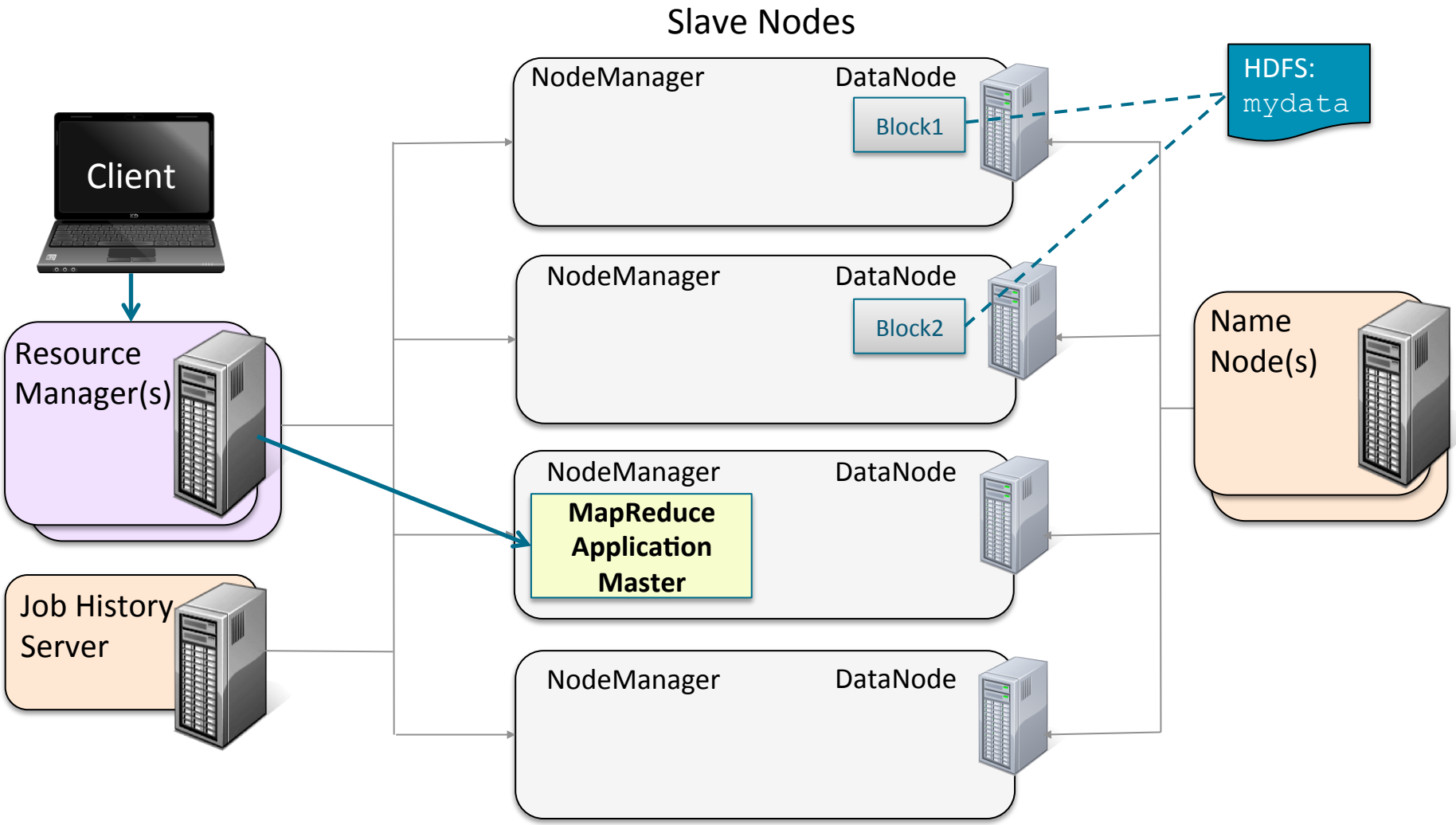
Running a Job on a MapReduce v1 Cluster (2)



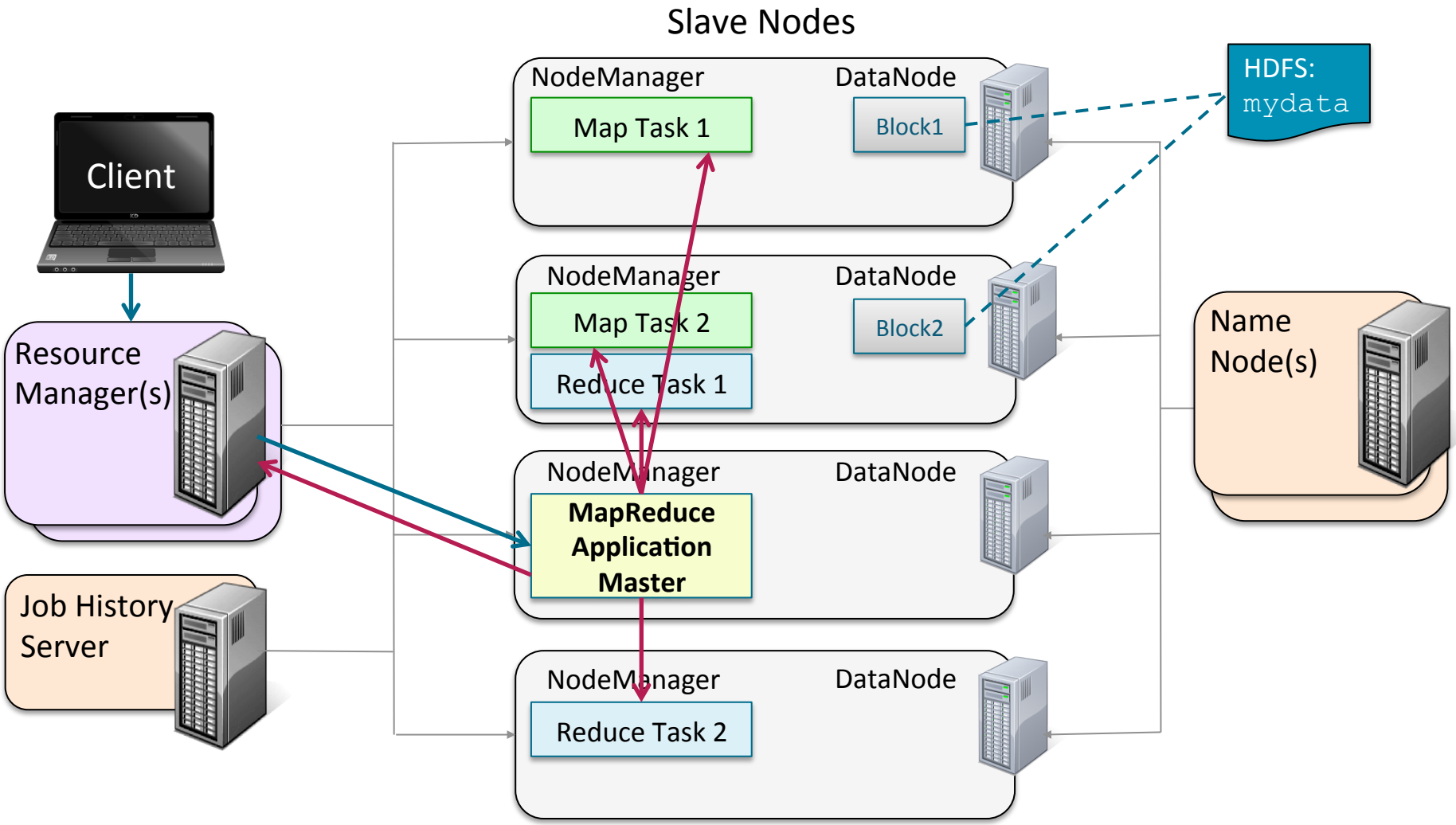
A MapReduce v2 Cluster



Running a Job on a MapReduce v2 Cluster (1)



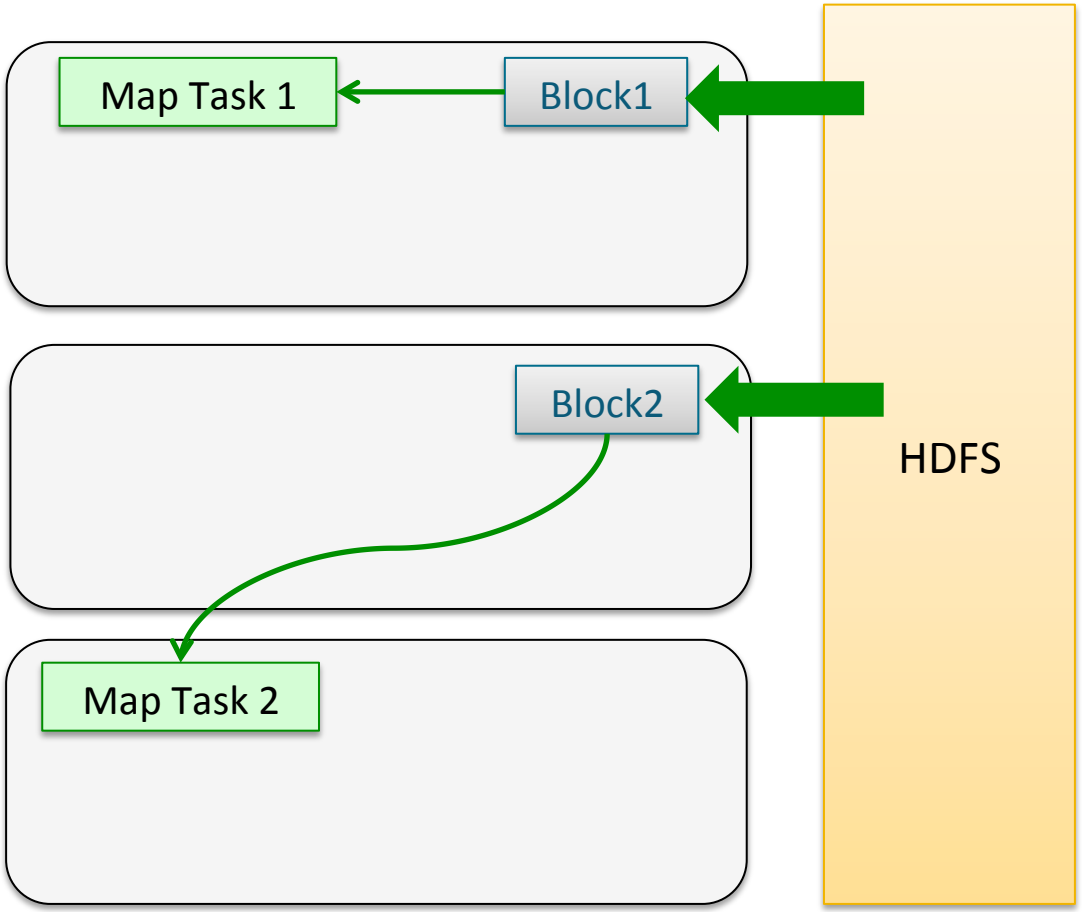
Running a Job on a MapReduce v2 Cluster (2)



Job Data: Mapper Data Locality

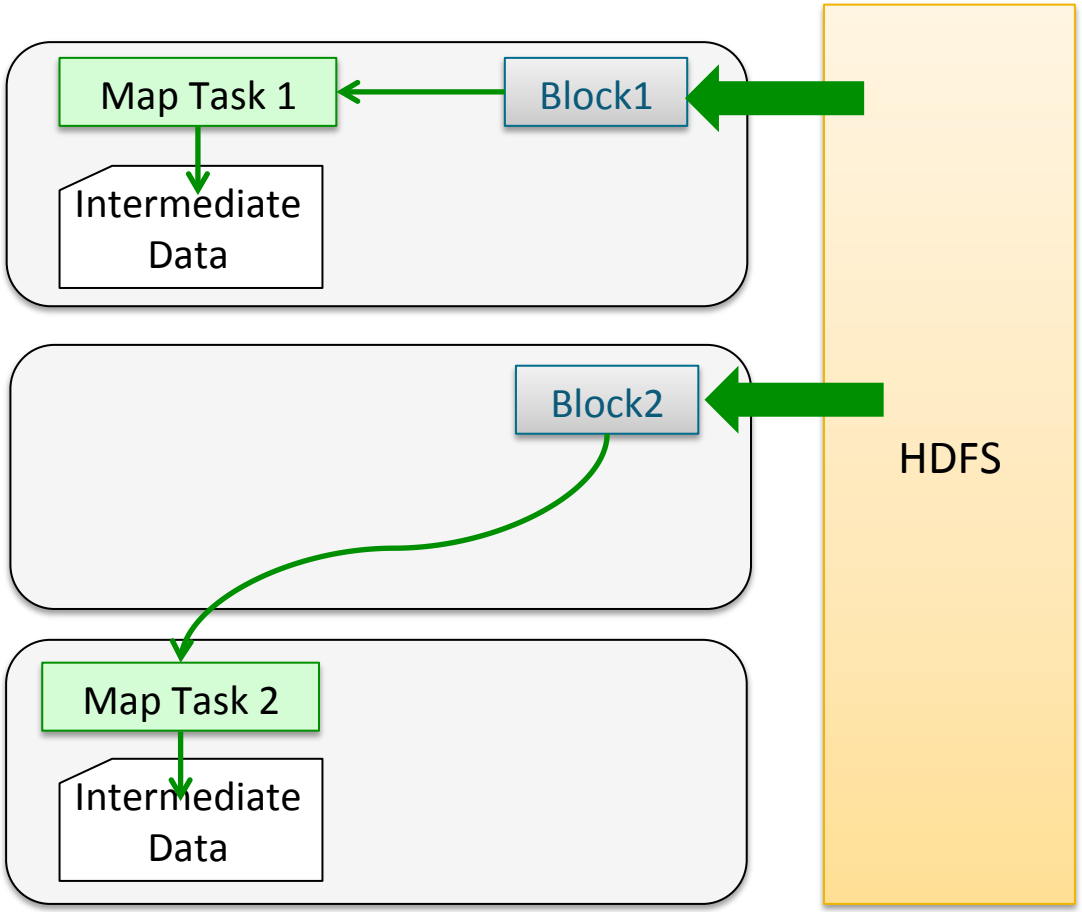
When possible, Map tasks run on a node where the block of data to be processed is stored locally

Otherwise, the Map task will transfer the data across the network and then process that data



Job Data: Intermediate Data

Map task intermediate data is stored on the local disk (not HDFS)

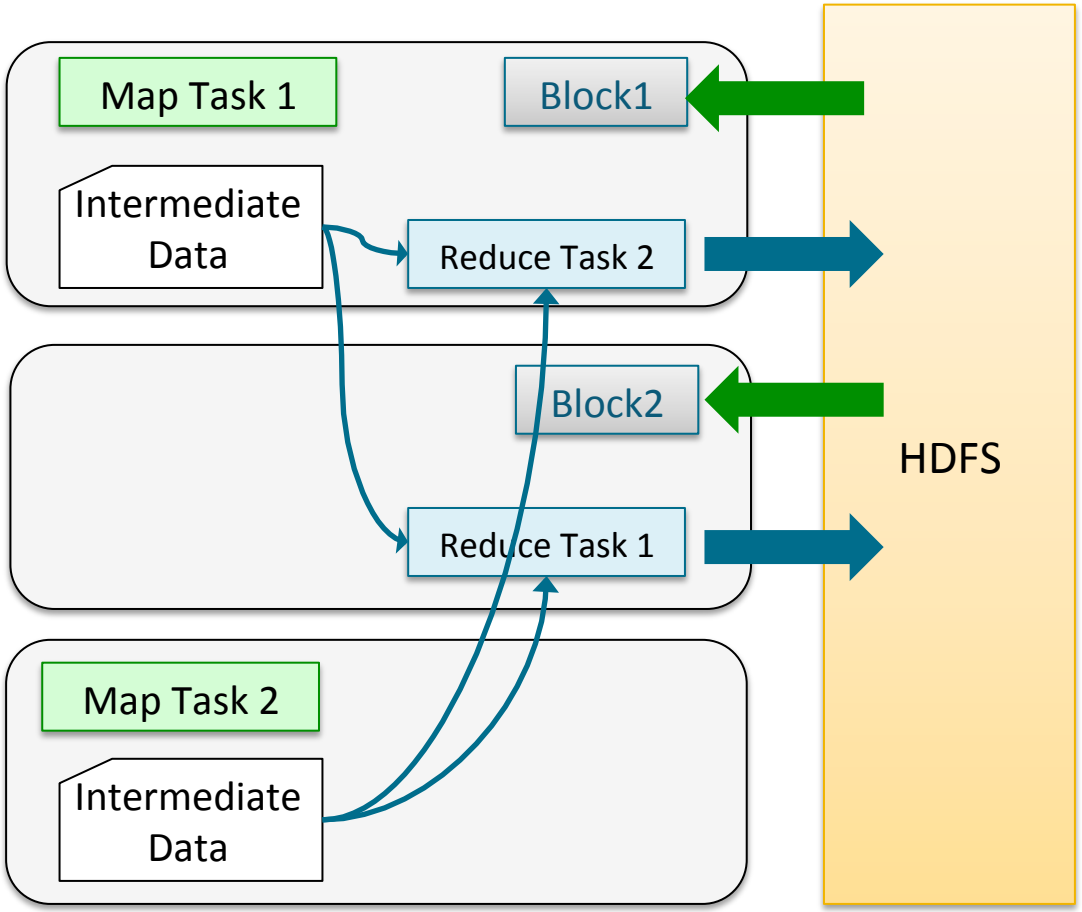


Job Data: Shuffle and Sort

There is no concept of data locality for Reducers

Intermediate data is transferred across the network to the Reducers

Reducers write their output to HDFS



Is Shuffle and Sort a Bottleneck?

- **It appears that the shuffle and sort phase is a bottleneck**
  - The `reduce` method in the Reducers cannot start until all Mappers have finished
- **In practice, Hadoop will start to transfer data from Mappers to Reducers as soon as the Mappers finish work**
  - This avoids a huge amount of data transfer starting as soon as the last Mapper finishes
  - The `reduce` method still does not start until all intermediate data has been transferred and sorted

Is a Slow Mapper a Bottleneck?

- **It is possible for one Map task to run more slowly than the others**
  - Perhaps due to faulty hardware, or just a very slow machine
- **It would appear that this would create a bottleneck**
  - The `reduce` method in the Reducer cannot start until every Mapper has finished
- **Hadoop uses *speculative execution* to mitigate against this**
  - If a Mapper appears to be running significantly more slowly than the others, a new instance of the Mapper will be started on another machine, operating on the same data
    - A new *task attempt* for the same task
  - The results of the first Mapper to finish will be used
  - Hadoop will kill off the Mapper which is still running

Creating and Running a MapReduce Job

- **Write the Mapper and Reducer classes**
- **Write a Driver class that configures the job and submits it to the cluster**
  - Driver classes are covered in the "Writing MapReduce" chapter
- **Compile the Mapper, Reducer, and Driver classes**

```
$ javac -classpath `hadoop classpath` MyMapper.java
  MyReducer.java MyDriver.java
```

- **Create a jar file with the Mapper, Reducer, and Driver classes**

```
$ jar cf MyMR.jar MyMapper.class MyReducer.class
  MyDriver.class
```

- **Run the `hadoop jar` command to submit the job to the Hadoop cluster**

```
$ hadoop jar MyMR.jar MyDriver in_file out_dir
```

Bibliography

The following offer more information on topics discussed in this chapter

- **Full Cloduera documentation available at**
  - **http://cloudera.com/**
- **Reference Hadoop: The Definitive Guide, third edition (TDG 3e) for details on job submission - Chapter 6, "How MapReduce Works", starting on page 189.**