



Cloudera Developer Training for Apache Hadoop



Chapter 1

Introduction

Introduction

 **About This Course**

About Cloudera

Course Logistics

Course Objectives

During this course, you will learn:

- The core technologies of Hadoop
- How HDFS and MapReduce work
- What other projects exist in the Hadoop ecosystem
- How to develop MapReduce jobs
- How Hadoop integrates into the datacenter
- Algorithms for common MapReduce tasks
- How to create large workflows using Oozie

Course Objectives (cont'd)

- How Hive and Pig can be used for rapid application development
- Best practices for developing and debugging MapReduce jobs
- Advanced features of the Hadoop API
- How to handle graph manipulation problems with MapReduce

Course Chapters

Introduction

The Motivation For Hadoop

Hadoop: Basic Contents

Writing a MapReduce Program

The Hadoop Ecosystem

Integrating Hadoop Into The Workflow

Delving Deeper Into The Hadoop API

Common MapReduce Algorithms

An Introduction to Hive and Pig

Debugging MapReduce Programs

Advanced MapReduce Programming

Joining Data Sets in MapReduce Jobs

Graph Manipulation In MapReduce

Conclusion

Appendix: Cloudera Enterprise

Introduction

About This Course



About Cloudera

Course Logistics

About Cloudera

- **Cloudera is “The commercial Hadoop company”**
- **Founded by leading experts on Hadoop from Facebook, Google, Oracle and Yahoo**
- **Provides consulting and training services for Hadoop users**
- **Staff includes committers to virtually all Hadoop projects**

Cloudera Software

- **Cloudera's Distribution including Apache Hadoop (CDH)**
 - A single, easy-to-install package from the Apache Hadoop core repository
 - Includes a stable version of Hadoop, plus critical bug fixes and solid new features from the development version
 - 100% open source
- **Cloudera Manager, Free Edition**
 - The easiest way to deploy a Hadoop cluster
 - Automates installation of Hadoop software
 - Installation, monitoring and configuration is performed from a central machine
 - Manages up to 50 nodes
 - Completely free

Cloudera Enterprise

- **Cloudera Enterprise**

- Complete package of software and support
- Built on top of CDH
- Includes full version of Cloudera Manager
 - Install, manage, and maintain a cluster of any size
 - Integration with LDAP
 - Includes powerful cluster monitoring and auditing tools
 - Resource consumption tracking
 - Proactive health checks
 - Alerting
 - Configuration change audit trails
 - Etc.
 - 24 x 7 support

Cloudera Services

- **Provides consultancy services to many key users of Hadoop**
 - Including AOL Advertising, Samsung, Groupon, NAVTEQ, Trulia, Tynt, RapLeaf, Explorys Medical...
- **Solutions Architects are experts in Hadoop and related technologies**
 - Several are committers to the Apache Hadoop project
- **Provides training in key areas of Hadoop administration and Development**
 - Courses include System Administrator training, Developer training, Hive and Pig training, HBase Training, Essentials for Managers
 - Custom course development available
 - Both public and on-site training available

Introduction

About This Course

About Cloudera

 **Course Logistics**

Logistics

- Course start and end times
- Lunch
- Breaks
- Restrooms
- Can I come in early/stay late?
- Certification

Introductions

- **About your instructor**
- **About you**
 - Experience with Hadoop?
 - Experience as a developer?
 - What programming languages do you use?
 - Expectations from the course?



Chapter 2

The Motivation For Hadoop

Course Chapters

Introduction

The Motivation For Hadoop

Hadoop: Basic Contents

Writing a MapReduce Program

Integrating Hadoop Into The Workflow

Delving Deeper Into The Hadoop API

Common MapReduce Algorithms

An Introduction to Hive and Pig

Practical Development Tips and Techniques

More Advanced MapReduce Programming

Joining Data Sets in MapReduce Jobs

Graph Manipulation In MapReduce

An Introduction to Oozie

Conclusion

Appendix: Cloudera Enterprise

The Motivation For Hadoop

In this chapter you will learn

- **What problems exist with ‘traditional’ large-scale computing systems**
- **What requirements an alternative approach should have**
- **How Hadoop addresses those requirements**

The Motivation For Hadoop



Problems with Traditional Large-Scale Systems

Requirements for a New Approach

Hadoop!

Conclusion

Traditional Large-Scale Computation

- Traditionally, computation has been processor-bound
 - Relatively small amounts of data
 - Significant amount of complex processing performed on that data
- For decades, the primary push was to increase the computing power of a single machine
 - Faster processor, more RAM
- Distributed systems evolved to allow developers to use multiple machines for a single job
 - MPI
 - PVM
 - Condor



MPI: Message Passing Interface

PVM: Parallel Virtual Machine

Distributed Systems: Problems

- **Programming for traditional distributed systems is complex**
 - Data exchange requires synchronization
 - Finite bandwidth is available
 - Temporal dependencies are complicated
 - It is difficult to deal with partial failures of the system
- **Ken Arnold, CORBA designer:**
 - “Failure is the defining difference between distributed and local programming, so you have to design distributed systems with the expectation of failure”
 - Developers spend more time designing for failure than they do actually working on the problem itself

CORBA: Common Object Request Broker Architecture

Distributed Systems: Data Storage

- Typically, data for a distributed system is stored on a SAN
- At compute time, data is copied to the compute nodes
- Fine for relatively limited amounts of data

SAN: Storage Area Network

The Data-Driven World

- **Modern systems have to deal with far more data than was the case in the past**
 - Organizations are generating huge amounts of data
 - That data has inherent value, and cannot be discarded
- **Examples:**
 - Facebook – over 70Pb of data
 - eBay – over 5Pb of data
- **Many organizations are generating data at a rate of terabytes per day**

Data Becomes the Bottleneck

- **Moore's Law has held firm for over 40 years**
 - Processing power doubles every two years
 - Processing speed is no longer the problem
- **Getting the data to the processors becomes the bottleneck**
- **Quick calculation**
 - Typical disk data transfer rate: 75MB/sec
 - Time taken to transfer 100GB of data to the processor: approx 22 minutes!
 - Assuming sustained reads
 - Actual time will be worse, since most servers have less than 100GB of RAM available
- **A new approach is needed**

The Motivation For Hadoop

Problems with Traditional Large-Scale Systems



Requirements for a New Approach

Hadoop!

Conclusion

Partial Failure Support

- **The system must support partial failure**
 - Failure of a component should result in a graceful degradation of application performance
 - Not complete failure of the entire system

Data Recoverability

- **If a component of the system fails, its workload should be assumed by still-functioning units in the system**
 - Failure should not result in the loss of any data

Component Recovery

- **If a component of the system fails and then recovers, it should be able to rejoin the system**
 - Without requiring a full restart of the entire system

Consistency

- Component failures during execution of a job should not affect the outcome of the job

Scalability

- **Adding load to the system should result in a graceful decline in performance of individual jobs**
 - Not failure of the system
- **Increasing resources should support a proportional increase in load capacity**

The Motivation For Hadoop

Problems with Traditional Large-Scale Systems

Requirements for a New Approach



Hadoop!

Conclusion

Hadoop's History

- **Hadoop is based on work done by Google in the late 1990s/early 2000s**
 - Specifically, on papers describing the Google File System (GFS) published in 2003, and MapReduce published in 2004
- **This work takes a radical new approach to the problem of distributed computing**
 - Meets all the requirements we have for reliability, scalability etc
- **Core concept: distribute the data as it is initially stored in the system**
 - Individual nodes can work on data local to those nodes
 - No data transfer over the network is required for initial processing

Core Hadoop Concepts

- **Applications are written in high-level code**
 - Developers do not worry about network programming, temporal dependencies etc
- **Nodes talk to each other as little as possible**
 - Developers should not write code which communicates between nodes
 - ‘Shared nothing’ architecture
- **Data is spread among machines in advance**
 - Computation happens where the data is stored, wherever possible
 - Data is replicated multiple times on the system for increased availability and reliability

Hadoop: Very High-Level Overview

- When data is loaded into the system, it is split into ‘blocks’
 - Typically 64MB or 128MB
- Map tasks (the first part of the MapReduce system) work on relatively small portions of data
 - Typically a single block
- A master program allocates work to nodes such that a Map task will work on a block of data stored locally on that node whenever possible
 - Many nodes work in parallel, each on their own part of the overall dataset

Fault Tolerance

- If a node fails, the master will detect that failure and re-assign the work to a different node on the system
- Restarting a task does not require communication with nodes working on other portions of the data
- If a failed node restarts, it is automatically added back to the system and assigned new tasks
- If a node appears to be running slowly, the master can redundantly execute another instance of the same task
 - Results from the first to finish will be used
 - Known as ‘speculative execution’

The Motivation For Hadoop

Problems with Traditional Large-Scale Systems

Requirements for a New Approach

Hadoop!



Conclusion

The Motivation For Hadoop

In this chapter you have learned

- **What problems exist with ‘traditional’ large-scale computing systems**
- **What requirements an alternative approach should have**
- **How Hadoop addresses those requirements**



Chapter 3

Hadoop: Basic Concepts

Course Chapters

- Introduction
- The Motivation For Hadoop
- Hadoop: Basic Contents**
- Writing a MapReduce Program
- Integrating Hadoop Into The Workflow
- Delving Deeper Into The Hadoop API
- Common MapReduce Algorithms
- An Introduction to Hive and Pig
- Practical Development Tips and Techniques
- More Advanced MapReduce Programming
- Joining Data Sets in MapReduce Jobs
- Graph Manipulation In MapReduce
- An Introduction to Oozie
- Conclusion
- Appendix: Cloudera Enterprise

Hadoop: Basic Concepts

In this chapter you will learn

- What Hadoop is
- What features the Hadoop Distributed File System (HDFS) provides
- The concepts behind MapReduce
- How a Hadoop cluster operates
- What other Hadoop Ecosystem projects exist

Hadoop: Basic Concepts



What Is Hadoop?

The Hadoop Distributed File System (HDFS)

Hands-On Exercise: Using HDFS

How MapReduce works

Hands-On Exercise: Running a MapReduce job

Anatomy of a Hadoop Cluster

Other Ecosystem Projects

Conclusion

The Hadoop Project

- **Hadoop is an open-source project overseen by the Apache Software Foundation**
- **Originally based on papers published by Google in 2003 and 2004**
- **Hadoop committers work at several different organizations**
 - Including Cloudera, Yahoo!, Facebook

Hadoop Components

- **Hadoop consists of two core components**
 - The Hadoop Distributed File System (HDFS)
 - MapReduce
- **There are many other projects based around core Hadoop**
 - Often referred to as the ‘Hadoop Ecosystem’
 - Pig, Hive, HBase, Flume, Oozie, Sqoop, etc
 - Many are discussed later in the course
- **A set of machines running HDFS and MapReduce is known as a *Hadoop Cluster***
 - Individual machines are known as *nodes*
 - A cluster can have as few as one node, as many as several thousands
 - More nodes = better performance!

Hadoop Components: HDFS

- **HDFS, the Hadoop Distributed File System, is responsible for storing data on the cluster**
- **Data is split into blocks and distributed across multiple nodes in the cluster**
 - Each block is typically 64MB or 128MB in size
- **Each block is replicated multiple times**
 - Default is to replicate each block three times
 - Replicas are stored on different nodes
 - This ensures both reliability and availability

Hadoop Components: MapReduce

- **MapReduce is the system used to process data in the Hadoop cluster**
- **Consists of two phases: Map, and then Reduce**
 - Between the two is a stage known as the *shuffle and sort*
- **Each Map task operates on a discrete portion of the overall dataset**
 - Typically one HDFS block of data
- **After all Maps are complete, the MapReduce system distributes the intermediate data to nodes which perform the Reduce phase**
 - Much more on this later!

Hadoop: Basic Concepts

What Is Hadoop?

→ **The Hadoop Distributed File System (HDFS)**

Hands-On Exercise: Using HDFS

How MapReduce works

Hands-On Exercise: Running a MapReduce job

Anatomy of a Hadoop Cluster

Other Ecosystem Projects

Conclusion

HDFS Basic Concepts

- **HDFS is a filesystem written in Java**
 - Based on Google's GFS
- **Sits on top of a native filesystem**
 - ext3, ext4, xfs, etc
- **Provides redundant storage for massive amounts of data**
 - Using cheap, unreliable computers

HDFS Basic Concepts (cont'd)

- **HDFS performs best with a ‘modest’ number of large files**
 - Millions, rather than billions, of files
 - Each file typically 100Mb or more
- **Files in HDFS are ‘write once’**
 - No random writes to files are allowed
 - Append support is included in Cloudera’s Distribution including Apache Hadoop (CDH) for HBase reliability
 - Not recommended for general use
- **HDFS is optimized for large, streaming reads of files**
 - Rather than random reads

How Files Are Stored

- **Files are split into blocks**
 - Each block is usually 64MB or 128MB
- **Data is distributed across many machines at load time**
 - Different blocks from the same file will be stored on different machines
 - This provides for efficient MapReduce processing (see later)
- **Blocks are replicated across multiple machines, known as *DataNodes***
 - Default replication is three-fold
 - i.e., each block exists on three different machines
- **A master node called the *NameNode* keeps track of which blocks make up a file, and where those blocks are located**
 - Known as the *metadata*

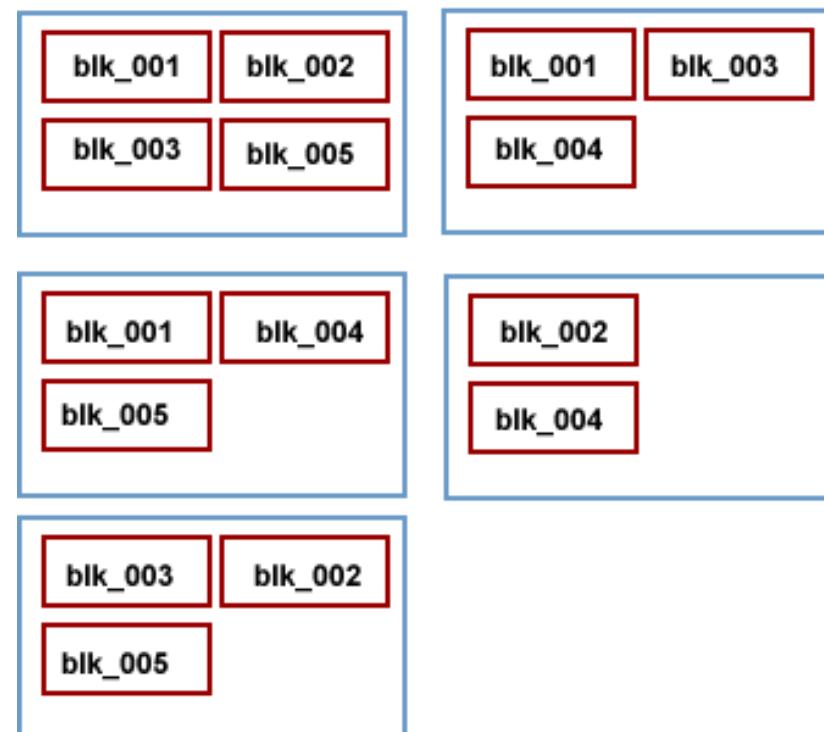
How Files Are Stored: Example

- **NameNode holds metadata for the two files (Foo.txt and Bar.txt)**
- **DataNodes hold the actual blocks**
 - Each block will be 64MB or 128MB in size
 - Each block is replicated three times on the cluster

NameNode

```
Foo.txt: blk_001, blk_002, blk_003  
Bar.txt: blk_004, blk_005
```

DataNodes



More On The HDFS NameNode

- **The NameNode daemon must be running at all times**
 - If the NameNode stops, the cluster becomes inaccessible
 - Your system administrator will take care to ensure that the NameNode hardware is reliable!
- **The NameNode holds all of its metadata in RAM for fast access**
 - It keeps a record of changes on disk for crash recovery
- **A separate daemon known as the *Secondary NameNode* takes care of some housekeeping tasks for the NameNode**
 - Be careful: The Secondary NameNode is *not* a backup NameNode!

HDFS: Points To Note

- Although files are split into 64MB or 128MB blocks, if a file is smaller than this the full 64MB/128MB will not be used
- Blocks are stored as standard files on the DataNodes, in a set of directories specified in Hadoop's configuration files
 - This will be set by the system administrator
- Without the metadata on the NameNode, there is no way to access the files in the HDFS cluster
- When a client application wants to read a file:
 - It communicates with the NameNode to determine which blocks make up the file, and which DataNodes those blocks reside on
 - It then communicates directly with the DataNodes to read the data
 - The NameNode will not be a bottleneck

Accessing HDFS

- Applications can read and write HDFS files directly via the Java API
 - Covered later in the course
- Typically, files are created on a local filesystem and must be moved into HDFS
- Likewise, files stored in HDFS may need to be moved to a machine's local filesystem
- Access to HDFS from the command line is achieved with the `hadoop fs` command

hadoop fs Examples

- **Copy file `foo.txt` from local disk to the user's directory in HDFS**

```
hadoop fs -copyFromLocal foo.txt foo.txt
```

- This will copy the file to `/user/username/foo.txt`

- **Get a directory listing of the user's home directory in HDFS**

```
hadoop fs -ls
```

- **Get a directory listing of the HDFS root directory**

```
hadoop fs -ls /
```

hadoop fs Examples (cont'd)

- **Display the contents of the HDFS file /user/fred/bar.txt**

```
hadoop fs -cat /user/fred/bar.txt
```

- **Move that file to the local disk, named as baz.txt**

```
hadoop fs -copyToLocal /user/fred/bar.txt baz.txt
```

- **Create a directory called input under the user's home directory**

```
hadoop fs -mkdir input
```

hadoop fs Examples (cont'd)

- Delete the directory `input_old` and all its contents

```
hadoop fs -rmdir input_old
```

Hadoop: Basic Concepts

What Is Hadoop?

The Hadoop Distributed File System (HDFS)

 **Hands-On Exercise: Using HDFS**

How MapReduce works

Hands-On Exercise: Running a MapReduce job

Anatomy of a Hadoop Cluster

Other Ecosystem Projects

Conclusion

Aside: The Training Virtual Machine

- During this course, you will perform numerous Hands-On Exercises using the Cloudera Training Virtual Machine (VM)
- The VM has Hadoop installed in *pseudo-distributed mode*
 - This essentially means that it is a cluster comprised of a single node
 - Using a pseudo-distributed cluster is the typical way to test your code before you run it on your full cluster
 - It operates almost exactly like a ‘real’ cluster
 - A key difference is that the data replication factor is set to 1, not 3

Hands-On Exercise: Using HDFS

- In this Hands-On Exercise you will gain familiarity with manipulating files in HDFS
- Please refer to the Hands-On Exercise Manual

Hadoop: Basic Concepts

What Is Hadoop?

The Hadoop Distributed File System (HDFS)

Hands-On Exercise: Using HDFS



How MapReduce works

Hands-On Exercise: Running a MapReduce job

Anatomy of a Hadoop Cluster

Other Ecosystem Projects

Conclusion

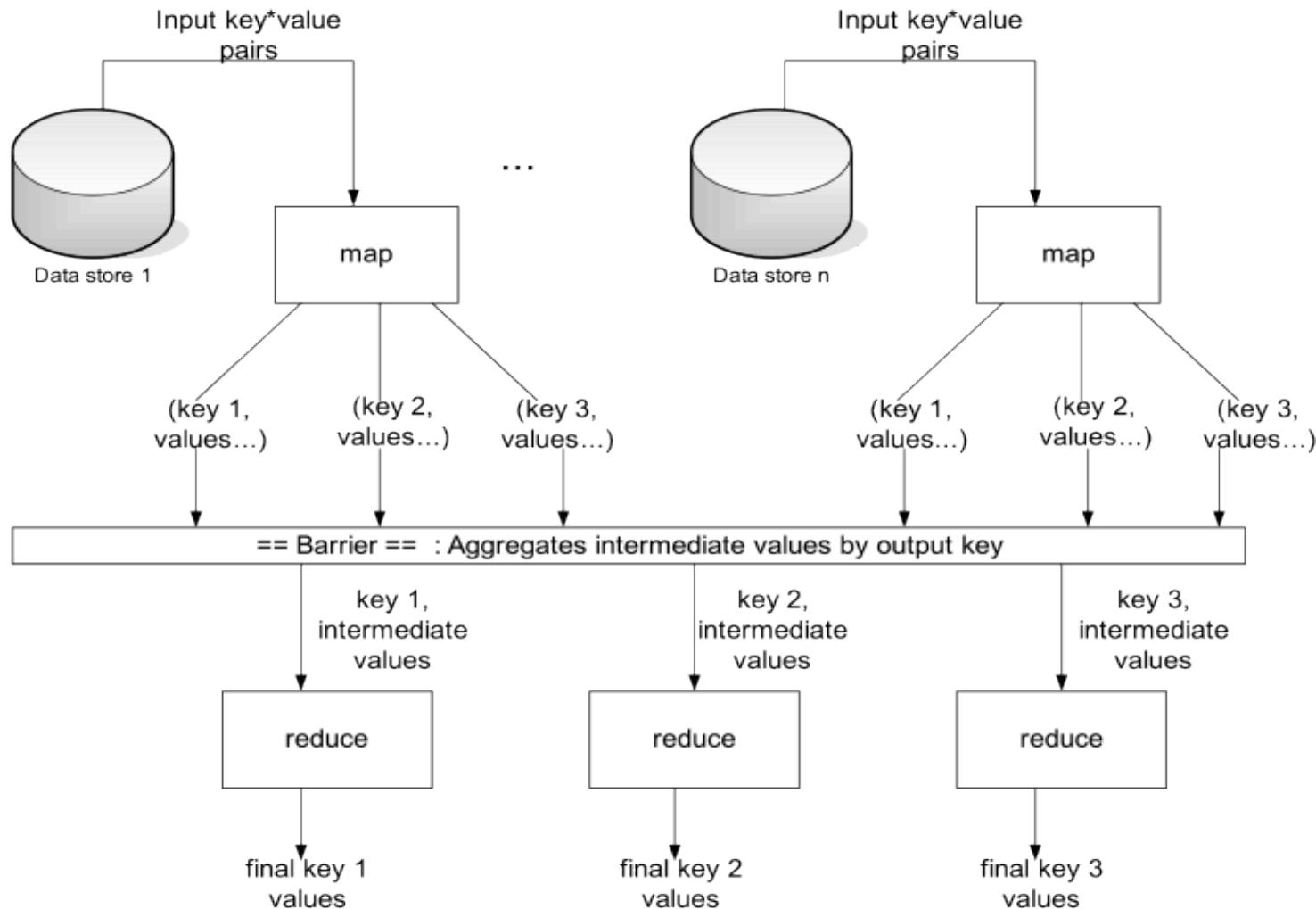
What Is MapReduce?

- **MapReduce is a method for distributing a task across multiple nodes**
- **Each node processes data stored on that node**
 - Where possible
- **Consists of two phases:**
 - Map
 - Reduce

Features of MapReduce

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

MapReduce: The Big Picture



MapReduce: The JobTracker

- MapReduce jobs are controlled by a software daemon known as the *JobTracker*
- The **JobTracker** resides on a ‘master node’
 - Clients submit MapReduce jobs to the JobTracker
 - The JobTracker assigns Map and Reduce tasks to other nodes on the cluster
 - These nodes each run a software daemon known as the *TaskTracker*
 - The TaskTracker is responsible for actually instantiating the Map or Reduce task, and reporting progress back to the JobTracker

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
 - *Speculative execution* (see later) can also result in more task attempts than completed tasks

MapReduce: The Mapper

- Hadoop attempts to ensure that Mappers run on nodes which hold their portion of the data locally, to avoid network traffic
 - Multiple Mappers run in parallel, each processing a portion of the input data
- The Mapper reads data in the form of key/value pairs
- It outputs zero or more key/value pairs

```
map(in_key, in_value) ->  
    (inter_key, inter_value) list
```

MapReduce: The Mapper (cont'd)

- **The Mapper may use or completely ignore the input key**
 - For example, a standard pattern is to read a 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**

Example Mapper: Upper Case Mapper

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

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

```
('foo', 'bar') -> ('FOO', 'BAR')  
('foo', 'other') -> ('FOO', 'OTHER')  
('baz', 'more data') -> ('BAZ', 'MORE DATA')
```

Example Mapper: Explode Mapper

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

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

```
('foo', 'bar') -> ('foo', 'b'), ('foo', 'a'),  
                      ('foo', 'r')
```

```
('baz', 'other') -> ('baz', 'o'), ('baz', 't'),  
                        ('baz', 'h'), ('baz', 'e'),  
                        ('baz', 'r')
```

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)
```

```
('foo', 7) -> ('foo', 7)  
('baz', 10) -> nothing
```

Example Mapper: Changing Keyspaces

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

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

```
('foo', 'bar') -> (3, 'bar')  
('baz', 'other') -> (5, 'other')  
('foo', 'abracadabra') -> (11, 'abracadabra')
```

MapReduce: 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
 - This is specified as part of the job configuration (see later)
 - 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 to the Reducer in sorted key order
 - This step is known as the ‘shuffle and sort’
- The Reducer outputs zero or more final key/value pairs
 - These are written to HDFS
 - In practice, the Reducer usually emits a single key/value pair for each input key

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)
```

```
('foo', [9, 3, -17, 44]) -> ('foo', 39)  
('bar', [123, 100, 77]) -> ('bar', 300)
```

Example Reducer: Identity Reducer

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

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

```
('foo', [9, 3, -17, 44]) -> ('foo', 9), ('foo', 3),  
                               ('foo', -17), ('foo', 44)  
('bar', [123, 100, 77]) -> ('bar', 123), ('bar', 100),  
                               ('bar', 77)
```

MapReduce: Data Localization

- Whenever possible, Hadoop will attempt to ensure that a Map task on a node is working on a block of data stored locally on that node via HDFS
- If this is not possible, the Map task will have to transfer the data across the network as it processes that data
- Once the Map tasks have finished, data is then transferred across the network to the Reducers
 - Although the Reducers may run on the same physical machines as the Map tasks, there is no concept of data locality for the Reducers
 - All Mappers will, in general, have to communicate with all Reducers

MapReduce: 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 the Mappers finish work
 - This mitigates against a huge amount of data transfer starting as soon as the last Mapper finishes

MapReduce: 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
 - The results of the first Mapper to finish will be used
 - Hadoop will kill off the Mapper which is still running

MapReduce: The Combiner

- Often, Mappers produce large amounts of intermediate data
 - That data must be passed to the Reducers
 - This can result in a lot of network traffic
- It is often possible to specify a **Combiner**
 - Like a ‘mini-Reduce’
 - Runs locally on a single Mapper’s output
 - Output from the Combiner is sent to the Reducers
- Combiner and Reducer code are often identical
 - Technically, this is possible if the operation performed is commutative and associative

MapReduce Example: Word Count

- Count the number of occurrences of each word in a large amount of input data
 - This is the ‘hello world’ of MapReduce programming

```
map(String input_key, String input_value)
    foreach word w in input_value:
        emit(w, 1)
```

```
reduce(String output_key,
           Iterator<int> intermediate_vals)
    set count = 0
    foreach v in intermediate_vals:
        count += v
    emit(output_key, count)
```

MapReduce Example: Word Count (cont'd)

- Input to the Mapper:

```
(3414, 'the cat sat on the mat')
(3437, 'the aardvark sat on the sofa')
```

- Output from the Mapper:

```
('the', 1), ('cat', 1), ('sat', 1), ('on', 1),
('the', 1), ('mat', 1), ('the', 1), ('aardvark', 1),
('sat', 1), ('on', 1), ('the', 1), ('sofa', 1)
```

MapReduce Example: Word Count (cont'd)

- Intermediate data sent to the Reducer:

```
('aardvark', [1])
('cat', [1])
('mat', [1])
('on', [1, 1])
('sat', [1, 1])
('sofa', [1])
('the', [1, 1, 1, 1])
```

- Final Reducer Output:

```
('aardvark', 1)
('cat', 1)
('mat', 1)
('on', 2)
('sat', 2)
('sofa', 1)
('the', 4)
```

Word Count With Combiner

- A Combiner would reduce the amount of data sent to the Reducer
 - Intermediate data sent to the Reducer after a Combiner using the same code as the Reducer:

```
('aardvark', [1])
('cat', [1])
('mat', [1])
('on', [2])
('sat', [2])
('sofa', [1])
('the', [4])
```

- Combiners decrease the amount of network traffic required during the shuffle and sort phase
 - Often also decrease the amount of work needed to be done by the Reducer

Hadoop: Basic Concepts

What Is Hadoop?

The Hadoop Distributed File System (HDFS)

Hands-On Exercise: Using HDFS

How MapReduce works

 **Hands-On Exercise: Running a MapReduce job**

Anatomy of a Hadoop Cluster

Other Ecosystem Projects

Conclusion

Hands-On Exercise: Running A MapReduce Job

- In this Hands-On Exercise, you will run a MapReduce job on your pseudo-distributed Hadoop cluster
- Please refer to the Hands-On Exercise Manual

Hadoop: Basic Concepts

What Is Hadoop?

The Hadoop Distributed File System (HDFS)

Hands-On Exercise: Using HDFS

How MapReduce works

Hands-On Exercise: Running a MapReduce job

 **Anatomy of a Hadoop Cluster**

Other Ecosystem Projects

Conclusion

Installing A Hadoop Cluster

- **Cluster installation is usually performed by the system administrator, and is outside the scope of this course**
 - Cloudera offers a training course for System Administrators specifically aimed at those responsible for commissioning and maintaining Hadoop clusters
- **However, it's very useful to understand how the component parts of the Hadoop cluster work together**
- **Typically, a developer will configure their machine to run in *pseudo-distributed mode***
 - This effectively creates a single-machine cluster
 - All five Hadoop daemons are running on the same machine
 - Very useful for testing code before it is deployed to the real cluster

Installing A Hadoop Cluster (cont'd)

- **Easiest way to download and install Hadoop, either for a full cluster or in pseudo-distributed mode, is by using Cloudera's Distribution including Apache Hadoop (CDH)**
 - Vanilla Hadoop plus many patches, backports, bugfixes
 - Supplied as a Debian package (for Linux distributions such as Ubuntu), an RPM (for CentOS/RedHat Enterprise Linux), and as a tarball
 - Full documentation available at <http://cloudera.com>

The Five Hadoop Daemons

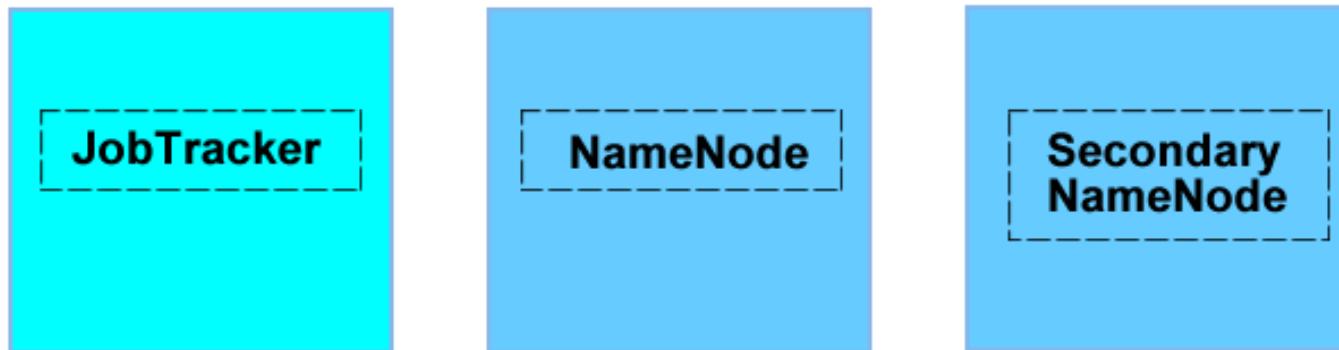
- **Hadoop is comprised of five separate daemons**
- **NameNode**
 - Holds the metadata for HDFS
- **Secondary NameNode**
 - Performs housekeeping functions for the NameNode
 - Is not a backup or hot standby for the NameNode!
- **DataNode**
 - Stores actual HDFS data blocks
- **JobTracker**
 - Manages MapReduce jobs, distributes individual tasks to machines running the...
- **TaskTracker**
 - Responsible for instantiating and monitoring individual Map and Reduce tasks

The Five Hadoop Daemons (cont'd)

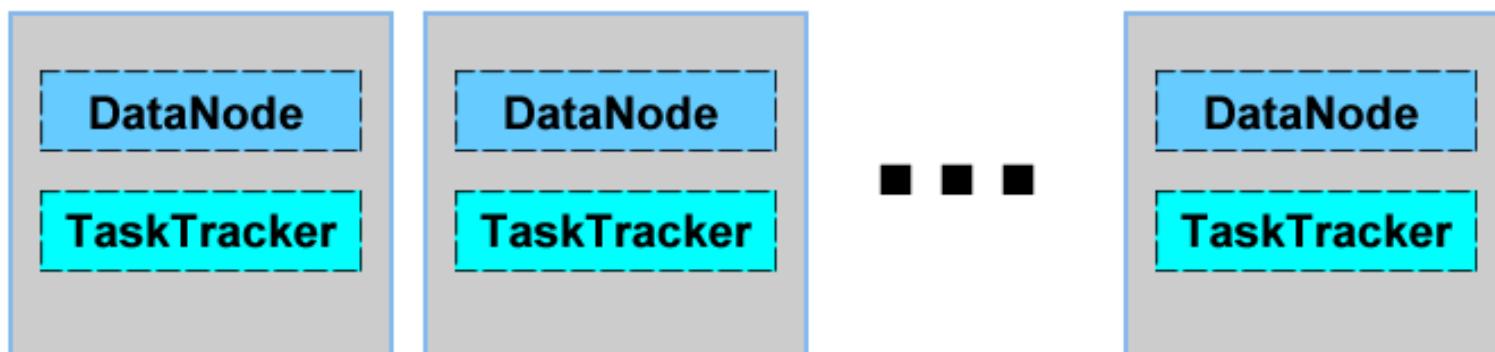
- **Each daemon runs in its own Java Virtual Machine (JVM)**
- **No node on a real cluster will run all five daemons**
 - Although this is technically possible
- **We can consider nodes to be in two different categories:**
 - Master Nodes
 - Run the NameNode, Secondary NameNode, JobTracker daemons
 - Only one of each of these daemons runs on the cluster
 - Slave Nodes
 - Run the DataNode and TaskTracker daemons
 - A slave node will run both of these daemons

Basic Cluster Configuration

Master Nodes



Slave Nodes



Basic Cluster Configuration (cont'd)

- On very small clusters, the NameNode, JobTracker and Secondary NameNode can all reside on a single machine
 - It is typical to put them on separate machines as the cluster grows beyond 20-30 nodes
- Each dotted box on the previous diagram represents a separate Java Virtual Machine (JVM)

Submitting A Job

- When a client submits a job, its configuration information is packaged into an XML file
- This file, along with the .jar file containing the actual program code, is handed to the JobTracker
 - The JobTracker then parcels out individual tasks to TaskTracker nodes
 - When a TaskTracker receives a request to run a task, it instantiates a separate JVM for that task
 - TaskTracker nodes can be configured to run multiple tasks at the same time
 - If the node has enough processing power and memory

Submitting A Job (cont'd)

- The intermediate data is held on the TaskTracker's local disk
- As Reducers start up, the intermediate data is distributed across the network to the Reducers
- Reducers write their final output to HDFS
- Once the job has completed, the TaskTracker can delete the intermediate data from its local disk
 - Note that the intermediate data is not deleted until the entire job completes

Hadoop: Basic Concepts

What Is Hadoop?

The Hadoop Distributed File System (HDFS)

Hands-On Exercise: Using HDFS

How MapReduce works

Hands-On Exercise: Running a MapReduce job

Anatomy of a Hadoop Cluster



Other Ecosystem Projects

Conclusion

Other Ecosystem Projects: Introduction

- **The term ‘Hadoop’ refers to HDFS and MapReduce**
- **Many other projects exist which use Hadoop**
 - Either both HDFS and MapReduce, or just HDFS
- **Many are Apache projects or Apache Incubator projects**
 - Some others are not hosted by the Apache Software Foundation
 - These are often hosted on GitHub or a similar repository
- **We will investigate many of these projects later in the course**
- **Following is an introduction to some of the most significant projects**

Hive

- Hive is an abstraction on top of MapReduce
- Allows users to query data in the Hadoop cluster without knowing Java or MapReduce
- Uses the HiveQL language
 - Very similar to SQL
- The Hive Interpreter runs on a client machine
 - Turns HiveQL queries into MapReduce jobs
 - Submits those jobs to the cluster
- Note: this does *not* turn the cluster into a relational database server!
 - It is still simply running MapReduce jobs
 - Those jobs are created by the Hive Interpreter

Hive (cont'd)

- Sample Hive query:

```
SELECT stock.product, SUM(orders.purchases)
      FROM stock INNER JOIN orders
        ON (stock.id = orders.stock_id)
   WHERE orders.quarter = 'Q1'
 GROUP BY stock.product;
```

- We will investigate Hive in greater detail later in the course

Pig

- **Pig is an alternative abstraction on top of MapReduce**
- **Uses a dataflow scripting language**
 - Called PigLatin
- **The Pig interpreter runs on the client machine**
 - Takes the PigLatin script and turns it into a series of MapReduce jobs
 - Submits those jobs to the cluster
- **As with Hive, nothing ‘magical’ happens on the cluster**
 - It is still simply running MapReduce jobs

Pig (cont'd)

- Sample Pig script:

```
stock = LOAD '/user/fred/stock' AS (id, item);
orders= LOAD '/user/fred/orders' AS (id, cost);
grpds = GROUP orders BY id;
totals = FOREACH grpds GENERATE group,
SUM(orders.cost) AS t;
result = JOIN stock BY id, totals BY group;
DUMP result;
```

- We will investigate Pig in more detail later in the course

Flume and Sqoop

- Flume provides a method to import data into HDFS as it is generated
 - Rather than batch-processing the data later
 - For example, log files from a Web server
- Sqoop provides a method to import data from tables in a relational database into HDFS
 - Does this very efficiently via a Map-only MapReduce job
 - Can also ‘go the other way’
 - Populate database tables from files in HDFS
- We will investigate Flume and Sqoop later in the course

Oozie

- **Oozie allows developers to create a workflow of MapReduce jobs**
 - Including dependencies between jobs
- **The Oozie server submits the jobs to the server in the correct sequence**
- **We will investigate Oozie later in the course**

HBase

- HBase is ‘the Hadoop database’
- A ‘NoSQL’ datastore
- Can store massive amounts of data
 - Gigabytes, Terabytes, even Petabytes of data in a table
- Scales to provide very high write throughput
 - Hundreds of thousands of inserts per second
- Copes well with sparse data
 - Tables can have many thousands of columns
 - Even if most columns are empty for any given row
- Has a very constrained access model
 - Insert a row, retrieve a row, do a full or partial table scan
 - 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
Security	Authentication/Authorization	TBD
Indexes	On arbitrary columns	Row-key only
Max data size	TBs	PB+
Read/write throughput limits	1000s queries/second	Millions of queries/second

Hadoop: Basic Concepts

What Is Hadoop?

The Hadoop Distributed File System (HDFS)

Hands-On Exercise: Using HDFS

How MapReduce works

Hands-On Exercise: Running a MapReduce job

Anatomy of a Hadoop Cluster

Other Ecosystem Projects



Conclusion

Conclusion

In this chapter you have learned

- **What Hadoop is**
- **What features the Hadoop Distributed File System (HDFS) provides**
- **The concepts behind MapReduce**
- **How a Hadoop cluster operates**
- **What other Hadoop Ecosystem projects exist**



Chapter 4

Writing a MapReduce Program

Course Chapters

- Introduction
- The Motivation For Hadoop
- Hadoop: Basic Contents
- Writing a MapReduce Program**
- Integrating Hadoop Into The Workflow
- Delving Deeper Into The Hadoop API
- Common MapReduce Algorithms
- An Introduction to Hive and Pig
- Practical Development Tips and Techniques
- More Advanced MapReduce Programming
- Joining Data Sets in MapReduce Jobs
- Graph Manipulation In MapReduce
- An Introduction to Oozie
- Conclusion
- Appendix: Cloudera Enterprise

Writing a MapReduce Program

In this chapter you will learn

- How to use the Hadoop API to write a MapReduce program in Java
- How to use the Streaming API to write Mappers and Reducers in other languages
- How to use Eclipse to speed up your Hadoop development
- The differences between the Old and New Hadoop APIs

Writing a MapReduce Program



The MapReduce Flow

Examining our Sample MapReduce program

The Driver Code

The Mapper

The Reducer

Hadoop's Streaming API

Using Eclipse for Rapid Development

Hands-On Exercise: Write a MapReduce program

The New MapReduce API

Conclusion

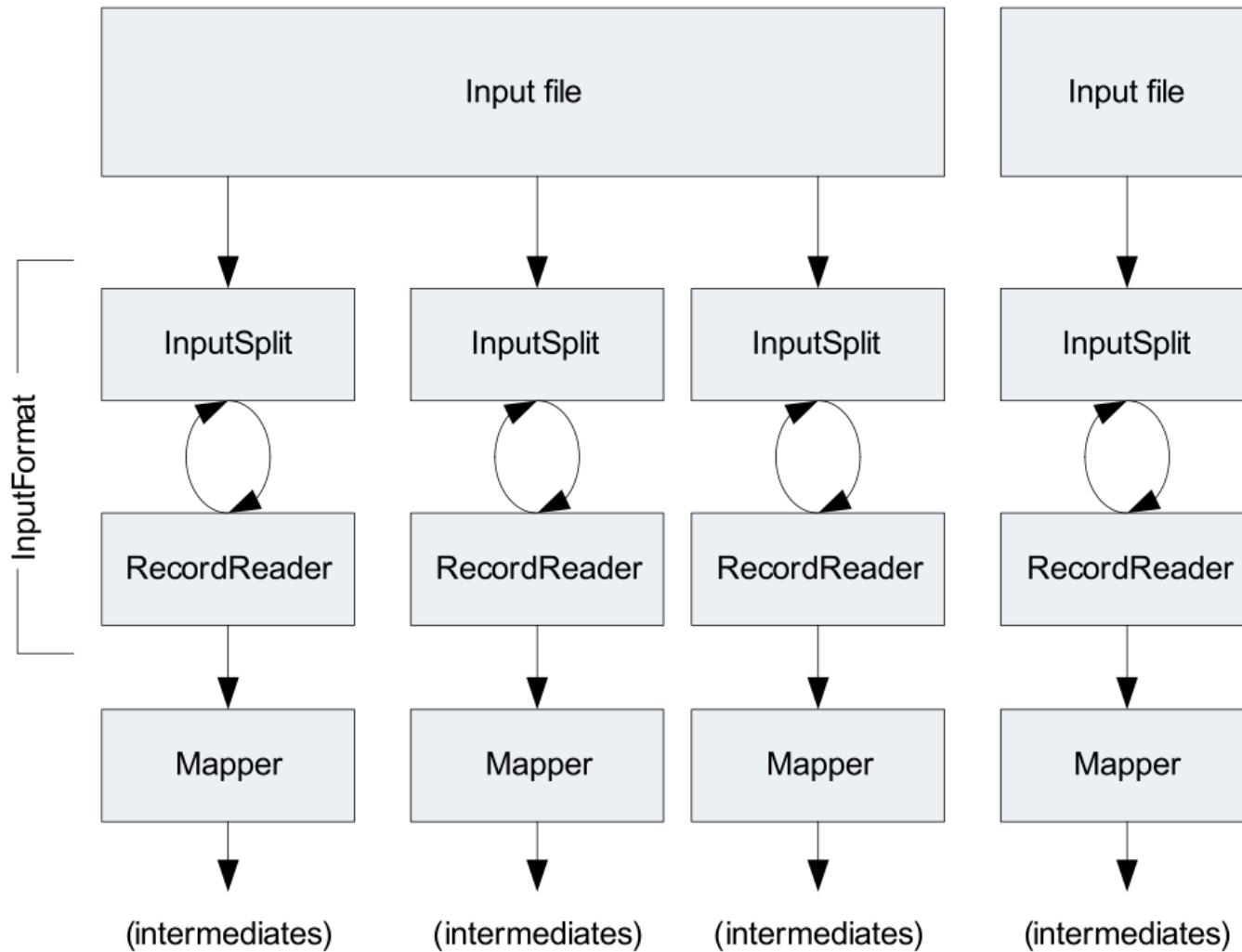
A Sample MapReduce Program: Introduction

- In the previous chapter, you ran a sample MapReduce program
 - WordCount, which counted the number of occurrences of each unique word in a set of files
- In this chapter, we will examine the code for WordCount
 - This will demonstrate the Hadoop API
- We will also investigate Hadoop Streaming
 - Allows you to write MapReduce programs in (virtually) any language

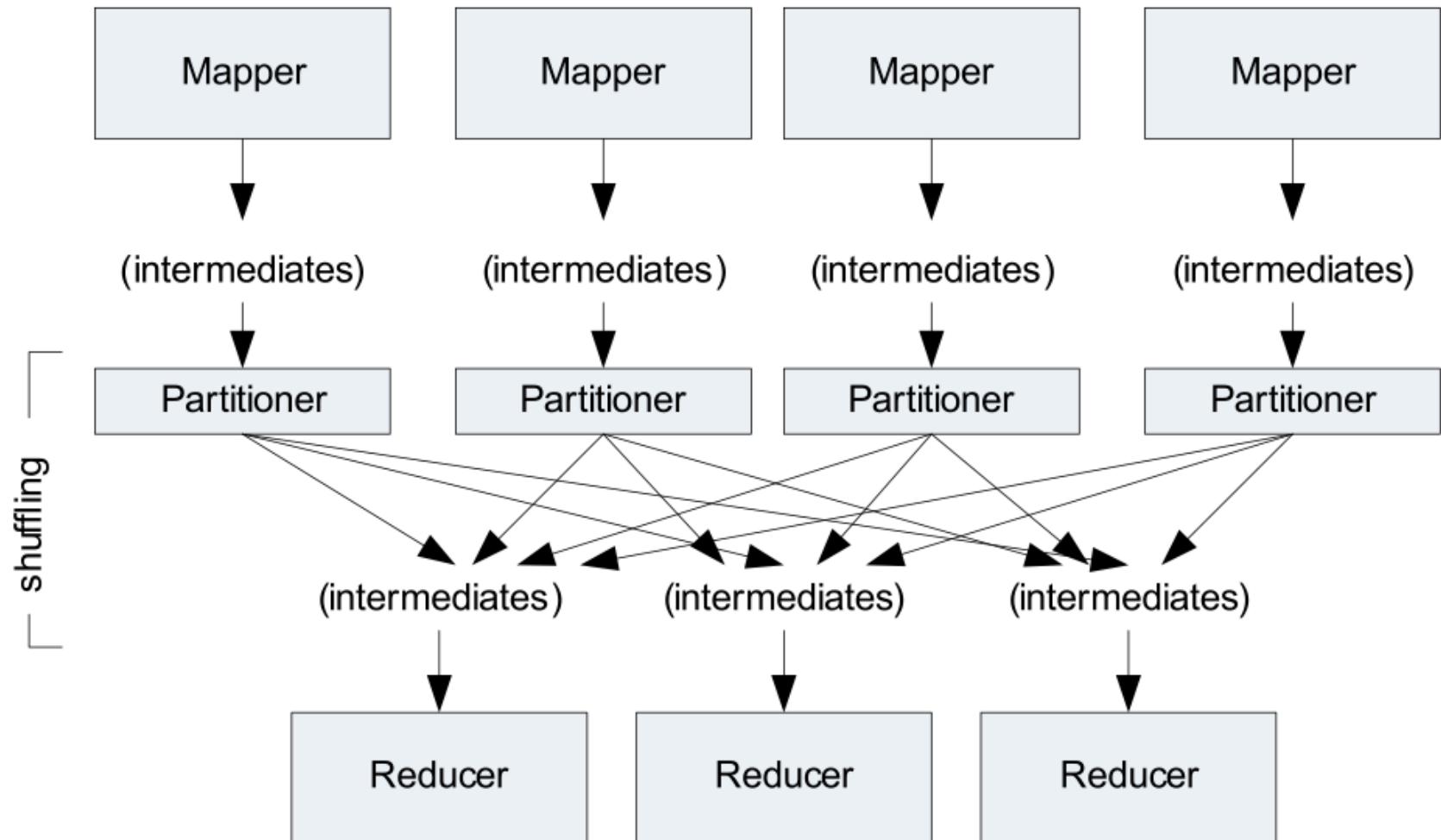
The MapReduce Flow: Introduction

- On the following slides we show the MapReduce flow
- Each of the portions (RecordReader, Mapper, Partitioner, Reducer, etc.) can be created by the developer
- We will cover each of these as we move through the course
- You will always create at least a Mapper, Reducer, and driver code
 - Those are the portions we will investigate in this chapter

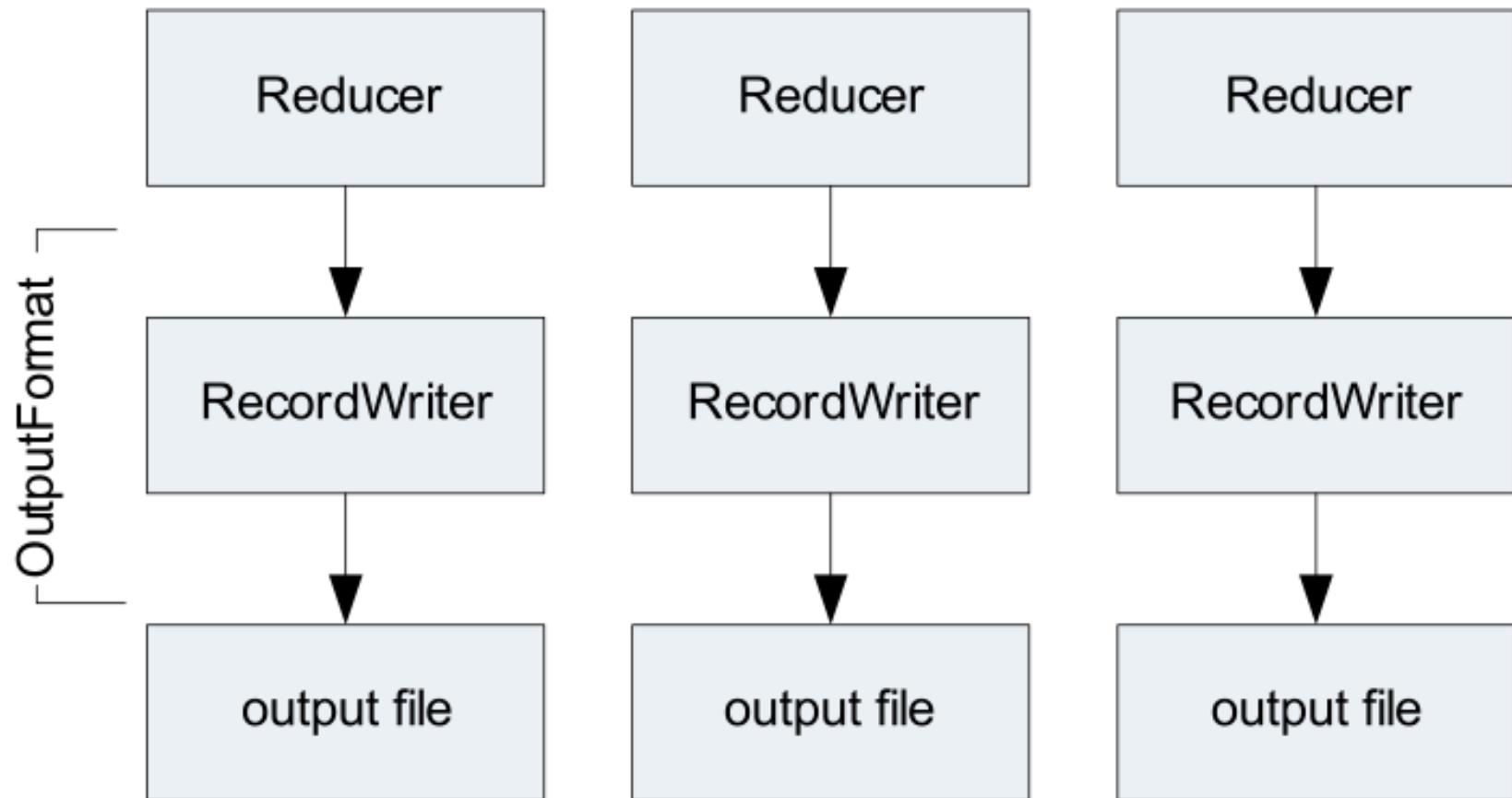
The MapReduce Flow: The Mapper



The MapReduce Flow: Shuffle and Sort



The MapReduce Flow: Reducers to Outputs



Writing a MapReduce Program

The MapReduce Flow



Examining our Sample MapReduce program

The Driver Code

The Mapper

The Reducer

Hadoop's Streaming API

Using Eclipse for Rapid Development

Hands-On Exercise: Write a MapReduce program

The New MapReduce API

Conclusion

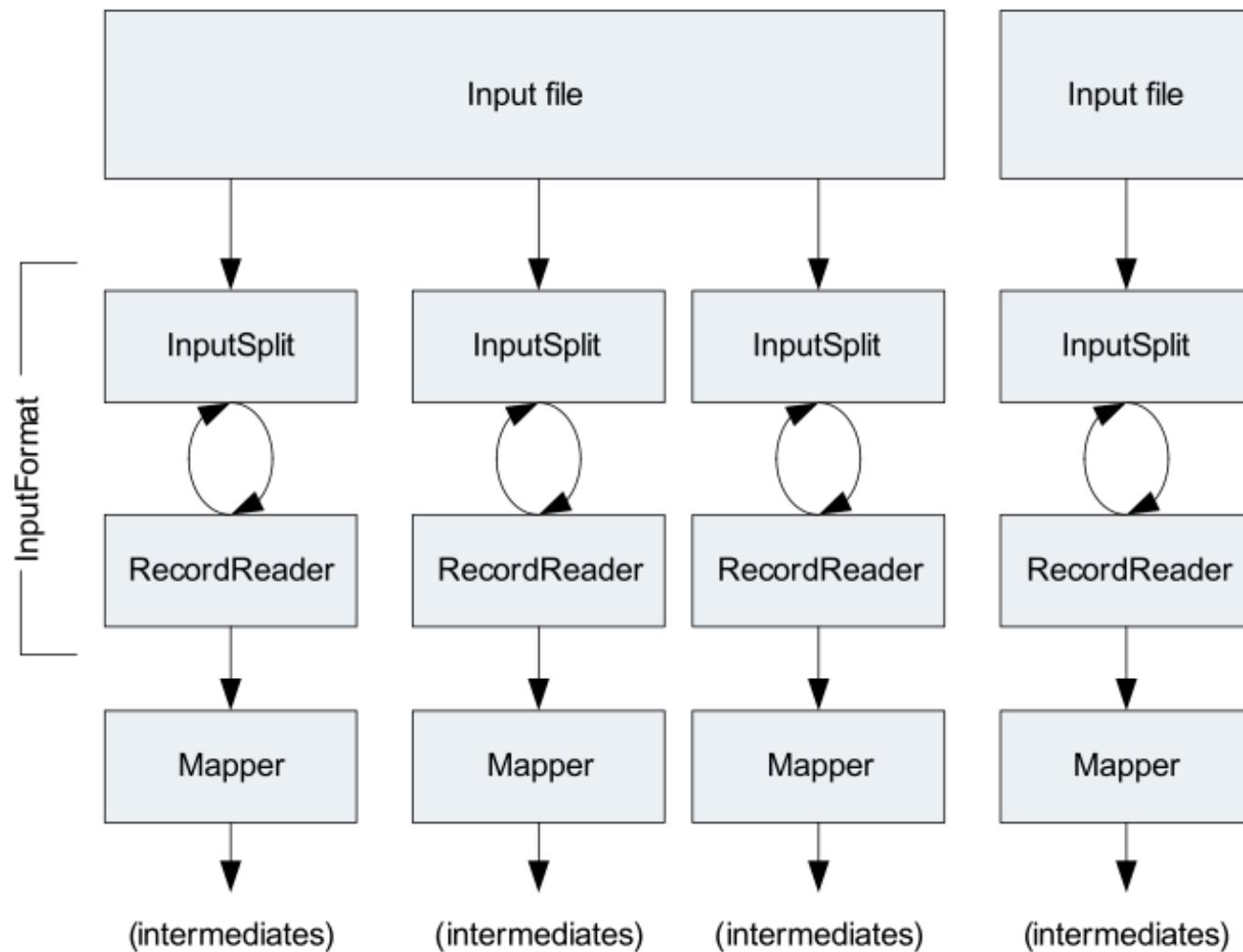
Our MapReduce Program: WordCount

- To investigate the API, we will dissect the WordCount program you ran in the previous chapter
- This consists of three portions
 - The driver code
 - Code that runs on the client to configure and submit the job
 - The Mapper
 - The Reducer
- Before we look at the code, we need to cover some basic Hadoop API concepts

Getting Data to the Mapper

- The data passed to the Mapper is specified by an *InputFormat*
 - Specified in the driver code
 - Defines the location of the input data
 - A file or directory, for example
 - Determines how to split the input data into *input splits*
 - Each Mapper deals with a single input split
 - InputFormat is a factory for RecordReader objects to extract (key, value) records from the input source

Getting Data to the Mapper (cont'd)



Some Standard InputFormats

- **FileInputFormat**
 - The base class used for all file-based InputFormats
- **TextInputFormat**
 - The default
 - Treats each \n-terminated line of a file as a value
 - Key is the byte offset within the file of that line
- **KeyValueTextInputFormat**
 - Maps \n-terminated lines as ‘key SEP value’
 - By default, separator is a tab
- **SequenceFileInputFormat**
 - Binary file of (key, value) pairs with some additional metadata
- **SequenceFileAsTextInputFormat**
 - Similar, but maps (key.toString(), value.toString())

Keys and Values are Objects

- Keys and values in Hadoop are Objects
- Values are objects which implement `Writable`
- Keys are objects which implement `WritableComparable`

What is Writable?

- Hadoop defines its own ‘box classes’ for strings, integers and so on
 - IntWritable for ints
 - LongWritable for longs
 - FloatWritable for floats
 - DoubleWritable for doubles
 - Text for strings
 - Etc.
- The Writable interface makes serialization quick and easy for Hadoop
- Any value’s type must implement the Writable interface

What is WritableComparable?

- A **WritableComparable** is a **Writable** which is also **Comparable**
 - Two **WritableComparables** can be compared against each other to determine their ‘order’
 - Keys must be **WritableComparables** because they are passed to the Reducer in sorted order
 - We will talk more about **WritableComparable** later
- Note that despite their names, all Hadoop box classes implement both **Writable** and **WritableComparable**
 - For example, **IntWritable** is actually a **WritableComparable**

Writing a MapReduce Program

The MapReduce Flow

Examining our Sample MapReduce program



The Driver Code

The Mapper

The Reducer

Hadoop's Streaming API

Using Eclipse for Rapid Development

Hands-On Exercise: Write a MapReduce program

The New MapReduce API

Conclusion

The Driver Code: Introduction

- The driver code runs on the client machine
- It configures the job, then submits it to the cluster

The Driver: Complete Code

```
import org.apache.hadoop.fs.Path;
import org.apache.hadoop.io.IntWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapred.FileInputFormat;
import org.apache.hadoop.mapred.FileOutputFormat;
import org.apache.hadoop.mapred.JobClient;
import org.apache.hadoop.mapred.JobConf;
import org.apache.hadoop.conf.Configured;
import org.apache.hadoop.util.Tool;
import org.apache.hadoop.util.ToolRunner;

public class WordCountDriver extends Configured implements Tool {
    public int run(String[] args) throws Exception {

        if (args.length != 2) {
            System.out.printf(
                "Usage: %s [generic options] <input dir> <output dir>\n",
                getClass().getSimpleName());
            ToolRunner.printGenericCommandUsage(System.out);
            return -1;
        }
        JobConf conf = new JobConf(getConf(), WordCountDriver.class);
        conf.setJobName(this.getClass().getName());

        FileInputFormat.setInputPaths(conf, new Path(args[0]));
        FileOutputFormat.setOutputPath(conf, new Path(args[1]));

        conf.setMapperClass(WordMapper.class);
        conf.setReducerClass(SumReducer.class);
    }
}
```

The Driver: Complete Code (cont'd)

```
conf.setMapOutputKeyClass(Text.class);
conf.setMapOutputValueClass(IntWritable.class);

conf.setOutputKeyClass(Text.class);
conf.setOutputValueClass(IntWritable.class);

JobClient.runJob(conf);
return 0;
}

public static void main(String[] args) throws Exception {
    int exitCode = ToolRunner.run(new WordCountDriver(), args);
    System.exit(exitCode);
}
}
```

The Driver: Import Statements

```
import org.apache.hadoop.fs.Path;
import org.apache.hadoop.io.IntWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapred.FileInputFormat;
import org.apache.hadoop.mapred.FileOutputFormat;
import org.apache.hadoop.mapred.JobClient;
import org.apache.hadoop.mapred.JobConf;
import org.apache.hadoop.conf.Configured;
import org.apache.hadoop.util.Tool;
import org.apache.hadoop.util.ToolRunner;

public class WordCount extends Configured implements Tool {
    public int run(String[] args) throws Exception {
        if (args.length != 2) {
            System.exit(2);
        }
        JobConf conf = getConf();
        conf.setJobName(this.getClass().getName());

        FileInputFormat.setInputPaths(conf, new Path(args[0]));
        FileOutputFormat.setOutputPath(conf, new Path(args[1]));

        conf.setMapperClass(WordMapper.class);
        conf.setReducerClass(SumReducer.class);
        return JobClient.runJob(conf).getCount();
    }
}
```

You will typically import these classes into every MapReduce job you write. We will omit the `import` statements in future slides for brevity.

The Driver: Main Code

```
public class WordCountDriver extends Configured implements Tool {
    public int run(String[] args) throws Exception {

        if (args.length != 2) {
            System.out.printf(
                "Usage: %s [generic options] <input dir> <output dir>\n",
                getClass().getSimpleName());
            ToolRunner.printGenericCommandUsage(System.out);
            return -1;
        }
        JobConf conf = new JobConf(getConf(), WordCountDriver.class);
        conf.setJobName(this.getClass().getName());

        FileInputFormat.setInputPaths(conf, new Path(args[0]));
        FileOutputFormat.setOutputPath(conf, new Path(args[1]));

        conf.setMapperClass(WordMapper.class);
        conf.setReducerClass(SumReducer.class);
        conf.setMapOutputKeyClass(Text.class);
        conf.setMapOutputValueClass(IntWritable.class);

        conf.setOutputKeyClass(Text.class);
        conf.setOutputValueClass(IntWritable.class);

        JobClient.runJob(conf);
        return 0;
    }

    public static void main(String[] args) throws Exception {
        int exitCode = ToolRunner.run(new WordCountDriver(), args);
        System.exit(exitCode);
    }
}
```

The Driver Class: Using ToolRunner

```
public class WordCountDriver extends Configured implements Tool {  
    public int run(String[] args) throws Exception {  
        Configuration conf = new Configuration();
```

Your driver class extends Configured and implements Tool. This allows the user to specify configuration settings on the command line, which will then be incorporated into the job's configuration when it is submitted to the server. Although this is not compulsory, it is considered a best practice. (We will discuss ToolRunner in more detail later.)

```
        conf.setMapOutputValueClass(IntWritable.class);  
  
        conf.setOutputKeyClass(Text.class);  
        conf.setOutputValueClass(IntWritable.class);  
  
        JobClient.runJob(conf);  
        return 0;  
    }  
  
    public static void main(String[] args) throws Exception {  
        int exitCode = ToolRunner.run(new WordCountDriver(), args);  
        System.exit(exitCode);  
    }  
}
```

me());

The Driver Class: Using ToolRunner (cont'd)

```
public class WordCountDriver extends Configured implements Tool {  
    public int run(String[] args) throws Exception {  
  
        if (args.length != 2) {  
            System.out.printf(  
                "Usage: %s [generic options] <input dir> <output dir>\n", getClass().getSimpleName());  
            ToolRunner.printGenericCommandUsage(System.out);  
            return -1;  
        }  
        JobConf conf = new JobConf(getConf(), WordCountDriver.class);  
        conf.setJobName(this.getClass().getName());  
  
        FileInputFormat.setInputPaths(conf, new Path(args[0]));  
        FileOutputFormat.setOutputPath(conf, new Path(args[1]));  
  
        conf.setMapperClass(WordMapper.class);  
    }  
}
```

The main method simply calls ToolRunner.run() , passing in the driver class and the command-line arguments. The job will then be configured and submitted in the run method.

```
public static void main(String[] args) throws Exception {  
    int exitCode = ToolRunner.run(new WordCountDriver(), args);  
    System.exit(exitCode);  
}
```

Sanity Checking The Job's Invocation

```
public class WordCountDriver extends Configured implements Tool {  
    public int run(String[] args) throws Exception {  
  
        if (args.length != 2) {  
            System.out.printf(  
                "Usage: %s [generic options] <input dir> <output dir>\n", getClass().getSimpleName());  
            ToolRunner.printGenericCommandUsage(System.out);  
            return -1;  
        }  
        JobConf conf = new JobConf(ToolRunner.getConf(), WordCountDriver.class);  
        ...  
    }  
}
```

The first step is to ensure that we have been given two command-line arguments. If not, print a help message and exit.

```
conf.setOutputKeyClass(Text.class);  
conf.setOutputValueClass(IntWritable.class);  
  
JobClient.runJob(conf);  
return 0;  
}  
  
public static void main(String[] args) throws Exception {  
    int exitCode = ToolRunner.run(new WordCountDriver(), args);  
    System.exit(exitCode);  
}
```

Configuring The Job With JobConf

```
public class WordCountDriver extends Configured implements Tool {  
    public int run(String[] args) throws Exception {  
  
        if (args.length != 2) {  
            System.out.printf(  
                "Usage: %s [generic options] <input dir> <output dir>\n", getClass().getSimpleName());  
            ToolRunner.printGenericCommandUsage(System.out);  
            return -1;  
        }  
        JobConf conf = new JobConf(getConf(), WordCountDriver.class);  
        conf.setJobName(this.getClass().getName());
```

To configure the job, create a new `JobConf` object and specify the class which will be called to run the job.

```
        conf.setMapOutputValueClass(IntWritable.class);  
  
        conf.setOutputKeyClass(Text.class);  
        conf.setOutputValueClass(IntWritable.class);  
  
        JobClient.runJob(conf);  
        return 0;  
    }  
  
    public static void main(String[] args) throws Exception {  
        int exitCode = ToolRunner.run(new WordCountDriver(), args);  
        System.exit(exitCode);  
    }  
}
```

Creating a New JobConf Object

- **The JobConf class allows you to set configuration options for your MapReduce job**
 - The classes to be used for your Mapper and Reducer
 - The input and output directories
 - Many other options
- **Any options not explicitly set in your driver code will be read from your Hadoop configuration files**
 - Usually located in /etc/hadoop/conf
- **Any options not specified in your configuration files will receive Hadoop's default values**

Naming The Job

```
public class WordCountDriver extends Configured implements Tool {  
    public int run(String[] args) throws Exception {  
  
        if (args.length != 2) {  
            System.out.printf(  
                "Usage: %s [generic options] <input dir> <output dir>\n", getClass().getSimpleName());  
            ToolRunner.printGenericCommandUsage(System.out);  
            return -1;  
        }  
        JobConf conf = new JobConf(getConf(), WordCountDriver.class);  
        conf.setJobName(this.getClass().getName());
```

Give the job a meaningful name.

```
        conf.setReducerClass(SumReducer.class);  
        conf.setMapOutputKeyClass(Text.class);  
        conf.setMapOutputValueClass(IntWritable.class);  
  
        conf.setOutputKeyClass(Text.class);  
        conf.setOutputValueClass(IntWritable.class);  
  
        JobClient.runJob(conf);  
        return 0;  
    }  
  
    public static void main(String[] args) throws Exception {  
        int exitCode = ToolRunner.run(new WordCountDriver(), args);  
        System.exit(exitCode);  
    }  
}
```

Specifying Input and Output Directories

```
public class WordCountDriver extends Configured implements Tool {  
    public int run(String[] args) throws Exception {  
  
        if (args.length != 2) {  
            System.out.printf(  
                "Usage: %s [generic options] <input dir> <output dir>\n", getClass().getSimpleName());  
            ToolRunner.printGenericCommandUsage(System.out);  
            return -1;  
        }  
        JobConf conf = new JobConf(getConf(), WordCountDriver.class);  
        conf.setJobName(this.getClass().getName());  
  
        FileInputFormat.setInputPaths(conf, new Path(args[0]));  
        FileOutputFormat.setOutputPath(conf, new Path(args[1]));  
    }  
}
```

Next, specify the input directory from which data will be read, and the output directory to which final output will be written.

```
    JobClient.runJob(conf);  
    return 0;  
}  
  
public static void main(String[] args) throws Exception {  
    int exitCode = ToolRunner.run(new WordCountDriver(), args);  
    System.exit(exitCode);  
}
```

Specifying the InputFormat

- The default InputFormat (`TextInputFormat`) will be used unless you specify otherwise
- To use an InputFormat other than the default, use e.g.

```
conf.setInputFormat(KeyValueTextInputFormat.class)
```

Determining Which Files To Read

- By default, `FileInputFormat.setInputPaths()` will read all files from a specified directory and send them to Mappers
 - Exceptions: items whose names begin with a period (.) or underscore (_)
 - Globs can be specified to restrict input
 - For example, /2010/*/01/*
- Alternatively, `FileInputFormat.addInputPath()` can be called multiple times, specifying a single file or directory each time
- More advanced filtering can be performed by implementing a `PathFilter`
 - Interface with a method named `accept`
 - Takes a path to a file, returns `true` or `false` depending on whether or not the file should be processed

Specifying Final Output With OutputFormat

- `FileOutputFormat.setOutputPath()` specifies the directory to which the Reducers will write their final output
- The driver can also specify the format of the output data
 - Default is a plain text file
 - Could be explicitly written as

```
conf.setOutputFormat(TextOutputFormat.class);
```
- We will discuss OutputFormats in more depth in a later chapter

Specify The Classes for Mapper and Reducer

```
public class WordCountDriver extends Configured implements Tool {  
    public int run(String[] args) throws Exception {  
  
        if (args.length != 2) {  
            System.out.printf(  
                "Usage: %s [generic options] <input dir> <output dir>\n", getClass().getSimpleName());  
            ToolRunner.printGenericCommandUsage(System.out);  
            return -1;  
        }  
        JobConf conf = new JobConf(getConf(), WordCountDriver.class);  
        conf.setJobName(this.getClass().getName());  
  
        FileInputFormat.setInputPaths(conf, new Path(args[0]));  
        FileOutputFormat.setOutputPath(conf, new Path(args[1]));  
  
        conf.setMapperClass(WordMapper.class);  
        conf.setReducerClass(SumReducer.class);  
        conf.setMapOutputKeyClass(Text.class);  
        conf.setMapOutputValueClass(IntWritable.class);  
    }  
  
    public static void main(String[] args) throws Exception {  
        int exitCode = ToolRunner.run(new WordCountDriver(), args);  
        System.exit(exitCode);  
    }  
}
```

Give the JobConf object information about which classes are to be instantiated as the Mapper and Reducer.

Specify The Intermediate Data Types

```
public class WordCountDriver extends Configured implements Tool {  
    public int run(String[] args) throws Exception {  
  
        if (args.length != 2) {  
            System.out.printf(  
                "Usage: %s [generic options] <input dir> <output dir>\n", getClass().getSimpleName());  
            ToolRunner.printGenericCommandUsage(System.out);  
            return -1;  
        }  
        JobConf conf = new JobConf(getConf(), WordCountDriver.class);  
        conf.setJobName(this.getClass().getName());  
  
        FileInputFormat.setInputPaths(conf, new Path(args[0]));  
        FileOutputFormat.setOutputPath(conf, new Path(args[1]));  
  
        conf.setMapperClass(WordMapper.class);  
        conf.setReducerClass(SumReducer.class);  
        conf.setMapOutputKeyClass(Text.class);  
        conf.setMapOutputValueClass(IntWritable.class);  
    }  
}
```

Specify the types of the intermediate output key and value produced by the Mapper.

```
public static void main(String[] args) throws Exception {  
    int exitCode = ToolRunner.run(new WordCountDriver(), args);  
    System.exit(exitCode);  
}
```

Specify The Final Output Data Types

```
public class WordCountDriver extends Configured implements Tool {  
    public int run(String[] args) throws Exception {  
  
        if (args.length != 2) {  
            System.out.printf(  
                "Usage: %s [generic options] <input dir> <output dir>\n", getClass().getSimpleName());  
            ToolRunner.printGenericCommandUsage(System.out);  
            return -1;  
        }  
        JobConf conf = new JobConf(getConf(), WordCountDriver.class);  
        conf.setJobName(this.getClass().getName());  
  
        FileInputFormat.setInputPaths(conf, new Path(args[0]));  
        FileOutputFormat.setOutputPath(conf, new Path(args[1]));  
  
        conf.setMapperClass(WordMapper.class);  
        conf.setReducerClass(SumReducer.class);  
        conf.setMapOutputKeyClass(Text.class);  
        conf.setMapOutputValueClass(IntWritable.class);  
  
        conf.setOutputKeyClass(Text.class);  
        conf.setOutputValueClass(IntWritable.class);  
    }  
}
```

Specify the types of the Reducer's output key and value.

```
public static void main(String[] args) throws Exception {  
    int exitCode = ToolRunner.run(new WordCountDriver(), args);  
    System.exit(exitCode);  
}
```

Running The Job

```
public class WordCountDriver extends Configured implements Tool {  
    public int run(String[] args) throws Exception {  
  
        if (args.length != 2) {  
            System.out.printf(  
                "Usage: %s [generic options] <input dir> <output dir>\n", getClass().getSimpleName());  
            ToolRunner.printGenericCommandUsage(System.out);  
            return -1;  
        }  
        JobConf conf = new JobConf(getConf(), WordCountDriver.class);  
        conf.setJobName(this.getClass().getName());  
  
        FileInputFormat.setInputPaths(conf, new Path(args[0]));  
        FileOutputFormat.setOutputPath(conf, new Path(args[1]));  
  
        conf.setMapperClass(WordMapper.class);  
        conf.setReducerClass(SumReducer.class);  
  
        JobClient.runJob(conf);  
        return 0;  
    }  
  
    public static void main(String[] args) throws Exception {  
        int exitCode = ToolRunner.run(new WordCountDriver(), args);  
        System.exit(exitCode);  
    }  
}
```

Finally, run the job by calling the `runJob` method.

Running The Job (cont'd)

- **There are two ways to run your MapReduce job:**
 - `JobClient.runJob(conf)`
 - Blocks (waits for the job to complete before continuing)
 - `JobClient.submitJob(conf)`
 - Does not block (driver code continues as the job is running)
- **JobClient determines the proper division of input data into InputSplits**
- **JobClient then sends the job information to the JobTracker daemon on the cluster**

Reprise: Driver Code

```
public class WordCountDriver extends Configured implements Tool {
    public int run(String[] args) throws Exception {

        if (args.length != 2) {
            System.out.printf(
                "Usage: %s [generic options] <input dir> <output dir>\n",
                getClass().getSimpleName());
            ToolRunner.printGenericCommandUsage(System.out);
            return -1;
        }
        JobConf conf = new JobConf(getConf(), WordCountDriver.class);
        conf.setJobName(this.getClass().getName());

        FileInputFormat.setInputPaths(conf, new Path(args[0]));
        FileOutputFormat.setOutputPath(conf, new Path(args[1]));

        conf.setMapperClass(WordMapper.class);
        conf.setReducerClass(SumReducer.class);
        conf.setMapOutputKeyClass(Text.class);
        conf.setMapOutputValueClass(IntWritable.class);

        conf.setOutputKeyClass(Text.class);
        conf.setOutputValueClass(IntWritable.class);

        JobClient.runJob(conf);
        return 0;
    }

    public static void main(String[] args) throws Exception {
        int exitCode = ToolRunner.run(new WordCountDriver(), args);
        System.exit(exitCode);
    }
}
```

Writing a MapReduce Program

The MapReduce Flow

Examining our Sample MapReduce program

The Driver Code



The Mapper

The Reducer

Hadoop's Streaming API

Using Eclipse for Rapid Development

Hands-On Exercise: Write a MapReduce program

The New MapReduce API

Conclusion

The Mapper: Complete Code

```
import java.io.IOException;

import org.apache.hadoop.io.IntWritable;
import org.apache.hadoop.io.LongWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapred.MapReduceBase;
import org.apache.hadoop.mapred.Mapper;
import org.apache.hadoop.mapred.OutputCollector;
import org.apache.hadoop.mapred.Reporter;

public class WordMapper extends MapReduceBase implements
    Mapper<LongWritable, Text, Text, IntWritable> {

    public void map(LongWritable key, Text value,
        OutputCollector<Text, IntWritable> output, Reporter reporter)
        throws IOException {
        String s = value.toString();
        for (String word : s.split("\\w+")) {
            if (word.length() > 0) {
                output.collect(new Text(word), new IntWritable(1));
            }
        }
    }
}
```

The Mapper: import Statements

```
import java.io.IOException;

import org.apache.hadoop.io.IntWritable;
import org.apache.hadoop.io.LongWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapred.MapReduceBase;
import org.apache.hadoop.mapred.Mapper;
import org.apache.hadoop.mapred.OutputCollector;
import org.apache.hadoop.mapred.Reporter;

public class MyMapper extends Mapper<LongWritable, Text, IntWritable, IntWritable> {
    public void map(LongWritable key, Text value, OutputCollector<IntWritable, IntWritable> output, Reporter reporter) throws IOException {
        String word = value.toString();
        for (String s : word.split(" ")) {
            if (s.length() > 0) {
                output.collect(new Text(s), new IntWritable(1));
            }
        }
    }
}
```

You will typically import `java.io.IOException`, and the `org.apache.hadoop` classes shown, in every Mapper you write. We will omit the import statements in future slides for brevity.

The Mapper: Main Code

```
public class WordMapper extends MapReduceBase
    implements Mapper<LongWritable, Text, Text, IntWritable> {

    public void map(LongWritable key, Text value,
                    OutputCollector<Text, IntWritable> output, Reporter reporter)
        throws IOException {
        String s = value.toString();
        for (String word : s.split("\\w+")) {
            if (word.length() > 0) {
                output.collect(new Text(word), new IntWritable(1));
            }
        }
    }
}
```

The Mapper: Main Code (cont'd)

```
public class WordMapper extends MapReduceBase  
    implements Mapper<LongWritable, Text, Text, IntWritable> {  
  
    public void map(LongWritable key, Text value,  
                    Text output, Reporter reporter)  
        throws IOException {  
            String str = value.toString();  
            for (String word : str.split(" ")) {  
                output.set(word);  
                IntWritable one = new IntWritable(1);  
                context.write(output, one);  
            }  
        }  
}
```

Your Mapper class should extend MapReduceBase, and implement the Mapper interface. The Mapper interface expects four generics, which define the types of the input and output key/value pairs. The first two parameters define the input key and value types, the second two define the output key and value types.

The map Method

```
public class WordMapper extends MapReduceBase  
    implements Mapper<LongWritable, Text, Text, IntWritable> {  
  
    public void map(LongWritable key, Text value,  
        OutputCollector<Text, IntWritable> output, Reporter reporter)  
        throws IOException {  
        String s = value.toString();  
        for (int i = 0; i < s.length(); i++) {  
            output.collect(new Text(s.substring(i, i + 1)),  
                new IntWritable(1));  
        }  
    }  
}
```

The `map` method's signature looks like this. It will be passed a key, a value, an `OutputCollector` object and a `Reporter` object. The `OutputCollector` is used to write the intermediate data; you must specify the data types that it will write.

The map Method: Processing The Line

```
public class WordMapper extends MapReduceBase
    implements Mapper<LongWritable, Text, Text, IntWritable> {

    public void map(LongWritable key, Text value,
                    OutputCollector<Text, IntWritable> output, Reporter reporter)
        throws IOException {
        String s = value.toString();
        for (String word : s.split("\\\\W+")) {
            output.collect(key, new Text(word));
        }
    }
}
```

value is a Text object, so we retrieve the string it contains.

The map Method: Processing The Line (cont'd)

```
public class WordMapper extends MapReduceBase  
    implements Mapper<LongWritable, Text, Text, IntWritable> {  
  
    public void map(LongWritable key, Text value,  
        OutputCollector<Text, IntWritable> output, Reporter reporter)  
        throws IOException {  
        String s = value.toString();  
        for (String word : s.split("\\w+")) {  
            if (word.length() > 0) {  
                output.collect(new Text(word), new IntWritable(1));  
            }  
        }  
    }  
}
```

We then split the string up into words using any non-alphanumeric characters as the word delimiter, and loop through those words.

Outputting Intermediate Data

```
public class WordMapper extends MapReduceBase  
    implements Mapper<LongWritable, Text, Text, IntWritable> {  
  
    public void map(LongWritable key, Text value,  
        OutputCollector<Text, IntWritable> output, Reporter reporter)  
        throws IOException {  
        String s = value.toString();  
        for (String word : s.split("\\w+")) {  
            if (word.length() > 0) {  
                output.collect(new Text(word), new IntWritable(1));  
            }  
        }  
    }  
}
```

To emit a (key, value) pair, we call the `collect` method of our `OutputCollector` object. The key will be the word itself, the value will be the number 1. Recall that the output key must be of type `WritableComparable`, and the value must be a `Writable`.

Reprise: The Map Method

```
public class WordMapper extends MapReduceBase
    implements Mapper<LongWritable, Text, Text, IntWritable> {

    public void map(LongWritable key, Text value,
                    OutputCollector<Text, IntWritable> output, Reporter reporter)
        throws IOException {
        String s = value.toString();
        for (String word : s.split("\\w+")) {
            if (word.length() > 0) {
                output.collect(new Text(word), new IntWritable(1));
            }
        }
    }
}
```

The Reporter Object

- Notice that in this example we have not used the **Reporter** object which was passed to the Mapper
- The **Reporter** object can be used to pass some information back to the driver code
- We will investigate the **Reporter** later in the course

Writing a MapReduce Program

The MapReduce Flow

Examining our Sample MapReduce program

The Driver Code

The Mapper



The Reducer

Hadoop's Streaming API

Using Eclipse for Rapid Development

Hands-On Exercise: Write a MapReduce program

The New MapReduce API

Conclusion

The Reducer: Complete Code

```
import java.io.IOException;
import java.util.Iterator;

import org.apache.hadoop.io.IntWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapred.OutputCollector;
import org.apache.hadoop.mapred.MapReduceBase;
import org.apache.hadoop.mapred.Reducer;
import org.apache.hadoop.mapred.Reporter;

public class SumReducer extends MapReduceBase implements
    Reducer<Text, IntWritable, Text, IntWritable> {

    public void reduce(Text key, Iterator<IntWritable> values,
                      OutputCollector<Text, IntWritable> output, Reporter reporter)
        throws IOException {

        int wordCount = 0;
        while (values.hasNext()) {
            IntWritable value = values.next();
            wordCount += value.get();
        }
        output.collect(key, new IntWritable(wordCount));
    }
}
```

The Reducer: Import Statements

```
import java.io.IOException;
import java.util.Iterator;

import org.apache.hadoop.io.IntWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapred.OutputCollector;
import org.apache.hadoop.mapred.MapReduceBase;
import org.apache.hadoop.mapred.Reducer;
import org.apache.hadoop.mapred.Reporter;

public class Reducer extends MapReduceBase {
    public void reduce(Text key, Iterator<IntWritable> values,
                      OutputCollector<Text, IntWritable> output)
        throws IOException {
        int wordCount = 0;
        while (values.hasNext()) {
            IntWritable value = values.next();
            wordCount += value.get();
        }
        output.collect(key, new IntWritable(wordCount));
    }
}
```

As with the Mapper, you will typically import `java.io.IOException`, and the `org.apache.hadoop` classes shown, in every Reducer you write. You will also import `java.util.Iterator`, which will be used to step through the values provided to the Reducer for each key. We will omit the import statements in future slides for brevity.

The Reducer: Main Code

```
public class SumReducer extends MapReduceBase implements  
    Reducer<Text, IntWritable, Text, IntWritable> {  
  
    public void reduce(Text key, Iterator<IntWritable> values,  
                      OutputCollector<Text, IntWritable> output, Reporter reporter)  
        throws IOException {  
  
        int wordCount = 0;  
        while (values.hasNext()) {  
            IntWritable value = values.next();  
            wordCount += value.get();  
        }  
        output.collect(key, new IntWritable(wordCount));  
    }  
}
```

The Reducer: Main Code (cont'd)

```
public class SumReducer extends MapReduceBase implements  
    Reducer<Text, IntWritable, Text, IntWritable> {
```

Your Reducer class should extend MapReduceBase and implement Reducer. The Reducer interface expects four generics, which define the types of the input and output key/value pairs. The first two parameters define the intermediate key and value types, the second two define the final output key and value types. The keys are WritableComparables, the values are Writables.

The reduce Method

```
public class SumReducer extends MapReduceBase implements  
    Reducer<Text, IntWritable, Text, IntWritable> {  
  
    public void reduce(Text key, Iterator<IntWritable> values,  
                      OutputCollector<Text, IntWritable> output, Reporter reporter)  
        throws IOException {  
  
        int wordCount = 0;  
        while (values.hasNext()) {  
            IntWritable value = values.next();  
            wordCount += value.get();  
        }  
        output.collect(key, new IntWritable(wordCount));  
    }  
}
```

The reduce method receives a key and an Iterator of values; it also receives an OutputCollector object and a Reporter object.

Processing The Values

```
public class SumReducer extends MapReduceBase implements  
    Reducer<Text, IntWritable, Text, IntWritable> {  
  
    public void reduce(Text key, Iterator<IntWritable> values,  
                      OutputCollector<Text, IntWritable> output, Reporter reporter)  
        throws IOException {  
  
        int wordCount = 0;  
        while (values.hasNext()) {  
            IntWritable value = values.next();  
            wordCount += value.get();  
        }  
        output  
    }  
}
```

We use the `hasNext()` and `next()` methods on `values` to step through all the elements in the iterator. In our example, we are merely adding all the values together. We use `value().get()` to retrieve the actual numeric value.

Writing The Final Output

```
public class SumReducer extends MapReduceBase implements  
    Reducer<Text, IntWritable, Text, IntWritable> {  
  
    public void reduce(Text key, Iterator<IntWritable> values,  
                      OutputCollector<Text, IntWritable> output, Reporter reporter)  
        throws IOException {  
  
        int wordCount = 0;  
        while (values.hasNext()) {  
            IntWritable value = values.next();  
            wordCount += value.get();  
        }  
        output.collect(key, new IntWritable(wordCount));  
    }  
}
```

Finally, we write the output (key, value) pair using the collect method of our OutputCollector object.

Reprise: The Reduce Method

```
public class SumReducer extends MapReduceBase implements  
    Reducer<Text, IntWritable, Text, IntWritable> {  
  
    public void reduce(Text key, Iterator<IntWritable> values,  
                      OutputCollector<Text, IntWritable> output, Reporter reporter)  
        throws IOException {  
  
        int wordCount = 0;  
        while (values.hasNext()) {  
            IntWritable value = values.next();  
            wordCount += value.get();  
        }  
        output.collect(key, new IntWritable(wordCount));  
    }  
}
```

Writing a MapReduce Program

The MapReduce Flow

Examining our Sample MapReduce program

The Driver Code

The Mapper

The Reducer



Hadoop's Streaming API

Using Eclipse for Rapid Development

Hands-On Exercise: Write a MapReduce program

The New MapReduce API

Conclusion

The Streaming API: Motivation

- Many organizations have developers skilled in languages other than Java
 - Perl
 - Ruby
 - Python
 - Etc
- The Streaming API allows developers to use any language they wish to write Mappers and Reducers
 - As long as the language can read from standard input and write to standard output

The Streaming API: Advantages

- **Advantages of the Streaming API:**

- No need for non-Java coders to learn Java
- Fast development time
- Ability to use existing code libraries

How Streaming Works

- To implement streaming, write separate Mapper and Reducer programs in the language of your choice
 - They will receive input via stdin
 - They should write their output to stdout
- If `TextInputFormat` (the default) is used, the streaming Mapper just receives each line from the file on stdin
 - No key is passed
- Streaming Mapper and streaming Reducer's output should be sent to stdout as key (tab) value (newline)
- Separators other than tab can be specified

Streaming: Example Mapper

- Example streaming wordcount Mapper:

```
#!/usr/bin/env perl
while (<>) {
    chomp;
    (@words) = split /\s+/;
    foreach $w (@words) {
        print "$w\t1\n";
    }
}
```

Streaming Reducers: Caution

- Recall that in Java, all the values associated with a key are passed to the Reducer as an Iterator
- Using Hadoop Streaming, the Reducer receives its input as (key, value) pairs
 - One per line of standard input
- Your code will have to keep track of the key so that it can detect when values from a new key start appearing

Launching a Streaming Job

- To launch a Streaming job, use e.g.,:

```
hadoop jar $HADOOP_HOME/contrib/streaming/hadoop-streaming*.jar \
  -input myInputDirs \
  -output myOutputDir \
  -mapper myMapScript.pl \
  -reducer myReduceScript.pl \
  -file myMapScript.pl \
  -file myReduceScript.pl
```

- Many other command-line options are available
 - See the documentation for full details
- Note that system commands can be used as a Streaming Mapper or Reducer
 - awk, grep, sed, wc etc

Writing a MapReduce Program

The MapReduce Flow

Examining our Sample MapReduce program

The Driver Code

The Mapper

The Reducer

Hadoop's Streaming API

 Using Eclipse for Rapid Development

Hands-On Exercise: Write a MapReduce program

The New MapReduce API

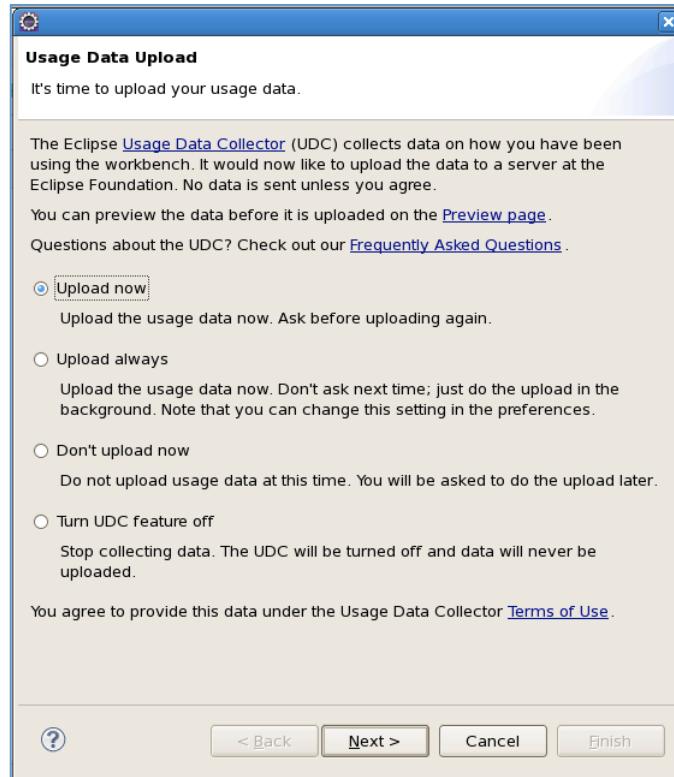
Conclusion

Integrated Development Environments

- There are many Integrated Development Environments (IDEs) available
- Eclipse is one such IDE
 - Open source
 - Very popular amongst Java developers
 - Has plug-ins to speed development in several different languages
- If you would prefer to write your code this week using a terminal-based editor such as vi, we certainly won't stop you!
 - But using Eclipse can dramatically speed up your development process
- On the next few slides we will demonstrate how to use Eclipse to write a MapReduce program

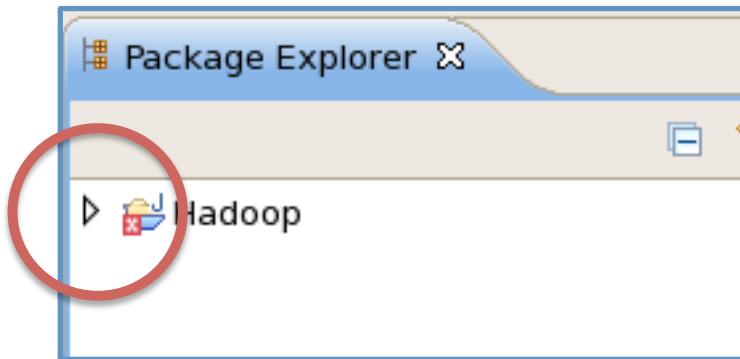
Using Eclipse

- Launch Eclipse by double-clicking on the Eclipse icon on the desktop
 - If you are asked whether you want to send usage data, hit **Cancel**

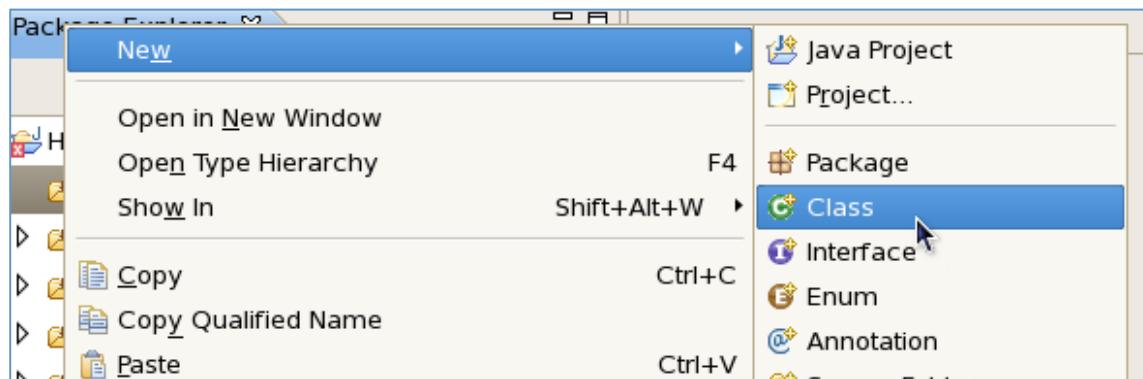


Using Eclipse (cont'd)

- Expand the ‘Hadoop’ project

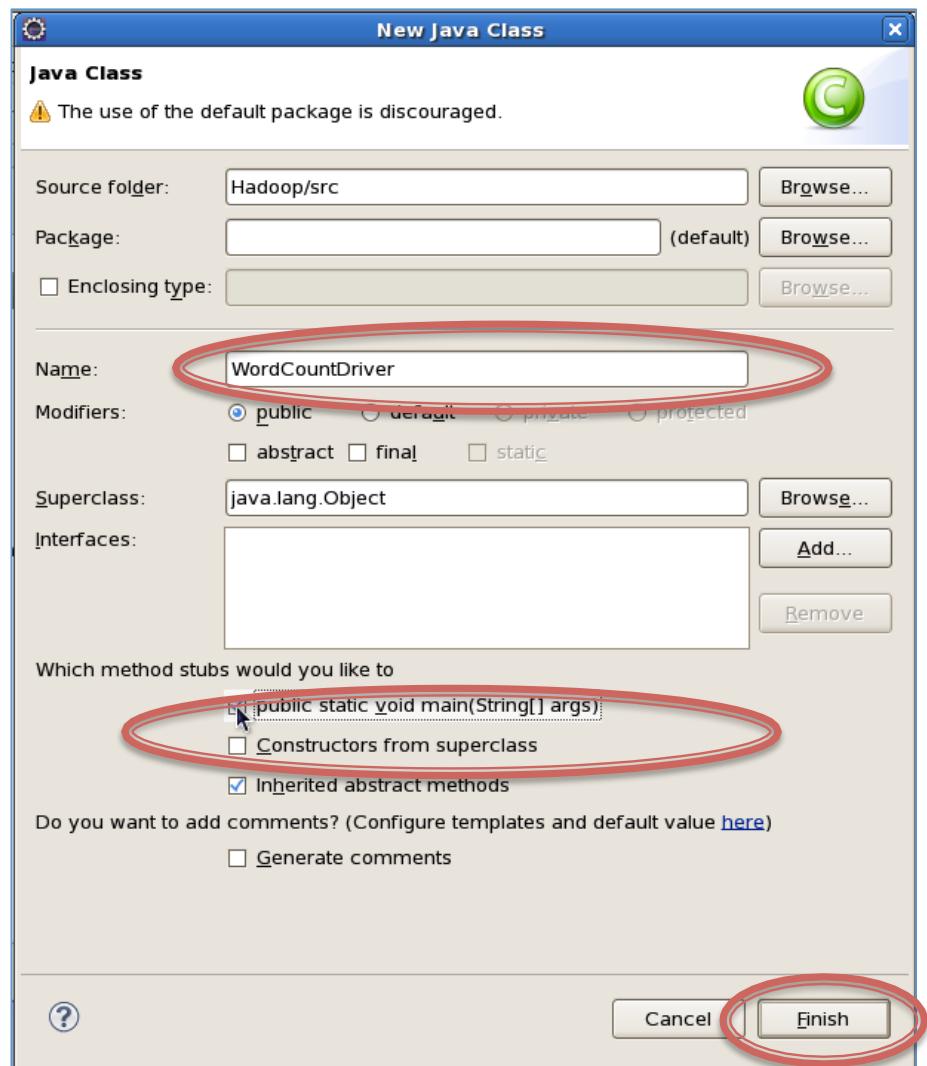


- Right-click on ‘src’, and choose New->Class



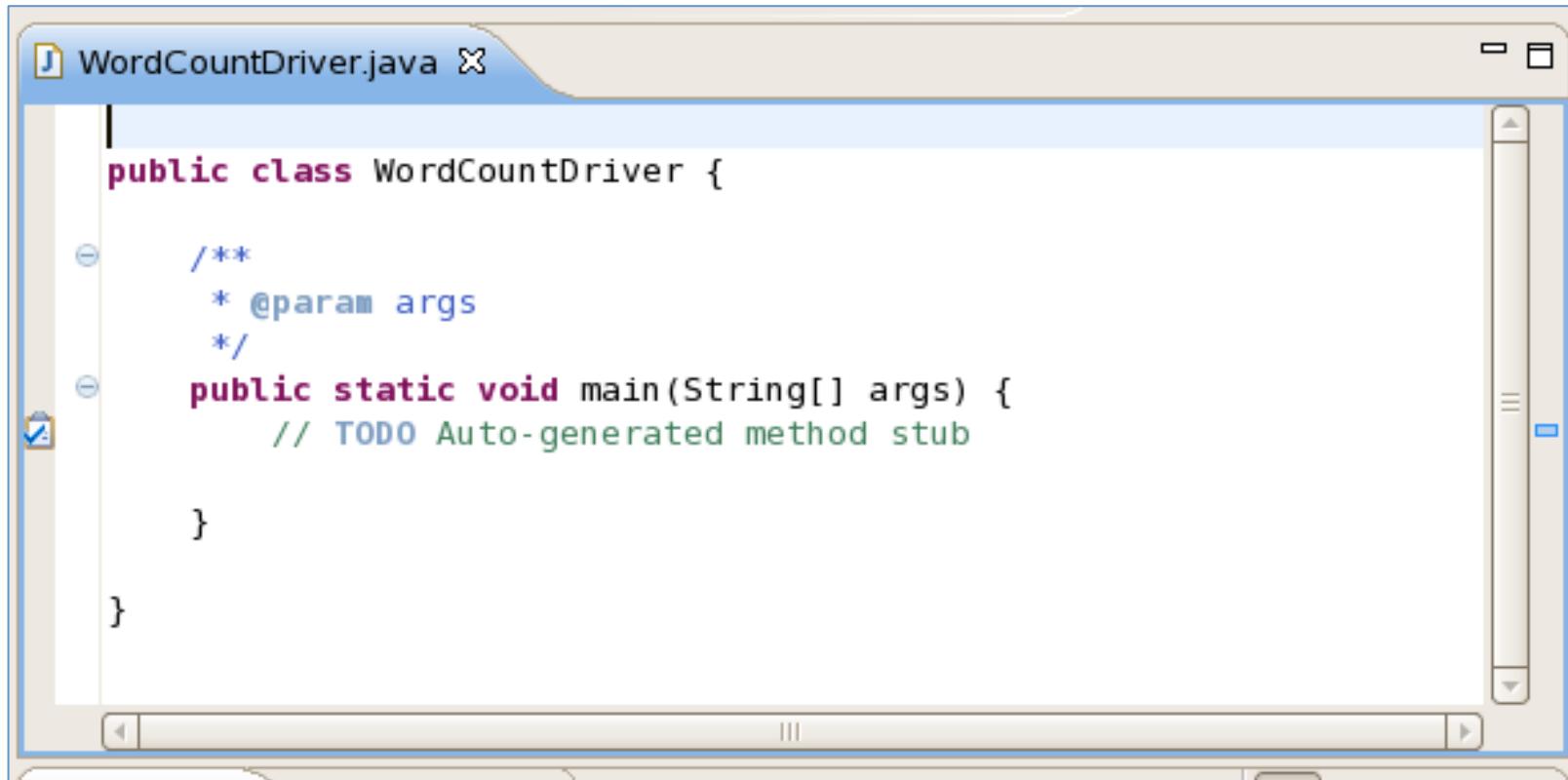
Using Eclipse (cont'd)

- Enter a class name, check the 'public static void main...' class, and hit Finish



Using Eclipse (cont'd)

- You can now edit your class



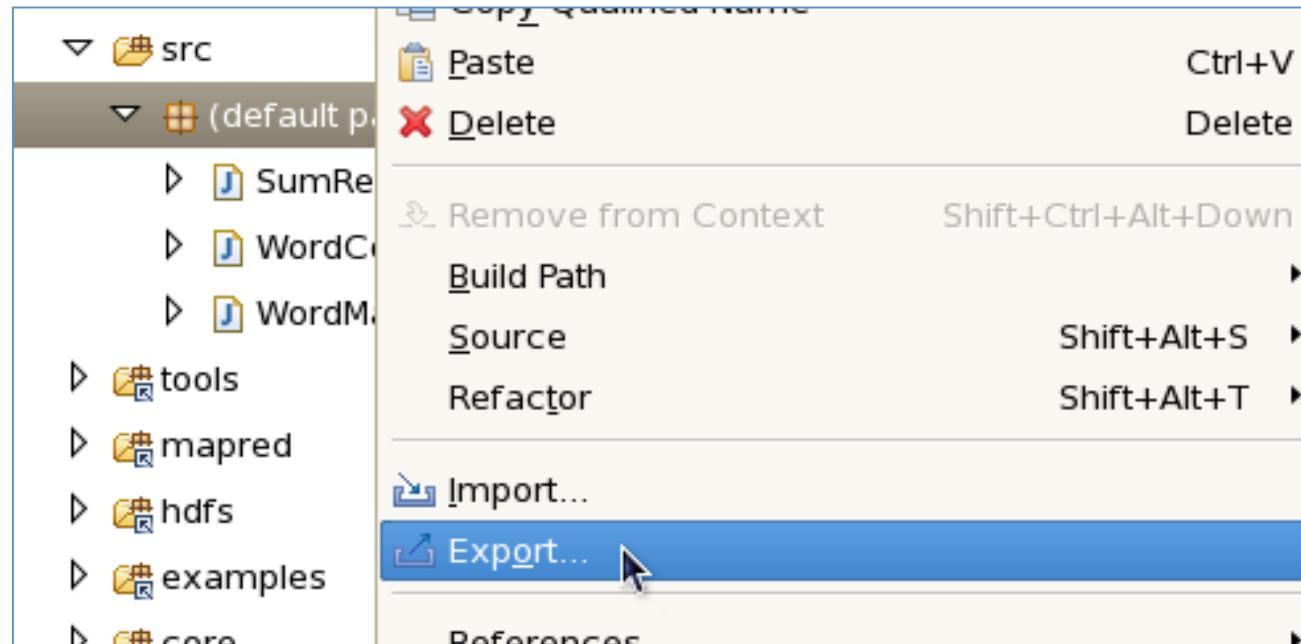
The screenshot shows the Eclipse Java IDE interface with a single open editor window titled "WordCountDriver.java". The code within the window is as follows:

```
public class WordCountDriver {  
    /**  
     * @param args  
     */  
    public static void main(String[] args) {  
        // TODO Auto-generated method stub  
    }  
}
```

The code editor features syntax highlighting and a vertical toolbar on the left side containing icons for cut, copy, paste, and other operations. The right side of the editor has scroll bars and a status bar at the bottom.

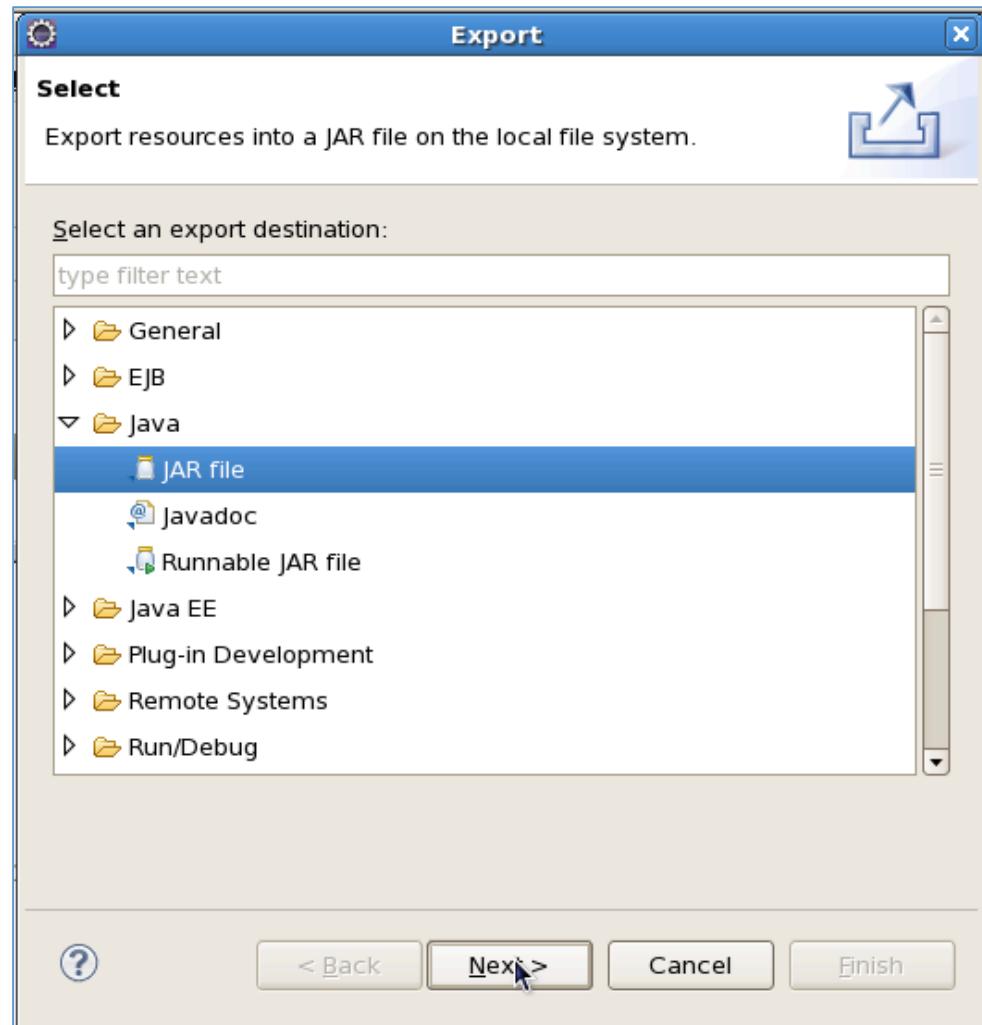
Using Eclipse (cont'd)

- Add other classes in the same way. When you are ready to test your code, right-click on the default package and choose Export



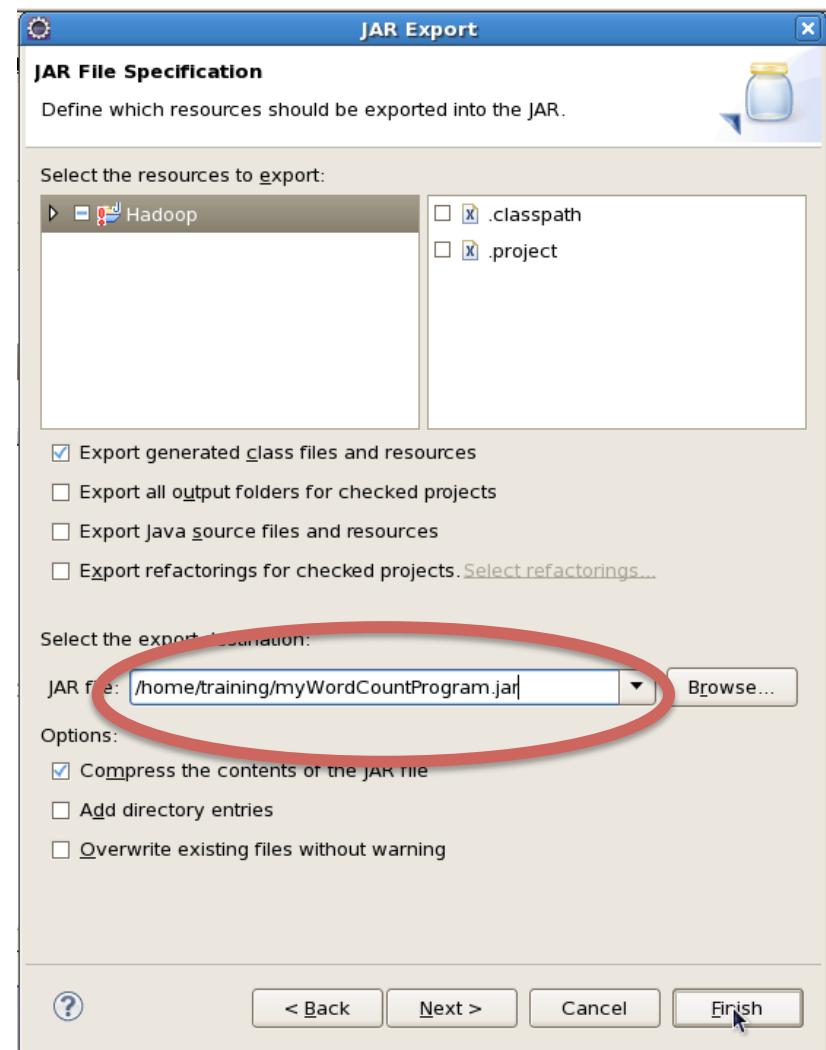
Using Eclipse (cont'd)

- Expand 'Java', and choose 'JAR file'. Then hit Next



Using Eclipse (cont'd)

- Enter a path and filename inside /home/training (your home directory), and hit Finish
- Your Jar file will be saved; you can now run it from the command line with the standard `hadoop jar...` command



Writing a MapReduce Program

The MapReduce Flow

Examining our Sample MapReduce program

The Driver Code

The Mapper

The Reducer

Hadoop's Streaming API

Using Eclipse for Rapid Development



Hands-On Exercise: Write a MapReduce program

The New MapReduce API

Conclusion

Hands-On Exercise: Write A MapReduce Program

- In this Hands-On Exercise, you will write a MapReduce program using either Java or Hadoop's Streaming interface
- Please refer to the PDF of exercise instructions, which can be found via the Desktop of the training Virtual Machine

Writing a MapReduce Program

The MapReduce Flow

Examining our Sample MapReduce program

The Driver Code

The Mapper

The Reducer

Hadoop's Streaming API

Using Eclipse for Rapid Development

Hands-On Exercise: Write a MapReduce program



The New MapReduce API

Conclusion

What Is The New API?

- When Hadoop 0.20 was released, a ‘New API’ was introduced
 - Designed to make the API easier to evolve in the future
 - Favors abstract classes over interfaces
- The ‘Old API’ was deprecated
- However, the New API is still not absolutely feature-complete in Hadoop 0.20.x
 - The Old API should not have been deprecated as quickly as it was
- Most developers still use the Old API
- All the code examples in this course use the Old API

Old API vs New API: Some Key Differences

Old API	New API
<pre>import org.apache.hadoop.mapred.*</pre>	<pre>import org.apache.hadoop.mapreduce.*</pre>
Driver code: <pre>JobConf conf = new JobConf(conf, Driver.class); conf.setSomeProperty(...); ... JobClient.runJob(conf);</pre>	Driver code: <pre>Configuration conf = new Configuration(); Job job = new Job(conf); job.setJarByClass(Driver.class); job.setSomeProperty(...); ... job.waitForCompletion(true);</pre>
Mapper: <pre>public class MyMapper extends MapReduceBase implements Mapper { public void map(Keytype k, Valuetype v, OutputCollector o, Reporter r) { ... o.collect(key, val); } }</pre>	Mapper: <pre>public class MyMapper extends Mapper { public void map(Keytype k, Valuetype v, Context c) { ... c.write(key, val); } }</pre>

Old API vs New API: Some Key Differences (cont'd)

Old API	New API
Reducer: <pre>public class MyReducer extends MapReduceBase implements Reducer { public void reduce(Keytype k, Iterator<Valuetype> v, OutputCollector o, Reporter r) { while(v.hasNext()) { // process v.next() o.collect(key, val); } } configure(JobConf job) (See next chapter) close() (See next chapter) }</pre>	Mapper: <pre>public class MyReducer extends Reducer { public void reduce(Keytype k, Iterable<Valuetype> v, Context c) { for(Valuetype v : eachval) { // process eachval c.write(key, val); } } setup(Context c) cleanup(Context c) }</pre>

Writing a MapReduce Program

The MapReduce Flow

Examining our Sample MapReduce program

The Driver Code

The Mapper

The Reducer

Hadoop's Streaming API

Using Eclipse for Rapid Development

Hands-On Exercise: Write a MapReduce program

The New MapReduce API



Conclusion

Conclusion

In this chapter you have learned

- **How to use the Hadoop API to write a MapReduce program in Java**
- **How to use the Streaming API to write Mappers and Reducers in other languages**
- **How to use Eclipse to speed up your Hadoop development**
- **The differences between the Old and New Hadoop APIs**



Chapter 5

Integrating Hadoop Into The Workflow

Course Chapters

- Introduction
- The Motivation For Hadoop
- Hadoop: Basic Contents
- Writing a MapReduce Program
- Integrating Hadoop Into The Workflow**
- Delving Deeper Into The Hadoop API
- Common MapReduce Algorithms
- An Introduction to Hive and Pig
- Practical Development Tips and Techniques
- More Advanced MapReduce Programming
- Joining Data Sets in MapReduce Jobs
- Graph Manipulation In MapReduce
- An Introduction to Oozie
- Conclusion
- Appendix: Cloudera Enterprise

Integrating Hadoop Into The Workflow

In this chapter you will learn

- How Hadoop can be integrating into an existing enterprise
- How to load data from an existing RDBMS into HDFS using Sqoop
- How to manage real-time data such as log files using Flume
- How to access HDFS from legacy systems with FuseDFS and Hoop

Integrating Hadoop Into The Workflow



Introduction

Relational Database Management Systems

Storage Systems

Importing Data From RDBMSs With Sqoop

Hands-On Exercise

Importing Real-Time Data With Flume

Accessing HDFS Using FuseDFS and Hoop

Conclusion

Introduction

- Your data center already has a lot of components
 - Database servers
 - Data warehouses
 - File servers
 - Backup systems
- How does Hadoop fit into this ecosystem?

Integrating Hadoop Into The Workflow

Introduction

 **Relational Database Management Systems**

Storage Systems

Importing Data From RDBMSs With Sqoop

Hands-On Exercise

Importing Real-Time Data With Flume

Accessing HDFS Using FuseDFS and Hoop

Conclusion

RDBMS Strengths

- **Relational Database Management Systems (RDBMSs) have many strengths**
 - Ability to handle complex transactions
 - Ability to process hundreds or thousands of queries per second
 - Real-time delivery of results
 - Simple but powerful query language

RDBMS Weaknesses

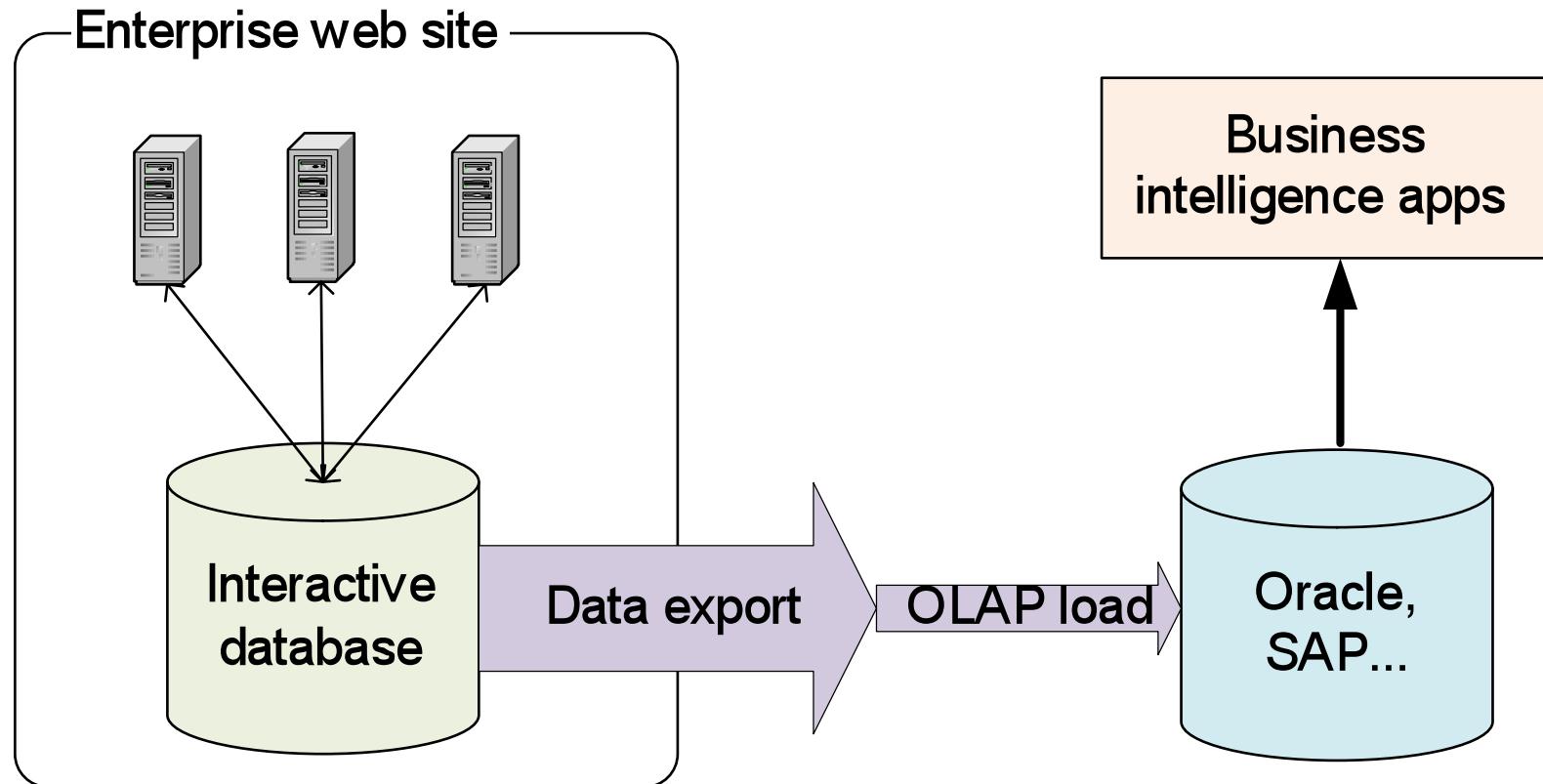
- **There are some areas where RDBMSs are less ideal**
 - Data schema is determined before data is ingested
 - Can make ad-hoc data collection difficult
 - Upper bound on data storage of 100s of Terabytes
 - Practical upper bound on data in a single query of 10s of Terabytes

Typical RDBMS Scenario

- **Typical scenario: use an interactive RDBMS to serve queries from a Web site etc**
- **Data is later extracted and loaded into a data warehouse for future processing and archiving**
 - Usually denormalized into an OLAP cube

OLAP: OnLine Analytical Processing

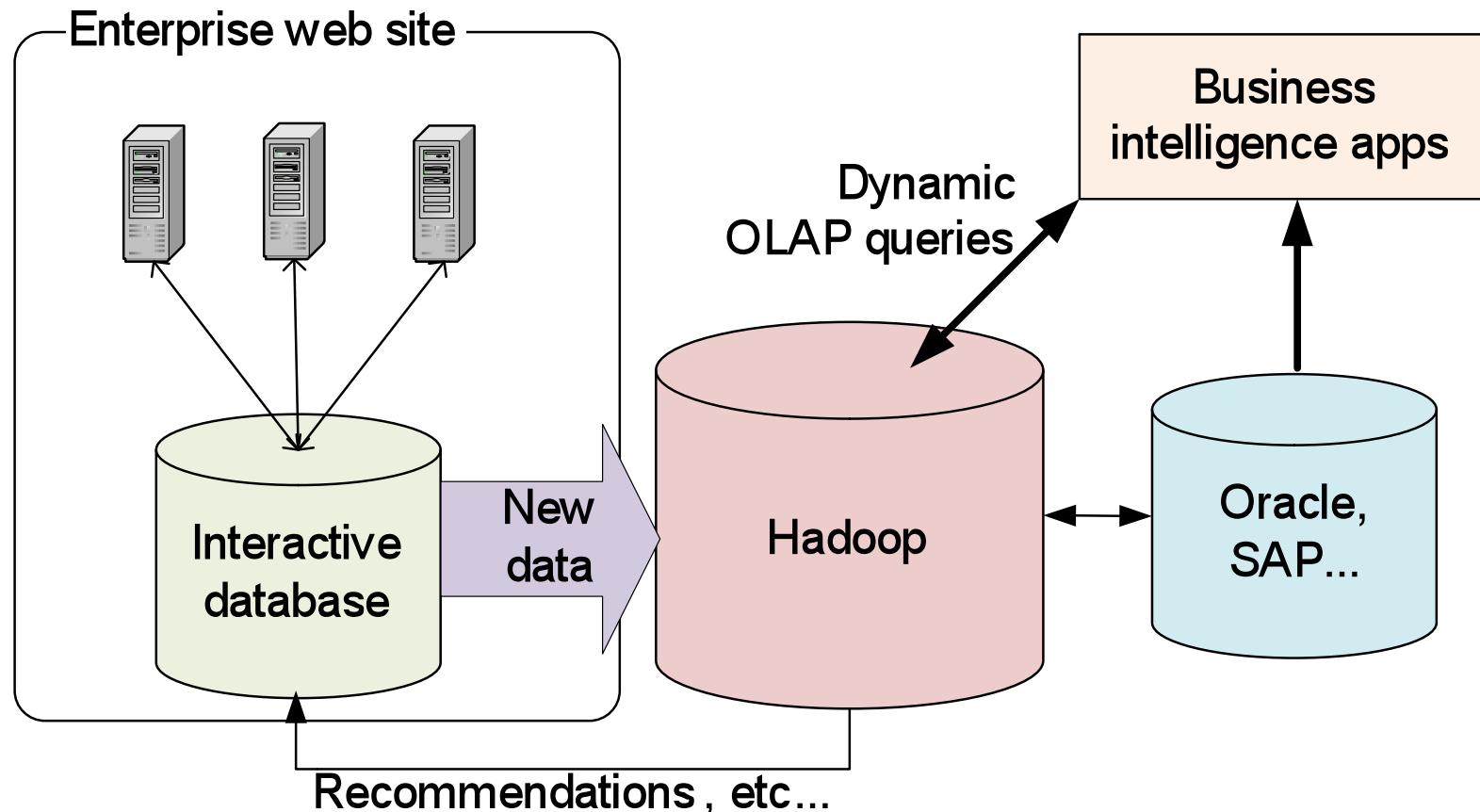
Typical RDBMS Scenario (cont'd)



OLAP Database Limitations

- **All dimensions must be prematerialized**
 - Re-materialization can be very time consuming
- **Daily data load-in times can increase**
 - Typically this leads to some data being discarded

Using Hadoop to Augment Existing Databases



Benefits of Hadoop

- **Processing power scales with data storage**
 - As you add more nodes for storage, you get more processing power ‘for free’
- **Views do not need prematerialization**
 - Ad-hoc full or partial dataset queries are possible
- **Total query size can be multiple Petabytes**

Hadoop Tradeoffs

- **Cannot serve interactive queries**
 - The fastest Hadoop job will still take several seconds to run
- **Less powerful updates**
 - No transactions
 - No modification of existing records

Integrating Hadoop Into The Workflow

Introduction

Relational Database Management Systems

 **Storage Systems**

Importing Data From RDBMSs With Sqoop

Hands-On Exercise

Importing Real-Time Data With Flume

Accessing HDFS Using FuseDFS and Hoop

Conclusion

Traditional High-Performance File Servers

- Enterprise data is often held on large fileservers
 - NetApp
 - EMC
 - Etc
- Advantages:
 - Fast random access
 - Many concurrent clients
- Disadvantages
 - High cost per Terabyte of storage

File Servers and Hadoop

- **Choice of destination medium depends on the expected access patterns**
 - Sequentially read, append-only data: HDFS
 - Random access: file server
- **HDFS can crunch sequential data faster**
- **Offloading data to HDFS leaves more room on file servers for ‘interactive’ data**
- **Use the right tool for the job!**

Integrating Hadoop Into The Workflow

Introduction

Relational Database Management Systems

Storage Systems

 **Importing Data From RDBMSs With Sqoop**

Hands-On Exercise

Importing Real-Time Data With Flume

Accessing HDFS Using FuseDFS and Hoop

Conclusion

Importing Data From an RDBMS to HDFS

- **Typical scenario: the need to use data stored in a Relational Database Management System or other data storage system (Oracle Database, MySQL, Teradata etc) in a MapReduce job**
 - Lookup tables
 - Legacy data
 - Etc
- **Possible to read directly from an RDBMS in your Mapper**
 - Can lead to the equivalent of a distributed denial of service (DDoS) attack on your RDBMS
 - In practice – don't do it!
- **Better scenario: import the data into HDFS beforehand**

Sqoop: SQL to Hadoop

- **Sqoop: open source tool written at Cloudera**
- **Imports tables from an RDBMS into HDFS**
 - Just one table
 - All tables in a database
 - Just portions of a table
 - Sqoop supports a WHERE clause
- **Uses MapReduce to actually import the data**
 - ‘Throttles’ the number of Mappers to avoid DDoS scenarios
 - Uses four Mappers by default
 - Value is configurable
- **Uses a JDBC interface**
 - Should work with any JDBC-compatible database

Sqoop: SQL to Hadoop (cont'd)

- **Imports data to HDFS as delimited text files or SequenceFiles**
 - Default is a comma-delimited text file
- **Can be used for incremental data imports**
 - First import retrieves all rows in a table
 - Subsequent imports retrieve just rows created since the last import
- **Generates a class file which can encapsulate a row of the imported data**
 - Useful for serializing and deserializing data in subsequent MapReduce jobs

Custom Swoop Connectors

- Cloudera has partnered with other organizations to create custom Swoop connectors
 - Use a system's native protocols to access data rather than JDBC
 - Provides much faster performance
- Current systems supported by custom connectors include:
 - Netezza
 - Teradata
 - MicroStrategy
 - Oracle Database (connector developed with Quest Software)
- Others are in development
- Custom connectors are not open-source, but are free
 - Available from the Cloudera Web site

Sqoop: Basic Syntax

- Standard syntax:

```
sqoop tool-name [tool-options]
```

- Tools include:

```
import  
import-all-tables  
list-tables
```

- Options include:

```
--connect  
--username  
--password
```

Sqoop: Example

- Example: import a table called ‘employees’ from a database called ‘personnel’ in a MySQL RDBMS

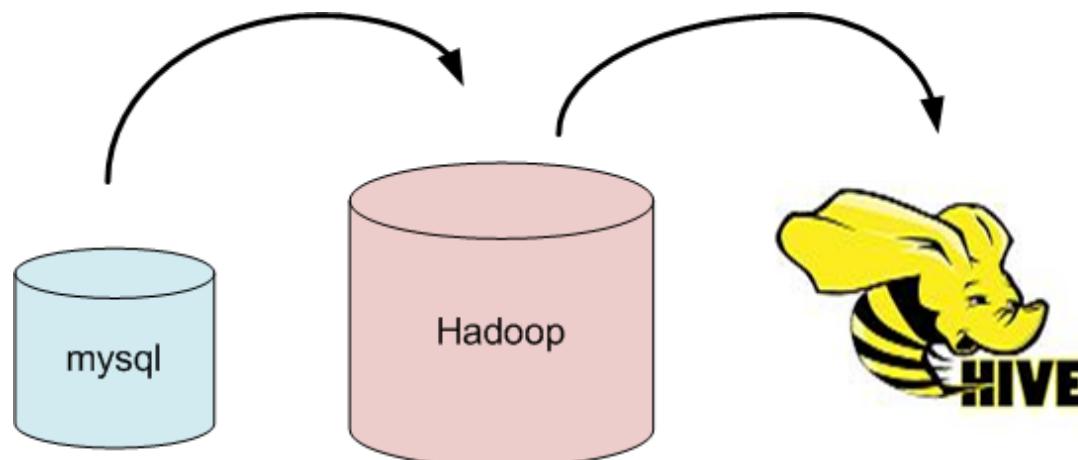
```
sqoop import --username fred --password derf \
--connect jdbc:mysql://database.example.com/personnel
--table employees
```

- Example: as above, but only records with an id greater than 1000

```
sqoop import --username fred --password derf \
--connect jdbc:mysql://database.example.com/personnel
--table employees
--where "id > 1000"
```

Sqoop: Importing To Hive Tables

- The Sqoop option `--hive-import` will automatically create a Hive table from the imported data
 - Imports the data
 - Generates the Hive `CREATE TABLE` statement
 - Runs the statement
 - Note: This will move the imported table into Hive's warehouse directory



Sqoop: Other Options

- Sqoop can take data from HDFS and insert it into an already-existing table in an RDBMS with the command

```
sqoop export [options]
```

- For general Sqoop help:

```
sqoop help
```

- For help on a particular command:

```
sqoop help command
```

Integrating Hadoop Into The Workflow

Introduction

Relational Database Management Systems

Storage Systems

Importing Data From RDBMSs With Sqoop

 **Hands-On Exercise**

Importing Real-Time Data With Flume

Accessing HDFS Using FuseDFS and Hoop

Conclusion

Hands-On Exercise: Importing Data

- In this Hands-On Exercise, you will import data into HDFS from MySQL
- Please refer to the Hands-On Exercise Manual

Integrating Hadoop Into The Workflow

Introduction

Relational Database Management Systems

Storage Systems

Importing Data From RDBMSs With Sqoop

Hands-On Exercise



Importing Real-Time Data With Flume

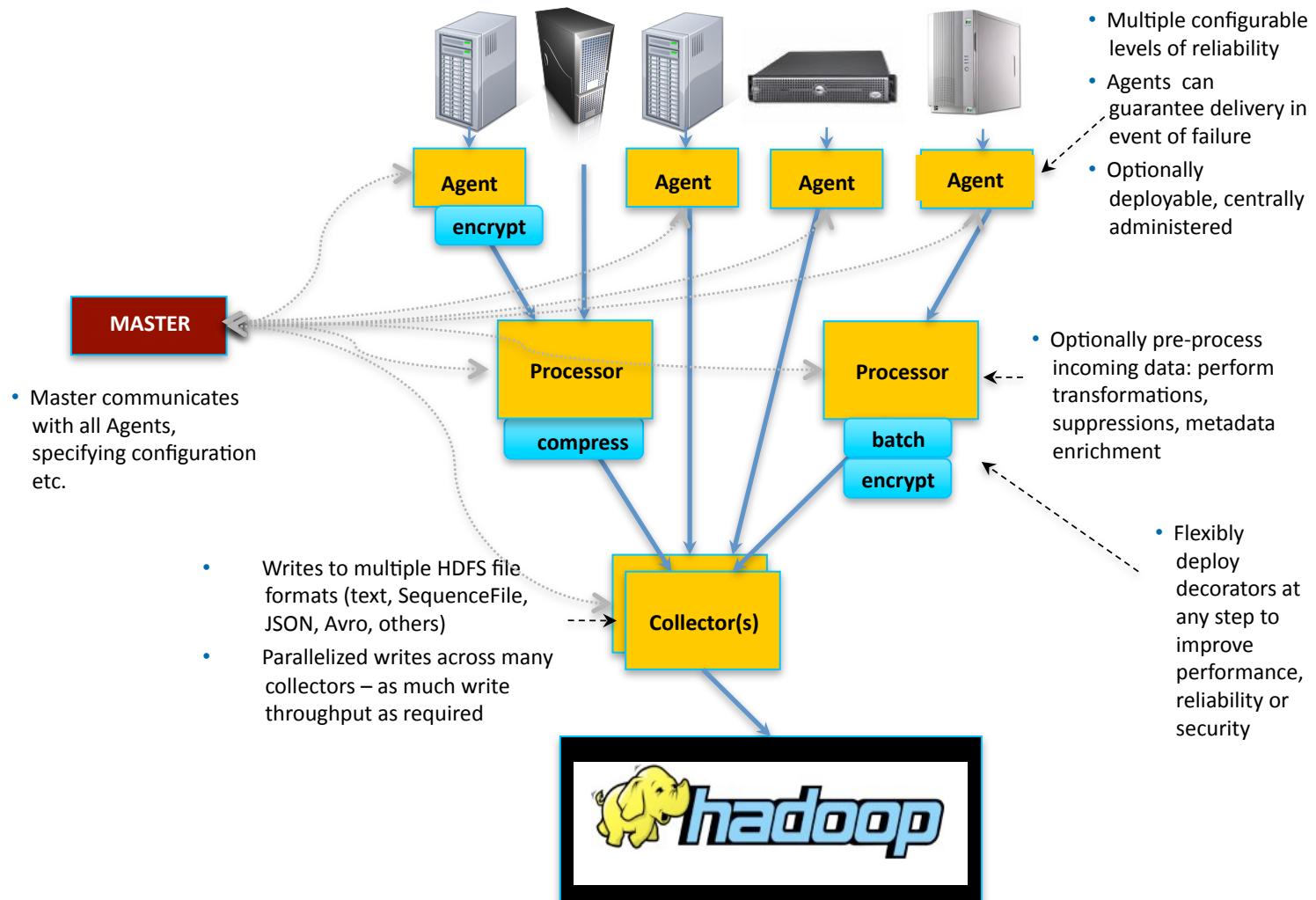
Accessing HDFS Using FuseDFS and Hoop

Conclusion

Flume: Basics

- **Flume is a distributed, reliable, available service for efficiently moving large amounts of data as it is produced**
 - Ideally suited to gathering logs from multiple systems and inserting them into HDFS as they are generated
- **Flume is Open Source**
 - Initially developed by Cloudera
- **Flume's design goals:**
 - Reliability
 - Scalability
 - Manageability
 - Extensibility

Flume: High-Level Overview



Flume Node Characteristics

- **Each Flume node has a *source* and a *sink***
- **Source**
 - Tells the node where to receive data from
- **Sink**
 - Tells the node where to send data to
- **Sink can have one or more *decorators***
 - Perform simple processing on the data as it passes through
 - Compression
 - awk, grep-like functionality
 - Etc

Flume's Design Goals: Reliability

- **Reliability**
 - The ability to continue delivering events in the face of system component failure
- **Provides user-configurable reliability guarantees**
 - End-to-end
 - Once Flume acknowledges receipt of an event, the event will eventually make it to the end of the pipeline
 - Store on failure
 - Nodes require acknowledgment of receipt from the node one hop downstream
 - Best effort
 - No attempt is made to confirm receipt of data from the node downstream

Flume's Design Goals: Scalability

- **Scalability**

- The ability to increase system performance linearly – or better – by adding more resources to the system
- Flume scales horizontally
 - As load increases, more machines can be added to the configuration

Flume's Design Goals: Manageability

- **Manageability**
 - The ability to control data flows, monitor nodes, modify the settings, and control outputs of a large system
- **Flume provides a central Master, where users can monitor data flows and reconfigure them on the fly**
 - Via a Web interface or a scriptable command-line shell

Flume's Design Goals: Extensibility

- **Extensibility**
 - The ability to add new functionality to a system
- **Flume can be extended by adding *connectors* to existing storage layers or data platforms**
 - General sources include data from files, syslog, and standard output from a process
 - General endpoints include files on the local filesystem or HDFS
 - Other connectors can be added
 - IRC
 - Twitter streams
 - HBase
 - Etc
 - Developers can write their own connectors in Java

Flume: Usage Patterns

- Flume is typically used to ingest log files from real-time systems such as Web servers, firewalls, mailservers etc into HDFS
- Currently in use in many large organizations, ingesting millions of events per day
 - At least one organization is using Flume to ingest over 200 million events per day
- Flume is typically installed and configured by a system administrator
 - Check the Flume documentation if you intend to install it yourself

Integrating Hadoop Into The Workflow

Introduction

Relational Database Management Systems

Storage Systems

Importing Data From RDBMSs With Sqoop

Hands-On Exercise

Importing Real-Time Data With Flume

 **Accessing HDFS Using FuseDFS and Hoop**

Conclusion

FuseDFS and Hoop: Motivation

- Many applications generate data which will ultimately reside in HDFS
- If Flume is not an appropriate solution for ingesting the data, some other method must be used
- Typically this is done as a batch process
- Problem: many legacy systems do not ‘understand’ HDFS
 - Difficult to write to HDFS if the application is not written in Java
 - May not have Hadoop installed on the system generating the data
- We need some way for these systems to access HDFS

FuseDFS

- **FuseDFS is based on FUSE (Filesystem in USEr space)**
- **Allows you to mount HDFS as a ‘regular’ filesystem**
- **Note: HDFS limitations still exist!**
 - Not intended as a general-purpose filesystem
 - Files are write-once
 - Not optimized for low latency
- **FuseDFS is included as part of the Hadoop distribution**

Hoop

- Hoop is an open-source project started at Cloudera
 - Distributed under the Apache Software License 2.0
- Provides an HTTP/HTTPS REST interface to HDFS
 - Supports both reads and writes from/to HDFS
 - Can be accessed from within a program
 - Can be used via command-line tools such as `curl`, `wget` etc
- Client accesses the Hoop server
 - Hoop server then accesses HDFS
- Available from <http://cloudera.github.com/hoop>

REST: REpresentational State Transfer

Integrating Hadoop Into The Workflow

Introduction

Relational Database Management Systems

Storage Systems

Importing Data From RDBMSs With Sqoop

Hands-On Exercise

Importing Real-Time Data With Flume

Accessing HDFS Using FuseDFS and Hoop



Conclusion

Conclusion

In this chapter you have learned

- **How Hadoop can be integrating into an existing enterprise**
- **How to load data from an existing RDBMS into HDFS using Sqoop**
- **How to manage real-time data such as log files using Flume**
- **How to access HDFS from legacy systems with FuseDFS and Hoop**



Chapter 6

Delving Deeper Into The Hadoop API

Course Chapters

- Introduction
- [The Motivation For Hadoop](#)
- Hadoop: Basic Contents
- Writing a MapReduce Program
- Integrating Hadoop Into The Workflow
- [Delving Deeper Into The Hadoop API](#)**
- Common MapReduce Algorithms
- An Introduction to Hive and Pig
- Practical Development Tips and Techniques
- More Advanced MapReduce Programming
- Joining Data Sets in MapReduce Jobs
- Graph Manipulation In MapReduce
- An Introduction to Oozie
- Conclusion
- Appendix: Cloudera Enterprise

Delving Deeper Into The Hadoop API

In this chapter you will learn

- More about ToolRunner
- How to write unit tests with MRUnit
- How to specify Combiners to reduce intermediate data
- How to use the configure and close methods for Map and Reduce setup and teardown
- How to write custom Partitioners for better load balancing
- How to directly access HDFS
- How to use the Distributed Cache

Delving Deeper Into The Hadoop API



More About ToolRunner

Testing With MRUnit

Reducing Intermediate Data with Combiners

The `configure` and `close` Methods

Writing Partitioners for Better Load Balancing

Directly Accessing HDFS

Using The DistributedCache

Hands-On Exercise

Conclusion

Why Use ToolRunner?

- In Chapter 4, we used ToolRunner in our driver class
 - This is not actually required, but is a best practice
- ToolRunner uses the GenericOptionsParser class internally
 - Allows you to specify configuration options on the command line
 - Also allows you to specify items for the Distributed Cache on the command line (see later)

ToolRunner Command Line Options

- ToolRunner allows the user to specify configuration options on the command line
- Commonly used to specify configuration settings with the **-D** flag
 - Will override any default or site properties in the configuration

```
hadoop jar myjar.jar MyDriver -D mapred.reduce.tasks=10
```

- Note: this will *not* override configurations set in the driver code itself
- Can specify an XML configuration file with **-conf**
- Can specify the default filesystem with **-fs uri**
 - Shortcut for **-D fs.default.name = uri**

Delving Deeper Into The Hadoop API

More About ToolRunner

 **Testing With MRUnit**

Reducing Intermediate Data with Combiners

The `configure` and `close` Methods

Writing Partitioners for Better Load Balancing

Directly Accessing HDFS

Using The DistributedCache

Hands-On Exercise

Conclusion

An Introduction to Unit Testing

- A ‘unit’ is a small piece of your code
 - A small piece of functionality
- A unit test verifies the correctness of that unit of code
 - A purist might say that in a well-written unit test, only a single ‘thing’ should be able to fail
 - Generally accepted rule-of-thumb: a unit test should take less than a second to complete

Why Write Unit Tests?

- **Unit testing provides verification that your code is functioning correctly**
- **Much faster than testing your entire program each time you modify the code**
 - Fastest MapReduce job on a cluster will take at least 15 seconds
 - Even in pseudo-distributed mode
 - Even running in LocalJobRunner mode will take several seconds
 - LocalJobRunner mode is discussed later in the course
 - Unit tests help you iterate faster in your code development

Why MRUnit?

- JUnit is a very popular Java unit testing framework
- Problem: JUnit cannot be used directly to test Mappers or Reducers
 - Unit tests require mocking up InputSplits, OutputCollector, Reporter, ...
 - A lot of work
- MRUnit is built on top of JUnit
 - Provides those mock objects
- Allows you to test your code from within an IDE
 - Much easier to debug

JUnit Basics

- We are using JUnit 4 in class
 - Earlier versions would also work
- @Test
 - Java annotation
 - Indicates that this method is a test which JUnit should execute
- @Before
 - Java annotation
 - Tells JUnit to call this method before every @Test method
 - Two @Test methods would result in the @Before method being called twice

JUnit Basics (cont'd)

- **JUnit test methods:**
 - `assertEquals()`, `assertNotNull()` etc
 - Fail if the conditions of the statement are not met
 - `fail(msg)`
 - Fails the test with the given error message
- **With a JUnit test open in Eclipse, run all tests in the class by going to Run -> Run**
- **Eclipse also provides functionality to run all JUnit tests in your project**
- **Other IDEs have similar functionality**

JUnit: Example Code

```
package hadoop.dev.api.mrunit.slides;

import static org.junit.Assert.assertEquals;

import org.junit.Before;
import org.junit.Test;

public class JUnitHelloWorld {
    protected String s;
    @Before
    public void setup() {
        s = "HELLO WORLD";
    }
    @Test
    public void testHelloWorldSuccess() {
        s = s.toLowerCase();
        assertEquals("hello world", s);
    }
    // will fail even if testHelloWorldSuccess is called first
    @Test
    public void testHelloWorldFail() {
        assertEquals("hello world", s);
    }
}
```

Using MRUnit to Test MapReduce Code

- MRUnit builds on top of JUnit
- Provides a mock InputSplit, mock Reporter, mock OutputCollector etc.
- Can test just the Mapper, just the Reducer, or the full MapReduce flow

MRUnit: Example Code – Mapper Unit Test

```
package hadoop.dev.api.mrunit.slides;

import hadoop.dev.wordcount.mapred.WordMapper;

import org.apache.hadoop.io.IntWritable;
import org.apache.hadoop.io.LongWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mrunit.MapDriver;
import org.junit.Before;
import org.junit.Test;

public class MRUnitHelloWorld {
    MapDriver<LongWritable, Text, Text, IntWritable> mapDriver;

    @Before
    public void setUp() {
        WordMapper mapper = new WordMapper();
        mapDriver = new MapDriver<LongWritable, Text, Text, IntWritable>();
        mapDriver.setMapper(mapper);
    }

    @Test
    public void testMapper() {
        mapDriver.withInput(new LongWritable(1), new Text("cat dog"));
        mapDriver.withOutput(new Text("cat"), new IntWritable(1));
        mapDriver.withOutput(new Text("dog"), new IntWritable(1));
        mapDriver.runTest();
    }
}
```

MRUnit: Example Code Mapper Unit Test(cont'd)

```
package hadoop.dev.api.mrunit.slides;

import hadoop.dev.wordcount.mapred.WordMapper;

import org.apache.hadoop.io.IntWritable;
import org.apache.hadoop.io.LongWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mrunit.MapDriver;
import org.junit.Before;
import org.junit.Test;

public class WordMapperTest {
    MapDriver<LongWritable, Text, Text, IntWritable> mapDriver;
    WordMapper wordMapper;

    @Before
    public void setup() {
        mapDriver = new MapDriver<LongWritable, Text, Text, IntWritable>();
        wordMapper = new WordMapper();
    }

    @Test
    public void testMapper() {
        mapDriver.withInput(new LongWritable(1), new Text("cat dog"));
        mapDriver.withOutput(new Text("cat"), new IntWritable(1));
        mapDriver.withOutput(new Text("dog"), new IntWritable(1));
        mapDriver.runTest();
    }
}
```

Import the relevant JUnit classes and the MRUnit MapDriver class as we will be writing a unit test for our Mapper

MRUnit: Example Code Mapper Unit Test(cont'd)

```
package hadoop.dev.api.mrunit.slides;

import hadoop.dev.wordcount.mapred.WordMapper;

import org.apache.hadoop.io.IntWritable;
import org.apache.hadoop.io.LongWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mrunit.MapDriver;
import org.junit.Before;
import org.junit.Test;

public class MRUnitHelloWorld {
    MapDriver<LongWritable, Text, Text, IntWritable> mapDriver;

    @Before
    public void setup() {
        WordMapper wordMapper = new WordMapper();
        mapDriver = MapDriver.newMapDriver(wordMapper);
    }

    @Test
    public void testMapper() {
        mapDriver.withInput(new LongWritable(1), new Text("cat dog"));
        mapDriver.withOutput(new Text("cat"), new IntWritable(1));
        mapDriver.withOutput(new Text("dog"), new IntWritable(1));
        mapDriver.runTest();
    }
}
```

MapDriver is an MRUnit class (not a user-defined driver)

MRUnit: Example Code Mapper Unit Test(cont'd)

```
package hadoop.dev.api.mrunit.slides;

import hadoop.dev.wordcount.mapred.WordMapper;

import org.apache.hadoop.io.IntWritable;
import org.apache.hadoop.io.LongWritable;
import org.junit.*;
import org.junit.rules.*;
import org.junit.runners.*;

public class MapDriverTest {
    MapDriver<LongWritable, Text, Text, IntWritable> mapDriver;
    WordMapper mapper;

    @Before
    public void setUp() {
        mapper = new WordMapper();
        mapDriver = new MapDriver<LongWritable, Text, Text, IntWritable>();
        mapDriver.setMapper(mapper);
    }

    @Test
    public void testMapper() {
        mapDriver.withInput(new LongWritable(1), new Text("cat dog"));
        mapDriver.withOutput(new Text("cat"), new IntWritable(1));
        mapDriver.withOutput(new Text("dog"), new IntWritable(1));
        mapDriver.runTest();
    }
}
```

Set up the test. This method will be called before every test, just as with JUnit.

MRUnit: Example Code Mapper Unit Test(cont'd)

```
package hadoop.dev.api.mrunit.slides;

import hadoop.dev.wordcount.mapred.WordMapper;

import org.apache.hadoop.io.IntWritable;
import org.apache.hadoop.io.LongWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mrunit.MapDriver;
import org.junit.Before;
import org.junit.Test;

public class MRUnitHelloWorld {
    MapDriver<LongWritable, Text, Text, IntWritable> mapDriver;

    @Before
    public void setUp() {
        WordMapper wordMapper = new WordMapper();
        mapDriver = MapDriver.createMapDriver(wordMapper);
    }

    @Test
    public void testMapper() {
        mapDriver.withInput(new LongWritable(1), new Text("cat dog"));
        mapDriver.withOutput(new Text("cat"), new IntWritable(1));
        mapDriver.withOutput(new Text("dog"), new IntWritable(1));
        mapDriver.runTest();
    }
}
```

The test itself.

MRUnit Drivers

- **MRUnit has a MapDriver, a ReduceDriver, and a MapReduceDriver**
- **Methods:**
 - `withInput`
 - Specifies input to the Mapper/Reducer
 - `withOutput`
 - Specifies expected output from the Mapper/Reducer
 - `with{Input,Output}` support the builder method and can be **chained**
 - `add{Input,Output}`
 - Similar to `with{Input,Output}` but returns `void`
 - `runTest`
 - Runs the test
 - `run`
 - Runs the test and allows you to retrieve the results

MRUnit Drivers (cont'd)

- Drivers take a single (key, value) pair as input
- Can take multiple (key, value) pairs as expected output
- If you are calling `driver.runTest()` multiple times, call `driver.resetOutput()` between each call
 - MRUnit will fail if you do not do this

Creating Failures, Not Errors

- MRUnit throws a RuntimeException when a test fails
- In JUnit terms, this means the test has ended as ‘erred’ rather than ‘failed’
- Best practice is to have a failed test be in failed rather than erred state
 - This is not required

```
@Test
public void testMapper() {
    mapDriver.withInput(new LongWritable(1), new Text("cat dog"));
    mapDriver.withOutput(new Text("cat"), new IntWritable(-1));
    mapDriver.withOutput(new Text("dog"), new IntWritable(1));
    try {
        mapDriver.runTest();
    } catch (Exception e) {
        fail(e.getMessage());
    }
}
```

MRUnit Conclusions

- You should write unit tests for your code!
- As you are performing the Hands-On Exercises in the rest of the course we strongly recommend that you write unit tests as you proceed
 - This will help greatly in debugging your code
 - Your instructor may ask to see your unit tests if you ask for assistance!

Delving Deeper Into The Hadoop API

More About ToolRunner

Testing With MRUnit

 Reducing Intermediate Data with Combiners

The `configure` and `close` Methods

Writing Partitioners for Better Load Balancing

Directly Accessing HDFS

Using The DistributedCache

Hands-On Exercise

Conclusion

Recap: Combiners

- Recall that a Combiner is like a ‘mini-Reducer’
 - Optional
 - Runs on the output from a single Mapper
 - Combiner’s output is then sent to the Reducer
- Combiners often use the same code as the Reducer
 - Only if the operation is commutative and associative
 - In this case, input and output data types for the Combiner/Reducer must be identical
- VERY IMPORTANT: Never put code in the Combiner that must be run as part of your MapReduce job
 - The Combiner may not be run on the output from some or all of the Mappers

Specifying a Combiner

- To specify the Combiner class to be used in your MapReduce code, put the following line in your Driver:

```
conf.setCombinerClass(YourCombinerClass.class);
```

- The Combiner uses the same interface as the Reducer
 - Takes in a key and a list of values
 - Outputs zero or more (key, value) pairs
 - The actual method called is the `reduce` method in the class

Delving Deeper Into The Hadoop API

More About ToolRunner

Testing With MRUnit

Reducing Intermediate Data with Combiners

 **The configure and close Methods**

Writing Partitioners for Better Load Balancing

Directly Accessing HDFS

Using The DistributedCache

Hands-On Exercise

Conclusion

The `configure` Method

- It is common to want your Mapper or Reducer to execute some code before the `map` or `reduce` method is called
 - Initialize data structures
 - Read data from an external file
 - Set parameters
 - Etc
- The `configure` method is run before the `map` or `reduce` method is called for the first time

```
public void configure(JobConf conf)
```

The close Method

- Similarly, you may wish to perform some action(s) after all the records have been processed by your Mapper or Reducer
- The `close` method is called before the Mapper or Reducer terminates

```
public void close() throws IOException
```

- Note that the `close()` method does not receive the `JobConf` object
 - You could save a reference to the `JobConf` object in the `configure` method and use it in the `close` method if necessary

Passing Parameters: The Wrong Way!

```
public class MyClass {  
    private static int param;  
    ...  
    private static class MyMapper extends MapReduceBase ... {  
        public void map... {  
            int v = param;  
        }  
    }  
    ...  
    public static void main(String[] args) throws IOException {  
        JobConf conf = new JobConf(MyClass.class);  
        param = 5;  
        ...  
        JobClient.runJob(conf);  
    }  
}
```

Passing Parameters: The Right Way

```
public class MyClass {  
    ...  
    private static class MyMapper extends MapReduceBase ... {  
        public void configure(JobConf job) {  
            int v = job.getInt("param", 0);  
            ...  
        }  
        ...  
        public void map...  
    }  
    ...  
    public static void main(String[] args) throws IOException {  
        JobConf conf = new JobConf(MyClass.class);  
        conf.setInt("param", 5)  
        ...  
        JobClient.runJob(conf);  
    }  
}
```

Delving Deeper Into The Hadoop API

More About ToolRunner

Testing With MRUnit

Reducing Intermediate Data with Combiners

The configure and close Methods

 **Writing Partitioners for Better Load Balancing**

Directly Accessing HDFS

Using The DistributedCache

Hands-On Exercise

Conclusion

What Does The Partitioner Do?

- **The Partitioner divides up the keyspace**
 - Controls which Reducer each intermediate key and its associated values goes to
- **Often, the default behavior is fine**
 - Default is the HashPartitioner

```
public class HashPartitioner<K2, V2> implements Partitioner<K2, V2> {

    public void configure(JobConf job) {}

    public int getPartition(K2 key, V2 value,
                           int numReduceTasks) {
        return (key.hashCode() & Integer.MAX_VALUE) % numReduceTasks;
    }

}
```

Custom Partitioners

- Sometimes you will need to write your own Partitioner
- Example: your key is a custom WritableComparable which contains a pair of values (a, b)
 - You may decide that all keys with the same value for a need to go to the same Reducer
 - The default Partitioner is not sufficient in this case

Custom Partitioners (cont'd)

- **Custom Partitioners are needed when performing a secondary sort (see later)**
- **Custom Partitioners are also useful to avoid potential performance issues**
 - To avoid one Reducer having to deal with many very large lists of values
 - Example: in our word count job, we wouldn't want a single reduce dealing with all the three- and four-letter words, while another only had to handle 10- and 11-letter words

Creating a Custom Partitioner

- To create a custom partitioner:

1. Create a class for the partitioner

- Should implement the Partitioner interface

2. Create a method in the class called `getPartition`

- Receives the key, the value, and the number of Reducers
- Should return an int between 0 and one less than the number of Reducers
 - e.g., if it is told there are 10 Reducers, it should return an int between 0 and 9

3. Specify the custom partitioner in your driver code

```
conf.setPartitionerClass(MyPartitioner.class);
```

Delving Deeper Into The Hadoop API

More About ToolRunner

Testing With MRUnit

Reducing Intermediate Data with Combiners

The configure and close Methods

Writing Partitioners for Better Load Balancing

 **Directly Accessing HDFS**

Using The DistributedCache

Hands-On Exercise

Conclusion

Accessing HDFS Programmatically

- In addition to using the command-line shell, you can access HDFS programmatically
 - Useful if your code needs to read or write ‘side data’ in addition to the standard MapReduce inputs and outputs
- Beware: HDFS is not a general-purpose filesystem!
 - Files cannot be modified once they have been written, for example
- Hadoop provides the FileSystem abstract base class
 - Provides an API to generic file systems
 - Could be HDFS
 - Could be your local file system
 - Could even be, e.g., Amazon S3

The FileSystem API

- In order to use the FileSystem API, retrieve an instance of it

```
Configuration conf = new Configuration();
FileSystem fs = FileSystem.get(conf);
```

- The `conf` object has read in the Hadoop configuration files, and therefore knows the address of the NameNode etc.
- A file in HDFS is represented by a `Path` object

```
Path p = new Path("/path/to/my/file");
```

The FileSystem API (cont'd)

- Some useful API methods:

- `FSDataOutputStream create(...)`
 - Extends `java.io.DataOutputStream`
 - Provides methods for writing primitives, raw bytes etc
- `FSDataInputStream open(...)`
 - Extends `java.io.DataInputStream`
 - Provides methods for reading primitives, raw bytes etc
- `boolean delete(...)`
- `boolean mkdirs(...)`
- `void copyFromLocalFile(...)`
- `void copyToLocalFile(...)`
- `FileStatus[] listStatus(...)`

The FileSystem API: Directory Listing

- Get a directory listing:

```
Path p = new Path("/my/path");

Configuration conf = new Configuration();
FileSystem fs = FileSystem.get(conf);
FileStatus[] fileStats = fs.listStatus(p);

for (int i = 0; i < fileStats.length; i++) {
    Path f = fileStats[i].getPath();

    // do something interesting
}
```

The FileSystem API: Writing Data

- Write data to a file

```
Configuration conf = new Configuration();
FileSystem fs = FileSystem.get(conf);

Path p = new Path("/my/path/foo");

FSDataOutputStream out = fs.create(path, false);

// write some raw bytes
out.writegetBytes();

// write an int
out.writeInt(getInt());

...
out.close();
```

Delving Deeper Into The Hadoop API

More About ToolRunner

Testing With MRUnit

Reducing Intermediate Data with Combiners

The configure and close Methods

Writing Partitioners for Better Load Balancing

Directly Accessing HDFS

 **Using The DistributedCache**

Hands-On Exercise

Conclusion

The Distributed Cache: Motivation

- A common requirement is for a Mapper or Reducer to need access to some ‘side data’
 - Lookup tables
 - Dictionaries
 - Standard configuration values
 - Etc
- One option: read directly from HDFS in the `configure` method
 - Works, but is not scalable
- The `DistributedCache` provides an API to push data to all slave nodes
 - Transfer happens behind the scenes before any task is executed
 - Note: `DistributedCache` is read only
 - Files in the `DistributedCache` are automatically deleted from slave nodes when the job finishes

Using the DistributedCache: The Difficult Way

- Place the files into HDFS
- Configure the DistributedCache in your driver code

```
JobConf job = new JobConf();
DistributedCache.addCacheFile(new URI("/myapp/lookup.dat"), job);
DistributedCache.addFileToClassPath(new Path("/myapp/mylib.jar"), job);
DistributedCache.addCacheArchive(new URI("/myapp/map.zip"), job);
DistributedCache.addCacheArchive(new URI("/myapp/mytar.tar"), job);
DistributedCache.addCacheArchive(new URI("/myapp/mytgz.tgz"), job);
DistributedCache.addCacheArchive(new URI("/myapp/mytargz.tar.gz"), job);
```

- .jar files added with addFileToClassPath will be added to your Mapper or Reducer's classpath
- Files added with addCacheArchive will automatically be dearchived/decompressed

Using the DistributedCache: The Easy Way

- If you are using ToolRunner, you can add files to the DistributedPath directly from the command line when you run the job
 - No need to copy the files to HDFS first
- Use the **-files** option to add files

```
hadoop jar myjar.jar MyDriver -files file1, file2, file3, ...
```

- The **-archives** flag adds archived files, and automatically unarchives them on the destination machines
- The **-libjars** flag adds jar files to the classpath

Accessing Files in the DistributedCache

- Files added to the DistributedCache are made available in your task's local working directory
 - Access them from your Mapper or Reducer the way you would read any ordinary local file

```
File f = new File("file_name_here");
```

Delving Deeper Into The Hadoop API

More About ToolRunner

Testing With MRUnit

Reducing Intermediate Data with Combiners

The configure and close Methods

Writing Partitioners for Better Load Balancing

Directly Accessing HDFS

Using The DistributedCache

 **Hands-On Exercise**

Conclusion

Hands-On Exercise

- In this Hands-On Exercise, you will gain practice writing combiners, creating Unit Tests, and accessing the DistributedCache
- Please refer to the Hands-On Exercise Manual

Delving Deeper Into The Hadoop API

More About ToolRunner

Testing With MRUnit

Reducing Intermediate Data with Combiners

The configure and close Methods

Writing Partitioners for Better Load Balancing

Directly Accessing HDFS

Using The DistributedCache

Hands-On Exercise



Conclusion

Conclusion

In this chapter you have learned

- **More about ToolRunner**
- **How to write unit tests with MRUnit**
- **How to specify Combiners to reduce intermediate data**
- **How to use the configure and close methods for Map and Reduce setup and teardown**
- **How to write custom Partitioners for better load balancing**
- **How to directly access HDFS**
- **How to use the Distributed Cache**



Chapter 7

Common MapReduce Algorithms

Course Chapters

- Introduction
- The Motivation For Hadoop
- Hadoop: Basic Contents
- Writing a MapReduce Program
- Integrating Hadoop Into The Workflow
- Delving Deeper Into The Hadoop API
- Common MapReduce Algorithms**
- An Introduction to Hive and Pig
- Practical Development Tips and Techniques
- More Advanced MapReduce Programming
- Joining Data Sets in MapReduce Jobs
- Graph Manipulation In MapReduce
- An Introduction to Oozie
- Conclusion
- Appendix: Cloudera Enterprise

Common MapReduce Algorithms

In this chapter you will learn

- Some typical MapReduce algorithms, including
 - Sorting
 - Searching
 - Indexing
 - Term Frequency – Inverse Document Frequency
 - Word Co-Occurrence
- We will also briefly discuss machine learning with Hadoop

Common MapReduce Algorithms



Introduction

Sorting and Searching

Indexing

Machine Learning

TF-IDF

Word Co-Occurrence

Hands-On Exercise

Conclusion

Introduction

- MapReduce jobs tend to be relatively short in terms of lines of code
- It is typical to combine multiple small MapReduce jobs together in a single workflow
 - Often using Oozie (see later)
- You are likely to find that many of your MapReduce jobs use very similar code
- In this chapter we present some very common MapReduce algorithms
 - These algorithms are frequently the basis for more complex MapReduce jobs

Common MapReduce Algorithms

Introduction

 **Sorting and Searching**

Indexing

Machine Learning

TF-IDF

Word Co-Occurrence

Hands-On Exercise

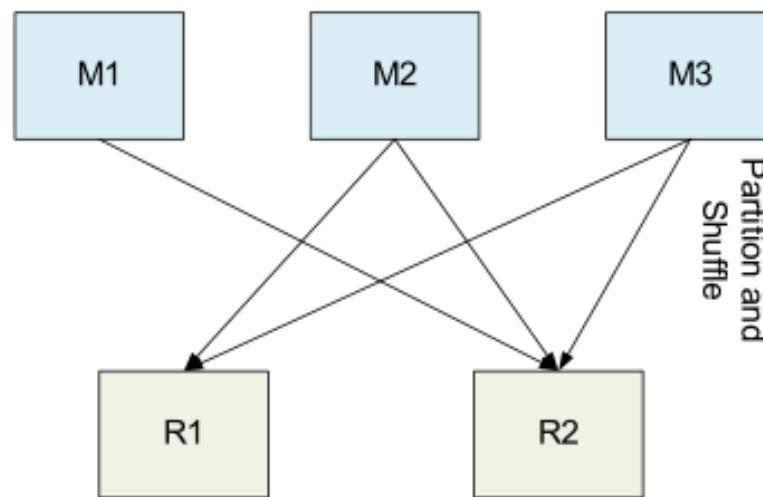
Conclusion

Sorting

- **MapReduce is very well suited to sorting large data sets**
- **Recall: keys are passed to the reducer in sorted order**
- **Assuming the file to be sorted contains lines with a single value:**
 - Mapper is merely the identity function for the value
 $(k, v) \rightarrow (v, \underline{\hspace{2cm}})$
 - Reducer is the identity function
 $(k, \underline{\hspace{2cm}}) \rightarrow (k, '')$

Sorting (cont'd)

- Trivial with a single reducer
- For multiple reducers, need to choose a partitioning function such that if $k_1 < k_2$, $\text{partition}(k_1) \leq \text{partition}(k_2)$



Sorting as a Speed Test of Hadoop

- **Sorting is frequently used as a speed test for a Hadoop cluster**
 - Mapper and Reducer are trivial
 - Therefore sorting is effectively testing the Hadoop framework's I/O
- **Good way to measure the increase in performance if you enlarge your cluster**
 - Run and time a sort job before and after you add more nodes
 - `terasort` is one of the sample jobs provided with Hadoop
 - Creates and sorts very large files

Searching

- Assume the input is a set of files containing lines of text
- Assume the Mapper has been passed the pattern for which to search as a special parameter
 - We saw how to pass parameters to your Mapper in the previous chapter
- Algorithm:
 - Mapper compares the line against the pattern
 - If the pattern matches, Mapper outputs (line, _)
 - Or (filename+line, _), or ...
 - If the pattern does not match, Mapper outputs nothing
 - Reducer is the Identity Reducer
 - Just outputs each intermediate key

Common MapReduce Algorithms

Introduction

Sorting and Searching



Indexing

Machine Learning

TF-IDF

Word Co-Occurrence

Hands-On Exercise

Conclusion

Indexing

- **Assume the input is a set of files containing lines of text**
- **Key is the byte offset of the line, value is the line itself**
- **We can retrieve the name of the file using the Reporter object**
 - More details on how to do this later

Inverted Index Algorithm

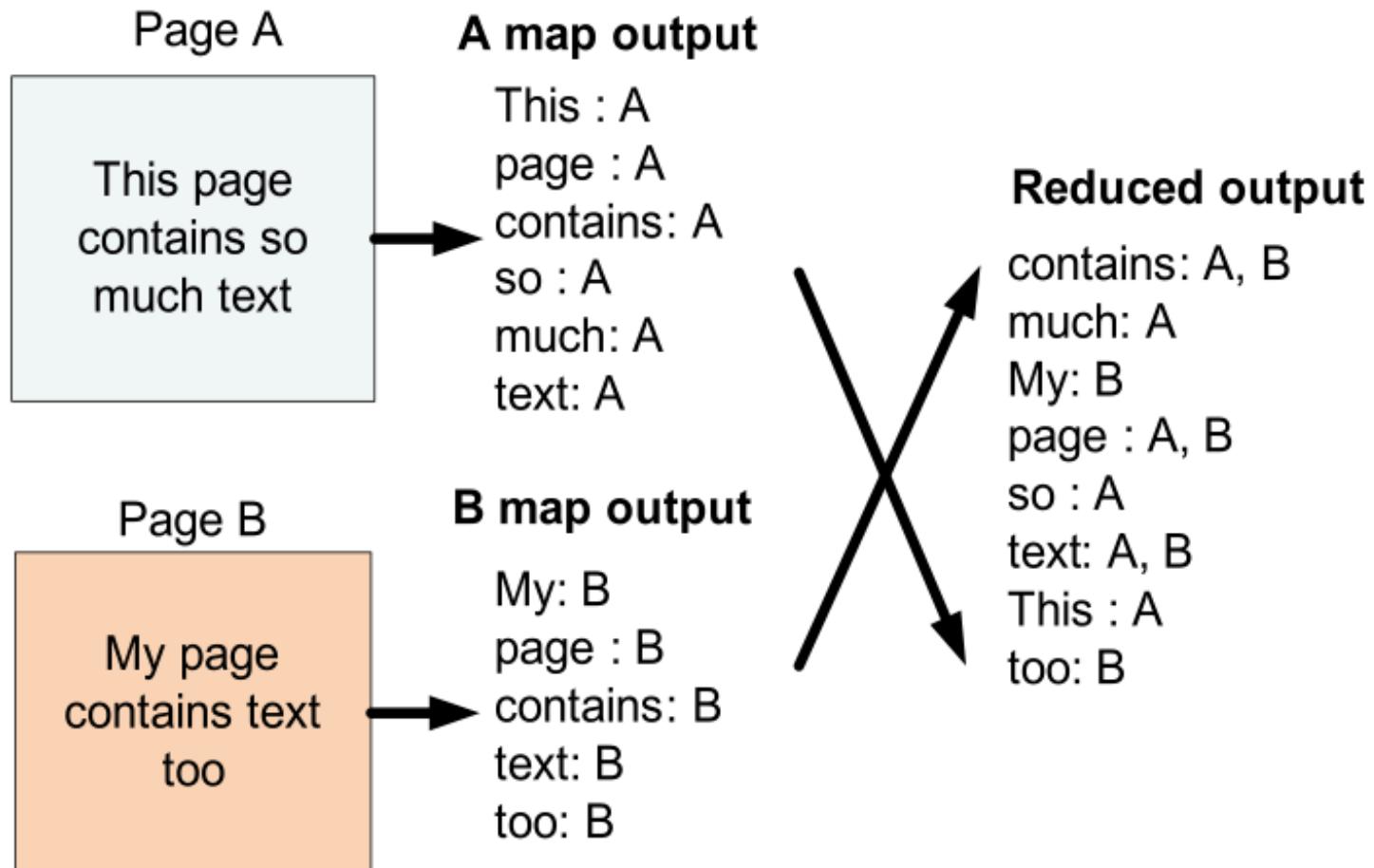
- **Mapper:**

- For each word in the line, emit (word, filename)

- **Reducer:**

- Identity function
 - Collect together all values for a given key (i.e., all filenames for a particular word)
 - Emit (word, filename_list)

Inverted Index: Dataflow



Aside: Word Count

- Recall the WordCount example we used earlier in the course
 - For each word, Mapper emitted (word, 1)
 - Very similar to the inverted index
- This is a common theme: reuse of existing Mappers, with minor modifications

Common MapReduce Algorithms

Introduction

Sorting and Searching

Indexing

 Machine Learning

TF-IDF

Word Co-Occurrence

Hands-On Exercise

Conclusion

Machine Learning: Introduction

- Machine Learning is a complex discipline
- Much research is ongoing
- Here we merely give a very high-level overview of some aspects of ML

What Is Machine Learning Not?

- Most programs tell computers exactly what to do
 - Database transactions and queries
 - Controllers
 - Phone systems, manufacturing processes, transport, weaponry, etc.
 - Media delivery
 - Simple search
 - Social systems
 - Chat, blogs, e-mail etc.

What Is Machine Learning?

- An alternative technique is to have computers *learn* what to do
- Machine Learning refers to a few classes of program that leverage collected data to drive future program behavior
- This represents another major opportunity to gain value from data

Why Use Hadoop for Machine Learning?

- Machine Learning systems are sensitive to the skill you bring to them
- However, practitioners often agree [Banko and Brill, 2001]:

**“It’s not who has the best algorithms that wins.
It’s who has the most data.”**

or...

“There’s no data like more data.”

The ‘Three Cs’

- Machine Learning is an active area of research and new applications
- There are three well-established categories of techniques for exploiting data:
 - Collaborative filtering (recommendations)
 - Clustering
 - Classification

Collaborative Filtering

- Collaborative Filtering is a technique for recommendations
- Example application: given people who each like certain books, learn to suggest what someone may like based on what they already like
- Very useful in helping users navigate data by expanding to topics that have affinity with their established interests
- Collaborative Filtering algorithms are agnostic to the different types of data items involved
 - So they are equally useful in many different domains

Clustering

- **Clustering algorithms discover structure in collections of data**
 - Where no formal structure previously existed
- **They discover what clusters, or ‘groupings’, naturally occur in data**
- **Examples:**
 - Finding related news articles
 - Computer vision (groups of pixels that cohere into objects)

Classification

- The previous two techniques are considered ‘unsupervised’ learning
 - The algorithm discovers groups or recommendations itself
- Classification is a form of ‘supervised’ learning
- A classification system takes a set of data records with known labels
 - Learns how to label new records based on that information
- Example:
 - Given a set of e-mails identified as spam/not spam, label new e-mails as spam/not spam
 - Given tumors identified as benign or malignant, classify new tumors

Mahout: A Machine Learning Library

- **Mahout is a Machine Learning library**
 - Included in CDH3
 - Contains algorithms for each of the categories listed
- **Algorithms included in Mahout:**

Recommendation	Clustering	Classification
Pearson correlation Log likelihood Spearman correlation Tanimoto coefficient Singular value decomposition (SVD) Linear interpolation Cluster-based recommenders	k-means clustering Canopy clustering Fuzzy k-means Latent Dirichlet analysis (LDA)	Stochastic gradient descent (SGD) Support vector machine (SVM) Naïve Bayes Complementary naïve Bayes Random forests

Common MapReduce Algorithms

Introduction

Sorting and Searching

Indexing

Machine Learning



TF-IDF

Word Co-Occurrence

Hands-On Exercise

Conclusion

Term Frequency – Inverse Document Frequency

- **Term Frequency – Inverse Document Frequency (TF-IDF)**
 - Answers the question “How important is this term in a document”
- **Known as a *term weighting function***
 - Assigns a score (weight) to each term (word) in a document
- **Very commonly used in text processing and search**
- **Has many applications in data mining**

TF-IDF: Motivation

- **Merely counting the number of occurrences of a word in a document is not a good enough measure of its relevance**
 - If the word appears in many other documents, it is probably less relevance
 - Some words appear too frequently in all documents to be relevant
 - Known as ‘stopwords’
- **TF-IDF considers both the frequency of a word in a given document and the number of documents which contain the word**

TF-IDF: Data Mining Example

- Consider a music recommendation system
 - Given many users' music libraries, provide "you may also like" suggestions
- If user A and user B have similar libraries, user A may like an artist in user B's library
 - But some artists will appear in almost everyone's library, and should therefore be ignored when making recommendations
 - E.g., Almost everyone has The Beatles in their record collection!

TF-IDF Formally Defined

- **Term Frequency (TF)**

- Number of times a term appears in a document (i.e., the count)

- **Inverse Document Frequency (IDF)**

$$idf = \log\left(\frac{N}{n}\right)$$

- N: total number of documents
 - n: number of documents that contain a term

- **TF-IDF**

- $TF \times IDF$

Computing TF-IDF

- **What we need:**

- Number of times t appears in a document
 - Different value for each document
 - Number of documents that contains t
 - One value for each term
 - Total number of documents
 - One value

Computing TF-IDF With MapReduce

- **Overview of algorithm: 3 MapReduce jobs**
 - Job 1: compute term frequencies
 - Job 2: compute number of documents each word occurs in
 - Job 3: compute TF-IDF
- **Notation in following slides:**
 - tf = term frequency
 - n = number of documents a term appears in
 - N = total number of documents
 - $docid$ = a unique id for each document

Computing TF-IDF: Job 1 – Compute tf

- **Mapper**

- Input: (docid, contents)
 - For each term in the document, generate a (term, docid) pair
 - i.e., we have seen this term in this document once
 - Output: ((term, docid), 1)

- **Reducer**

- Sums counts for word in document
 - Outputs ((term, docid), tf)
 - I.e., the term frequency of term in docid is tf

- **We can add a Combiner, which will use the same code as the Reducer**

Computing TF-IDF: Job 2 – Compute n

- **Mapper**

- Input: $((\text{term}, \text{docid}), tf)$
 - Output: $(\text{term}, (\text{docid}, tf, 1))$

- **Reducer**

- Sums 1s to compute n (number of documents containing term)
 - Note: need to buffer (docid, tf) pairs while we are doing this (more later)
 - Outputs $((\text{term}, \text{docid}), (tf, n))$

Computing TF-IDF: Job 3 – Compute TF-IDF

- **Mapper**

- Input: ((term, docid), (tf , n))
 - Assume N is known (easy to find)
 - Output ((term, docid), $TF \times IDF$)

- **Reducer**

- The identity function

Computing TF-IDF: Working At Scale

- **Job 2: We need to buffer (docid , tf) pairs counts while summing 1's (to compute n)**
 - Possible problem: pairs may not fit in memory!
 - How many documents does the word “the” occur in?
- **Possible solutions**
 - Ignore very-high-frequency words
 - Write out intermediate data to a file
 - Use another MapReduce pass

TF-IDF: Final Thoughts

- **Several small jobs add up to full algorithm**
 - Thinking in MapReduce often means decomposing a complex algorithm into a sequence of smaller jobs
- **Beware of memory usage for large amounts of data!**
 - Any time when you need to buffer data, there's a potential scalability bottleneck

Common MapReduce Algorithms

Introduction

Sorting and Searching

Indexing

Machine Learning

TF-IDF

 Word Co-Occurrence

Hands-On Exercise

Conclusion

Word Co-Occurrence: Motivation

- **Word Co-Occurrence measures the frequency with which two words appear close to each other in a corpus of documents**
 - For some definition of ‘close’
- **This is at the heart of many data-mining techniques**
 - Provides results for “people who did this, also do that”
 - Examples:
 - Shopping recommendations
 - Credit risk analysis
 - Identifying ‘people of interest’

Word Co-Occurrence: Algorithm

- Mapper

```
map(docid a, doc d) {  
    foreach w in d do  
        foreach u near w do  
            emit(pair(w, u), 1)  
}
```

- Reducer

```
reduce(pair p, Iterator counts) {  
    s = 0  
    foreach c in counts do  
        s += c  
    emit(p, s)  
}
```

Common MapReduce Algorithms

Introduction

Sorting and Searching

Indexing

Machine Learning

TF-IDF

Word Co-Occurrence

 **Hands-On Exercise**

Conclusion

Hands-On Exercise: Inverted Index

- In this Hands-On Exercise, you will write a MapReduce program to generate an inverted index of a set of documents
- Please refer to the Hands-On Exercise Instructions

Common MapReduce Algorithms

Introduction

Sorting and Searching

Indexing

Machine Learning

TF-IDF

Word Co-Occurrence

Hands-On Exercise



Conclusion

Conclusion

In this chapter you have learned

- **Some typical MapReduce algorithms, including**
 - Sorting
 - Searching
 - Indexing
 - Term Frequency – Inverse Document Frequency
 - Word Co-Occurrence
- **We also briefly discussed machine learning with Hadoop**



Chapter 8

An Introduction to Hive and Pig

Course Chapters

- Introduction
 - The Motivation For Hadoop
 - Hadoop: Basic Contents
 - Writing a MapReduce Program
 - Integrating Hadoop Into The Workflow
 - Delving Deeper Into The Hadoop API
 - Common MapReduce Algorithms
- An Introduction to Hive and Pig**
- Practical Development Tips and Techniques
 - More Advanced MapReduce Programming
 - Joining Data Sets in MapReduce Jobs
 - Graph Manipulation In MapReduce
 - An Introduction to Oozie
 - Conclusion
- Appendix: Cloudera Enterprise**

An Introduction to Hive and Pig

In this chapter you will learn

- **What features Hive provides**
- **What features Pig provides**

An Introduction to Hive and Pig



Motivation for Hive and Pig

Hive Basics

Pig Basics

Which To Use?

Hands-On Exercise

Conclusion

Hive and Pig: Motivation

- **MapReduce code is typically written in Java**
 - Although it can be written in other languages using Hadoop Streaming
- **Requires:**
 - A programmer
 - Who is a *good* Java programmer
 - Who understands how to think in terms of MapReduce
 - Who understands the problem they're trying to solve
 - Who has enough time to write and test the code
 - Who will be available to maintain and update the code in the future as requirements change

Hive and Pig: Motivation (cont'd)

- Many organizations have only a few developers who can write good MapReduce code
- Meanwhile, many other people want to analyze data
 - Business analysts
 - Data scientists
 - Statisticians
 - Data analysts
 - Etc
- What's needed is a higher-level abstraction on top of MapReduce
 - Providing the ability to query the data without needing to know MapReduce intimately
 - Hive and Pig address these needs

An Introduction to Hive and Pig

Motivation for Hive and Pig



Hive Basics

Pig Basics

Which To Use?

Hands-On Exercise

Conclusion

Hive: Introduction

- **Hive was originally developed at Facebook**
 - Provides a very SQL-like language
 - Can be used by people who know SQL
 - Under the covers, generates MapReduce jobs that run on the Hadoop cluster
 - Enabling Hive requires almost no extra work by the system administrator

The Hive Data Model

- Hive ‘layers’ table definitions on top of data in HDFS
- Tables
 - Typed columns (int, float, string, boolean etc)
 - Also, list: map (for JSON-like data)
- Partitions
 - e.g., to range-partition tables by date
- Buckets
 - Hash partitions within ranges (useful for sampling, join optimization)

Hive Datatypes

- **Primitive types:**

- TINYINT
- INT
- BIGINT
- BOOLEAN
- DOUBLE
- STRING

- **Type constructors:**

- ARRAY < *primitive-type* >
- MAP < *primitive-type*, *data-type* >
- STRUCT < *col-name* : *data-type*, ... >

The Hive Metastore

- **Hive's Metastore is a database containing table definitions and other metadata**
 - By default, stored locally on the client machine in a Derby database
 - If multiple people will be using Hive, the system administrator should create a shared Metastore
 - Usually in MySQL or some other relational database server

Hive Data: Physical Layout

- Hive tables are stored in Hive's 'warehouse' directory in HDFS
 - By default, /user/hive/warehouse
- Tables are stored in subdirectories of the warehouse directory
 - Partitions form subdirectories of tables
- Possible to create *external tables* if the data is already in HDFS and should not be moved from its current location
- Actual data is stored in flat files
 - Control character-delimited text, or SequenceFiles
 - Can be in arbitrary format with the use of a custom Serializer/Deserializer ('SerDe')

Starting The Hive Shell

- To launch the Hive shell, start a terminal and run

```
$ hive
```

- Results in the Hive prompt:

```
hive>
```

Hive Basics: Creating Tables

```
hive> SHOW TABLES;

hive> CREATE TABLE shakespeare
      (freq INT, word STRING)
      ROW FORMAT DELIMITED
      FIELDS TERMINATED BY '\t'
      STORED AS TEXTFILE;

hive> DESCRIBE shakespeare;
```

Loading Data Into Hive

- Data is loaded into Hive with the LOAD DATA INPATH statement
 - Assumes that the data is already in HDFS

```
LOAD DATA INPATH "shakespeare_freq" INTO TABLE shakespeare;
```

- If the data is on the local filesystem, use LOAD DATA LOCAL INPATH
 - Automatically loads it into HDFS

Basic SELECT Queries

- Hive supports most familiar SELECT syntax

```
hive> SELECT * FROM shakespeare LIMIT 10;  
  
hive> SELECT * FROM shakespeare  
      WHERE freq > 100 ORDER BY freq ASC  
      LIMIT 10;
```

Joining Tables

- Joining datasets is a complex operation in standard Java MapReduce
 - We will cover this later in the course
- In Hive, it's easy!

```
SELECT s.word, s.freq, k.freq FROM
    shakespeare s JOIN kjv k ON
    (s.word = k.word)
WHERE s.freq >= 5;
```

Storing Output Results

- The SELECT statement on the previous slide would write the data to the console
- To store the results in HDFS, create a new table then write, for example:

```
INSERT OVERWRITE TABLE newTable
    SELECT s.word, s.freq, k.freq FROM
        shakespeare s JOIN kjv k ON
        (s.word = k.word)
    WHERE s.freq >= 5;
```

- Results are stored in the table
- Results are just files within the *newTable* directory
 - Data can be used in subsequent queries, or in MapReduce jobs

Creating User-Defined Functions

- Hive supports manipulation of data via user-created functions
- Example:

```
INSERT OVERWRITE TABLE u_data_new
SELECT
    TRANSFORM (userid, movieid, rating, unixtime)
    USING 'python weekday_mapper.py'
    AS (userid, movieid, rating, weekday)
FROM u_data;
```

Hive Limitations

- **Not all ‘standard’ SQL is supported**
 - No correlated subqueries, for example
- **No support for UPDATE or DELETE**
- **No support for INSERTing single rows**
- **Relatively limited number of built-in functions**
- **No datatypes for date or time**
 - Use the STRING datatype instead

Hive: Where To Learn More

- Main Web site is at <http://hive.apache.org>
- Cloudera training course: Analyzing Data With Hive And Pig

An Introduction to Hive and Pig

Motivation for Hive and Pig

Hive Basics



Pig Basics

Which To Use?

Hands-On Exercise

Conclusion

Pig: Introduction

- **Pig was originally created at Yahoo! to answer a similar need to Hive**
 - Many developers did not have the Java and/or MapReduce knowledge required to write standard MapReduce programs
 - But still needed to query data
- **Pig is a dataflow language**
 - Language is called PigLatin
 - Relatively simple syntax
 - Under the covers, PigLatin scripts are turned into MapReduce jobs and executed on the cluster

Pig Installation

- **Installation of Pig requires no modification to the cluster**
- **The Pig interpreter runs on the client machine**
 - Turns PigLatin into standard Java MapReduce jobs, which are then submitted to the JobTracker
- **There is (currently) no shared metadata, so no need for a shared metastore of any kind**

Pig Concepts

- In Pig, a single element of data is an **atom**
- A collection of atoms – such as a row, or a partial row – is a **tuple**
- Tuples are collected together into **bags**
- Typically, a PigLatin script starts by loading one or more datasets into bags, and then creates new bags by modifying those it already has

Pig Features

- **Pig supports many features which allow developers to perform sophisticated data analysis without having to write Java MapReduce code**
 - Joining datasets
 - Grouping data
 - Referring to elements by position rather than name
 - Useful for datasets with many elements
 - Loading non-delimited data using a custom SerDe
 - Creation of user-defined functions, written in Java
 - And more

A Sample Pig Script

```
emps = LOAD 'people.txt' AS (id, name, salary);  
rich = FILTER emps BY salary > 100000;  
srted = ORDER rich BY salary DESC;  
STORE srted INTO 'rich_people';
```

- Here, we load a file into a bag called `emps`
- Then we create a new bag called `rich` which contains just those records where the salary portion is greater than 100000
- Finally, we write the contents of the `srted` bag to a new directory in HDFS
 - By default, the data will be written in tab-separated format
- Alternatively, to write the contents of a bag to the screen, say

```
DUMP srted;
```

More PigLatin

- To view the structure of a bag:

```
DESCRIBE bagnname;
```

- Joining two datasets:

```
data1 = LOAD 'data1' AS (col1, col2, col3, col4);
data2 = LOAD 'data2' AS (colA, colB, colC);
jnd = JOIN data1 BY col3, data2 BY colA;
STORE jnd INTO 'outfile';
```

More PigLatin: Grouping

- **Grouping:**

```
grpds = GROUP bag1 BY elementX
```

- **Creates a new bag**

- Each tuple in grpds has an element called group, and an element called bag1
- The group element has a unique value for elementX from bag1
- The bag1 element is itself a bag, containing all the tuples from bag1 with that value for elementX

More PigLatin: FOREACH

- The FOREACH . . . GENERATE statement iterates over members of a bag
- Example:

```
justnames = FOREACH emps GENERATE name;
```

- Can combine with COUNT:

```
summedUp = FOREACH grpds GENERATE group,
           COUNT(bag1) AS elementCount;
```

Pig: Where To Learn More

- Main Web site is at <http://pig.apache.org>
 - Follow the links on the left-hand side of the page to Documentation, then Release 0.7.0, then Pig Latin 1 and Pig Latin 2
- Cloudera training course: Analyzing Data With Hive And Pig

An Introduction to Hive and Pig

Motivation for Hive and Pig

Hive Basics

Pig Basics

 **Which To Use?**

Hands-On Exercise

Conclusion

Choosing Between Pig and Hive

- Typically, organizations wanting an abstraction on top of standard MapReduce will choose to use either Hive or Pig
- Which one is chosen depends on the skillset of the target users
 - Those with an SQL background will naturally gravitate towards Hive
 - Those who do not know SQL will often choose Pig
- Each has strengths and weaknesses; it is worth spending some time investigating each so you can make an informed decision
- Some organizations are now choosing to use both
 - Pig deals better with less-structured data, so Pig is used to manipulate the data into a more structured form, then Hive is used to query that structured data

An Introduction to Hive and Pig

Motivation for Hive and Pig

Hive Basics

Pig Basics

Which To Use?

 **Hands-On Exercise**

Conclusion

Hands-On Exercise: Manipulating Data with Hive or Pig

- In this Hands-On Exercise, you will manipulate a dataset using Hive
- Please refer to the Hands-On Exercise Manual

An Introduction to Hive and Pig

Motivation for Hive and Pig

Hive Basics

Pig Basics

Which To Use?

Hands-On Exercise



Conclusion

Conclusion

In this chapter you have learned

- **What features Hive provides**
- **What features Pig provides**



Chapter 9

Practical Development Tips and Techniques

Course Chapters

- Introduction
- The Motivation For Hadoop
- Hadoop: Basic Contents
- Writing a MapReduce Program
- Integrating Hadoop Into The Workflow
- Delving Deeper Into The Hadoop API
- Common MapReduce Algorithms
- An Introduction to Hive and Pig
- Practical Development Tips and Techniques**
- More Advanced MapReduce Programming
- Joining Data Sets in MapReduce Jobs
- Graph Manipulation In MapReduce
- An Introduction to Oozie
- Conclusion
- Appendix: Cloudera Enterprise

Practical Development Tips and Techniques

In this chapter you will learn

- Strategies for debugging your MapReduce code
- How to write and subsequently view log files
- How to retrieve job information with Counters
- What issues to consider when using file compression
- How to determine the optimal number of Reducers for a job
- How to create Map-only MapReduce jobs

Practical Development Tips and Techniques



- Strategies for Debugging MapReduce Code**
- Using LocalJobRunner for Easier Debugging**
- Logging**
- Retrieving Job Information with Counters**
- Splittable File Formats**
- Determining the Optimal Number of Reducers**
- Map-Only MapReduce Jobs**
- Hands-On Exercise**
- Conclusion**

Introduction to Debugging

- **Debugging MapReduce code is difficult!**
 - Each instance of a Mapper runs as a separate task
 - Often on a different machine
 - Difficult to attach a debugger to the process
 - Difficult to catch ‘edge cases’
- **Very large volumes of data mean that unexpected input is likely to appear**
 - Code which expects all data to be well-formed is likely to fail

Common-Sense Debugging Tips

- **Code defensively**
 - Ensure that input data is in the expected format
 - Expect things to go wrong
 - Catch exceptions
- **Start small, build incrementally**
- **Make as much of your code as possible Hadoop-agnostic**
 - Makes it easier to test
- **Write Unit Tests**
- **Test locally whenever possible**
 - With small amounts of data
- **Then test in pseudo-distributed mode**
- **Finally, test on the cluster**

Basic Debugging Strategies

- You can throw exceptions if a particular condition is met
 - E.g., if illegal data is found

```
throw new RuntimeException("Your message here");
```

- This causes the task to fail
- If a task fails four times, the entire job will fail

Basic Debugging Strategies (cont'd)

- **If you suspect the input data of being faulty, you may be tempted to log the (key, value) pairs your Mapper receives**
 - Reasonable for small amounts of input data
 - Caution! If your job runs across 500GB of input data, you will be writing 500GB of log files!
 - Remember to think at scale...
- **Instead, wrap vulnerable sections of code in `try { . . . }` blocks**
 - Write logs in the `catch { . . . }` block
 - This way only critical data is logged

Testing Strategies

- When testing in pseudo-distributed mode, ensure that you are testing with a similar environment to that on the real cluster
 - Same amount of RAM allocated to the task JVMs
 - Same version of Hadoop
 - Same version of Java
 - Same versions of third-party libraries

Practical Development Tips and Techniques

Strategies for Debugging MapReduce Code

 **Using LocalJobRunner for Easier Debugging**

Logging

Retrieving Job Information with Counters

Splittable File Formats

Determining the Optimal Number of Reducers

Map-Only MapReduce Jobs

Hands-On Exercise

Conclusion

Testing Locally

- **Hadoop can run MapReduce in a single, local process**
 - Does not require any Hadoop daemons to be running
 - Uses the local filesystem
 - Known as the LocalJobRunner
- **This is a very useful way of quickly testing incremental changes to code**
- **To run in LocalJobRunner mode, add the following lines to your driver code:**

```
conf.set("mapred.job.tracker", "local");
conf.set("fs.default.name", "file:///");
```

- Or set these options on the command line with the `-D` flag

Testing Locally (cont'd)

- Some limitations of LocalJobRunner mode:

- DistributedCache does not work
 - The job can only specify a single Reducer
 - Some 'beginner' mistakes may not be caught
 - For example, attempting to share data between Mappers will work, because the code is running in a single JVM

LocalJobRunner Mode in Eclipse

- Eclipse can be configured to run Hadoop code in LocalJobRunner mode
 - From within the IDE
- This allows very rapid development iterations
 - ‘Agile programming’

Practical Development Tips and Techniques

Strategies for Debugging MapReduce Code

Using LocalJobRunner for Easier Debugging

 **Logging**

Retrieving Job Information with Counters

Splittable File Formats

Determining the Optimal Number of Reducers

Map-Only MapReduce Jobs

Hands-On Exercise

Conclusion

Before Logging: `stdout` and `stderr`

- Tried-and-true debugging technique: write to `stdout` or `stderr`
- If running in LocalJobRunner mode, you will see the results of `System.err.println()`
- If running on a cluster, that output will not appear on your console
 - Output is visible via Hadoop's Web UI

Aside: The Hadoop Web UI

- **All Hadoop daemons contain a Web server**
 - Exposes information on a well-known port
- **Most important for developers is the JobTracker Web UI**
 - `http://<job_tracker_address>:50030`
 - `http://localhost:50030` if running in pseudo-distributed mode
- **Also useful: the NameNode Web UI**
 - `http://<name_node_address>:50070`

Aside: The Hadoop Web UI (cont'd)

- Your instructor will now demonstrate the JobTracker UI

The screenshot shows a Mozilla Firefox browser window displaying the 'Hadoop Task Details' page. The URL in the address bar is http://node1:50030/taskdetails.jsp?jobid=job_201009070723_0017&tipid=task_2. The page title is 'Hadoop Task Details - Mozilla Firefox'. The main content area shows a table titled 'All Task Attempts' for job 'job_201009070723_0017'. The table has columns: Task Attempts, Machine, Status, Progress, Start Time, Finish Time, Errors, Task Logs, Counters, and Actions. One row is visible: 'attempt_201009070723_0017_m_000000_0' on machine '/default-rack/node4' with status 'SUCCEEDED', progress '100.00%', start time '9-Oct-2010 17:38:40', finish time '9-Oct-2010 17:39:02 (21sec)', and errors '0'. The 'Task Logs' column for this row is circled in red. It contains links: 'Last 4KB', 'Last 8KB', and 'All'. Below the table, there's a section for 'Input Split Locations' with links to go back to the job or JobTracker, and a note about Cloudera's Distribution for Hadoop, 2010.

Task Attempts	Machine	Status	Progress	Start Time	Finish Time	Errors	Task Logs	Counters	Actions
attempt_201009070723_0017_m_000000_0	/default-rack/node4	SUCCEEDED	100.00%	9-Oct-2010 17:38:40	9-Oct-2010 17:39:02 (21sec)	0	Last 4KB Last 8KB All	8	

Logging: Better Than Printing

- **println statements rapidly become awkward**
 - Turning them on and off in your code is tedious, and leads to errors
- **Logging provides much finer-grained control over:**
 - What gets logged
 - When something gets logged
 - How something is logged

Logging With log4j

- Hadoop uses log4j to generate all its log files
- Your Mappers and Reducers can also use log4j
 - All the initialization is handled for you by Hadoop

```
import org.apache.commons.logging.Log;
import org.apache.commons.logging.LogFactory;

class FooMapper implements Mapper {
    public static final Log LOGGER =
        LogFactory.getLog(FooMapper.class.getName());
    ...
}
```

Logging With log4j (cont'd)

- Simply send strings to loggers tagged with severity levels:

```
LOGGER.debug("message") ;  
LOGGER.info("message") ;  
LOGGER.warn("message") ;  
LOGGER.error("message") ;
```

log4j Configuration

- Configuration for log4j is stored in
`/etc/hadoop/conf/log4j.properties`
- Can change global log settings with `hadoop.root.log` property
- Can override log level on a per-class basis:

```
log4j.logger.org.apache.hadoop.mapred.JobTracker=WARN
```

```
log4j.logger.org.apache.hadoop.mapred.FooMapper=DEBUG
```

- Programmatically:

```
LOGGER.setLevel(Level.WARN);
```

Dynamically Setting Log Levels

- Although log levels can be set in `log4j.properties`, this would require modification of files on all slave nodes
 - In practice, this is unrealistic
- Instead, a good solution is to set the log level in your code based on a command-line parameter

Dynamically Setting Log Levels (cont'd)

- In the code for your Mapper or Reducer:

```
public void configure(JobConf conf) {  
    if  
    ("DEBUG".equals(conf.get("com.cloudera.job.logging")))  
    {  
        LOGGER.setLevel(Level.DEBUG);  
    }  
    LOGGER.debug("**** configure(): Log Level set to  
DEBUG");  
}
```

- Then on the command line, specify the log level with e.g.:

```
$ hadoop jar wc.jar WordCountWTool \  
-D com.cloudera.job.logging=DEBUG myclass indir outdir
```

Where Are Log Files Stored?

- Log files are stored by default at
`/var/log/hadoop/userlogs/${task.id}/syslog`
on the machine where the task attempt ran
 - Configurable
- Tedium to have to ssh in to a node to view its logs
 - Much easier to use the JobTracker Web UI
 - Automatically retrieves and displays the log files for you

Practical Development Tips and Techniques

Strategies for Debugging MapReduce Code

Using LocalJobRunner for Easier Debugging

Logging



Retrieving Job Information with Counters

Splittable File Formats

Determining the Optimal Number of Reducers

Map-Only MapReduce Jobs

Hands-On Exercise

Conclusion

What Are Counters?

- **Counters provide a way for Mappers or Reducers to pass aggregate values back to the driver after the job has completed**
 - Their values are also visible from the JobTracker's Web UI
 - And are reported on the console when the job ends
- **Very basic: just have a name and a value**
 - Value can be incremented within the code
- **Counters are collected into Groups**
 - Within the group, each Counter has a name
- **Example: A group of Counters called RecordType**
 - Names: TypeA, TypeB, TypeC
 - Appropriate Counter will be incremented as each record is read in the Mapper

What Are Counters?

- **Counters provide a way for Mappers or Reducers to pass aggregate values back to the driver after the job has completed**
 - Their values are also visible from the JobTracker's Web UI
 - And are reported on the console when the job ends
- **Counters can be set and incremented via the method**

```
Reporter.incrCounter(group, name, amount);
```

- **Example:**

```
r.incrCounter("RecordType", "TypeA", 1);
```

Retrieving Counters in the Driver Code

- To retrieve Counters in the Driver code after the job is complete, use code like this:

```
RunningJob job = JobClient.runJob(conf);
long typeARecords =
    job.getCounters().findCounter("RecordType", "TypeA").getValue();

long typeBRecords =
    job.getCounters().findCounter("RecordType", "TypeB").getValue();
```

Counters: Caution

- **Do not rely on a counter's value from the Web UI while a job is running**
 - Due to possible speculative execution, a counter's value could appear larger than the actual final value
 - Modifications to counters from subsequently killed/failed tasks will be removed from the final count

Practical Development Tips and Techniques

Strategies for Debugging MapReduce Code

Using LocalJobRunner for Easier Debugging

Logging

Retrieving Job Information with Counters



Splittable File Formats

Determining the Optimal Number of Reducers

Map-Only MapReduce Jobs

Hands-On Exercise

Conclusion

InputSplits: Reprise

- Recall that each Mapper processes a single *InputSplit*
- InputSplits are calculated by the client program before the job is submitted to the JobTracker
- Typically, an InputSplit equates to an HDFS block
 - So the number of Mappers will equal the number of HDFS blocks in the input data

Hadoop and Compressed Files

- Hadoop understands a variety of file compression formats
 - Including GZip
- If a compressed file is included as one of the files to be processed, Hadoop will automatically decompress it and pass the decompressed contents to the Mapper
 - There is no need for the developer to worry about decompressing the file
- However, GZip is not a ‘splittable file format’
 - A GZipped file can only be decompressed by starting at the beginning of the file and continuing on to the end
 - You cannot start decompressing the file part of the way through it

Non-Splittable File Formats and Hadoop

- If the MapReduce framework receives a non-splittable file (such as a GZipped file) it passes the *entire* file to a single Mapper
- This can result in one Mapper running for far longer than the others
 - It is dealing with an entire file, while the others are dealing with smaller portions of files
 - Speculative execution can occur
 - Although this will provide no benefit
- Typically it is not a good idea to use GZip to compress files which will be processed by MapReduce

Splittable Compression Formats: LZO

- One splittable compression format is LZO
- Because of licensing restrictions, LZO cannot be shipped with Hadoop
 - But it is easy to add
 - See <https://github.com/cloudera/hadoop-lzo>
- LZO files have set points within the file at which compression can begin
 - InputSplits are calculated based on those points
 - The point closest to each block boundary is chosen

Splittable Compression Formats: Snappy

- **Snappy is a relatively new compression format**
 - Developed at Google
 - Splittable
 - Very fast
- **Snappy does not compress a file and produce, e.g., a file with a .snappy extension**
 - Instead, it compresses data within a file
 - That data can be decompressed automatically by Hadoop (or other programs) when the file is read
 - Works well with SequenceFiles, Avro files
- **Snappy is still new to Hadoop, but is already seeing significant adoption**

Practical Development Tips and Techniques

Strategies for Debugging MapReduce Code

Using LocalJobRunner for Easier Debugging

Logging

Retrieving Job Information with Counters

Splittable File Formats

 **Determining the Optimal Number of Reducers**

Map-Only MapReduce Jobs

Hands-On Exercise

Conclusion

How Many Reducers Do You Need?

- An important consideration when creating your job is to determine the number of Reducers specified
- Default is a single Reducer
- With a single Reducer, one task receives *all* keys in sorted order
 - This is sometimes advantageous if the output must be in completely sorted order
 - Can cause significant problems if there is a large amount of intermediate data
 - Node on which the Reducer is running may not have enough disk space to hold all intermediate data
 - The Reducer will take a long time to run

Jobs Which Require a Single Reducer

- If a job needs to output a file where all keys are listed in sorted order, a single Reducer must be used
- Alternatively, the **TotalOrderPartitioner** can be used
 - Uses an externally generated file which contains information about intermediate key distribution
 - Partitions data such that all keys which go to the first reducer are smaller than any which go to the second, etc
 - In this way, multiple Reducers can be used
 - Concatenating the Reducers' output files results in a totally ordered list

Jobs Which Require a Fixed Number of Reducers

- Some jobs will require a specific number of Reducers
- Example: a job must output one file per day of the week
- Key will be the weekday
- Seven Reducers will be specified
- A Partitioner will be written which sends one key to each Reducer

Jobs With a Variable Number of Reducers

- Many jobs can be run with a variable number of Reducers
- Developer must decide how many to specify
 - Each Reducer should get a reasonable amount of intermediate data, but not too much
 - Chicken-and-egg problem
- Typical way to determine how many Reducers to specify:
 - Test the job with a relatively small test data set
 - Extrapolate to calculate the amount of intermediate data expected from the 'real' input data
 - Use that to calculate the number of Reducers which should be specified

Practical Development Tips and Techniques

Strategies for Debugging MapReduce Code

Using LocalJobRunner for Easier Debugging

Logging

Retrieving Job Information with Counters

Splittable File Formats

Determining the Optimal Number of Reducers

 **Map-Only MapReduce Jobs**

Hands-On Exercise

Conclusion

Map-Only MapReduce Jobs

- There are many types of job where only a Mapper is needed
- Examples:
 - Image processing
 - File format conversion
 - Input data sampling
 - ETL

Creating Map-Only Jobs

- To create a Map-only job, set the number of Reducers to 0 in your Driver code

```
conf.setNumReduceTasks(0);
```

- Anything written using the `OutputCollector.collect` method will be written to HDFS
 - Rather than written as intermediate data
 - One file per Mapper will be written

Practical Development Tips and Techniques

Strategies for Debugging MapReduce Code

Using LocalJobRunner for Easier Debugging

Logging

Retrieving Job Information with Counters

Splittable File Formats

Determining the Optimal Number of Reducers

Map-Only MapReduce Jobs

 **Hands-On Exercise**

Conclusion

Hands-On Exercise

- In this Hands-On Exercise you will write a Map-Only MapReduce job using Counters
- Please refer to the Hands-On Exercise Manual

Practical Development Tips and Techniques

Strategies for Debugging MapReduce Code

Using LocalJobRunner for Easier Debugging

Logging

Retrieving Job Information with Counters

Splittable File Formats

Determining the Optimal Number of Reducers

Map-Only MapReduce Jobs

Hands-On Exercise



Conclusion

Conclusion

In this chapter you have learned

- **Strategies for debugging your MapReduce code**
- **How to write and subsequently view log files**
- **How to retrieve job information with Counters**
- **What issues to consider when using file compression**
- **How to determine the optimal number of Reducers for a job**
- **How to create Map-only MapReduce jobs**



Chapter 10

More Advanced

MapReduce Programming

Course Chapters

- Introduction
- The Motivation For Hadoop
- Hadoop: Basic Contents
- Writing a MapReduce Program
- Integrating Hadoop Into The Workflow
- Delving Deeper Into The Hadoop API
- Common MapReduce Algorithms
- An Introduction to Hive and Pig
- Practical Development Tips and Techniques
- More Advanced MapReduce Programming**
- Joining Data Sets in MapReduce Jobs
- Graph Manipulation In MapReduce
- An Introduction to Oozie
- Conclusion
- Appendix: Cloudera Enterprise

More Advanced MapReduce Programming

In this chapter you will learn

- How to create custom **Writables** and **WritableComparables**
- How to save binary data using **SequenceFiles** and **Avro**
- How to build custom **InputFormats** and **OutputFormats**

More Advanced MapReduce Programming



A Recap of the MapReduce Flow

Custom Writables and WritableComparables

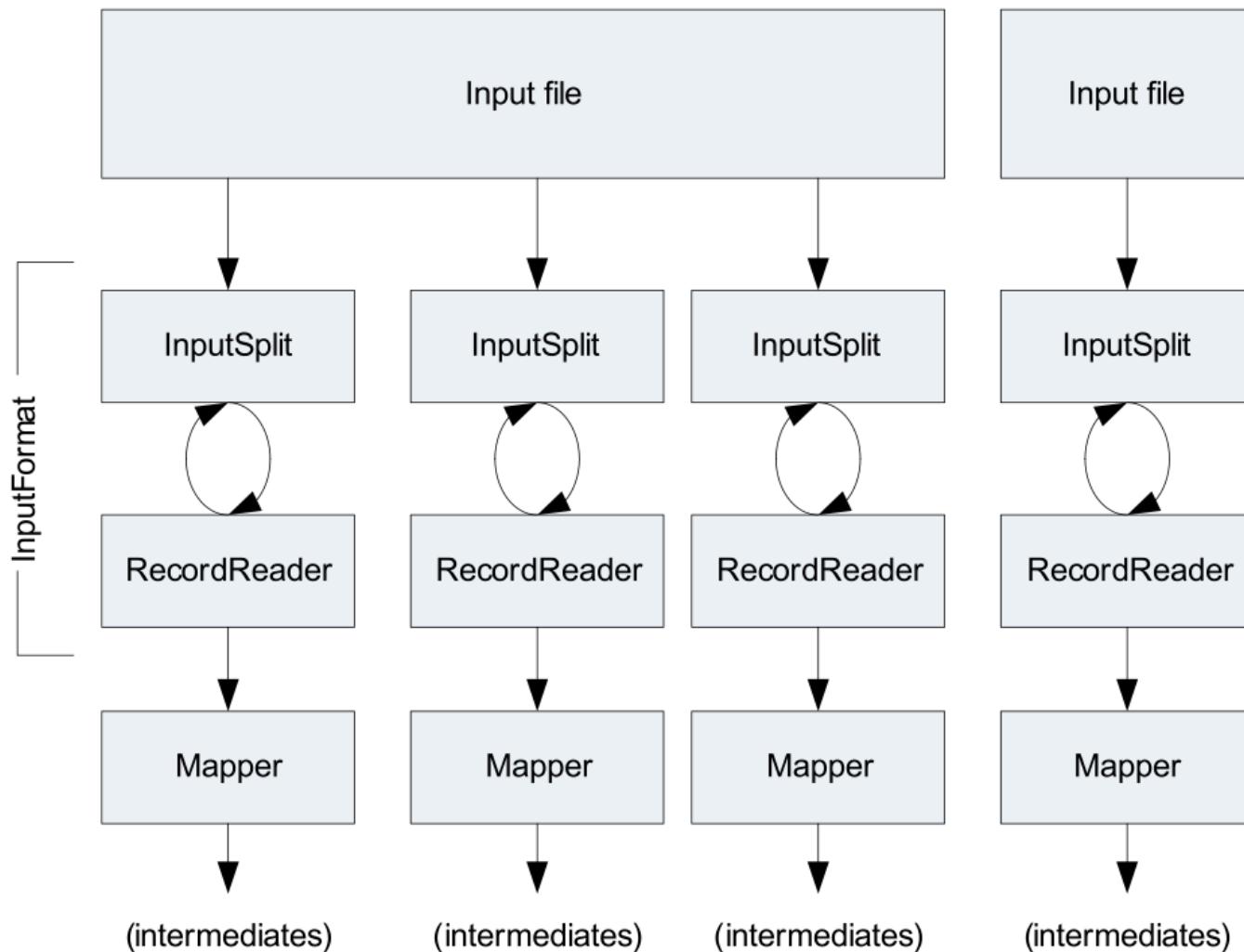
Saving Binary Data

Creating InputFormats and OutputFormats

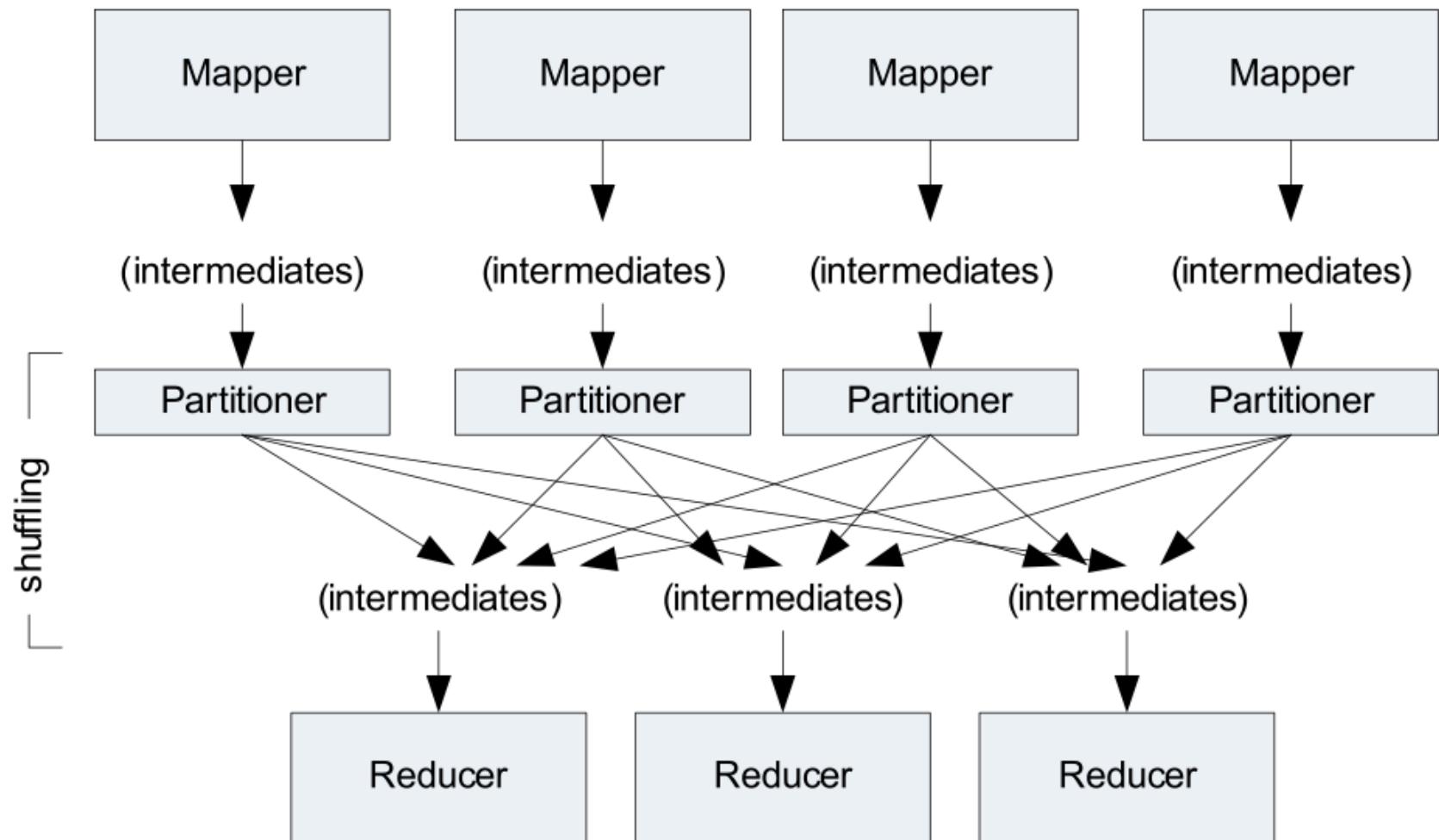
Hands-On Exercise

Conclusion

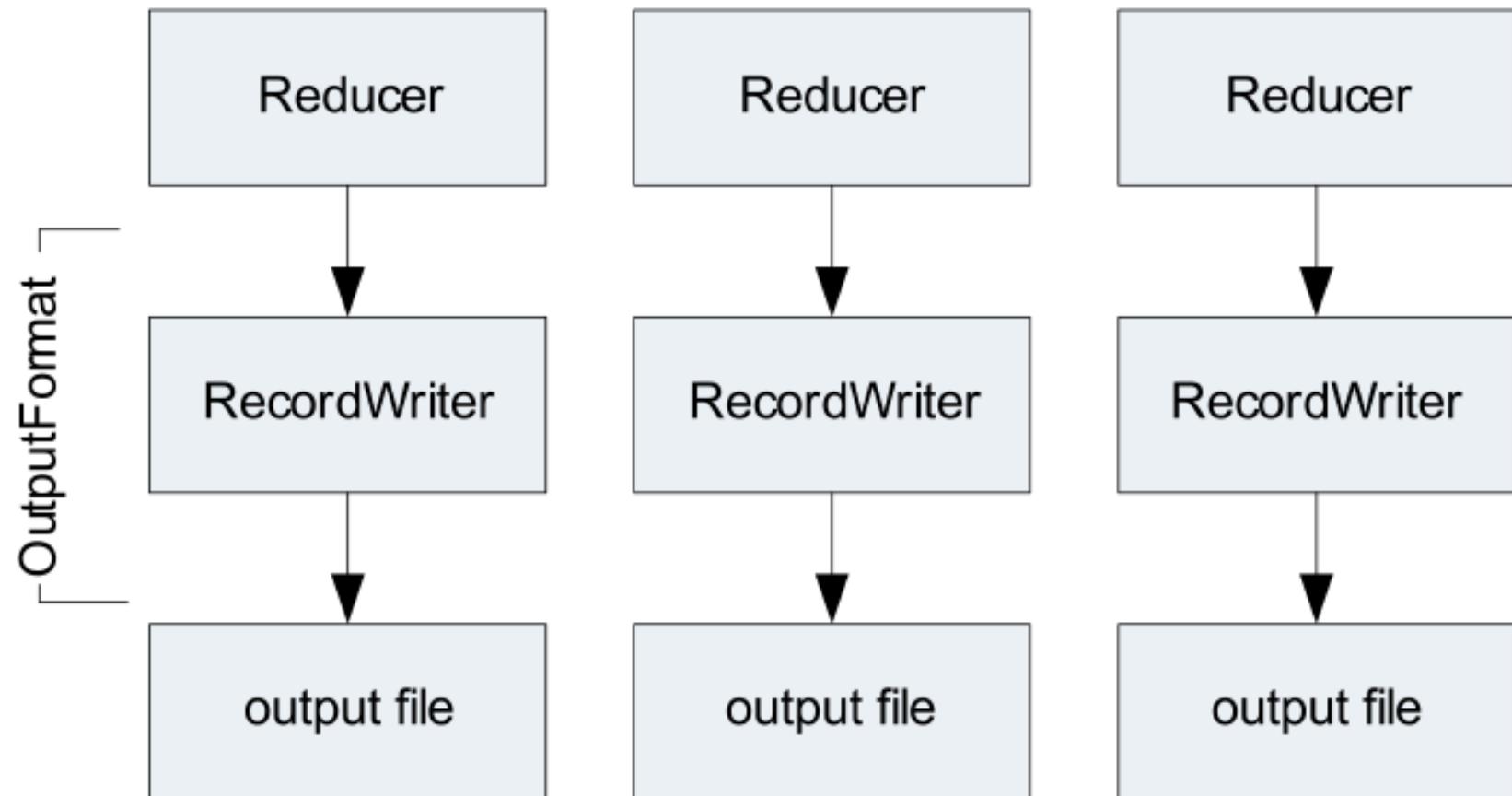
Recap: Inputs to Mappers



Recap: Sort and Shuffle



Recap: Reducers to Outputs



More Advanced MapReduce Programming

A Recap of the MapReduce Flow

 **Custom Writables and WritableComparables**

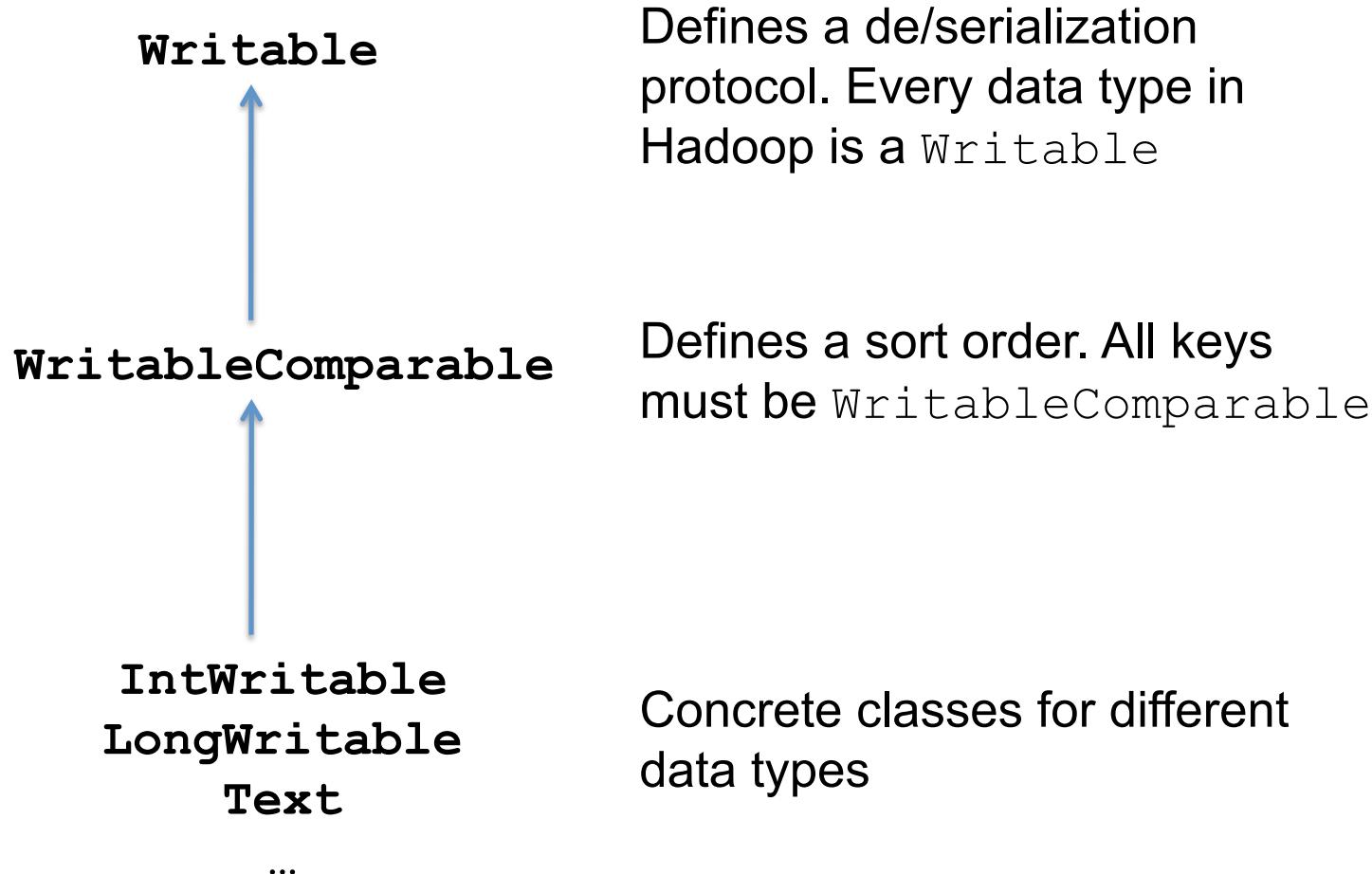
Saving Binary Data

Creating InputFormats and OutputFormats

Hands-On Exercise

Conclusion

Data Types in Hadoop



'Box' Classes in Hadoop

- Hadoop data types are 'box' classes
 - Text: string
 - IntWritable: int
 - LongWritable: long
 - FloatWritable: float
 - ...
- Writable defines wire transfer format

Creating a Complex Writable

- Example: say we want a tuple (a, b)
 - We could artificially construct it by, for example, saying

```
Text t = new Text(a + "," + b);  
...  
String[] arr = t.toString().split(",");
```

- Inelegant
- Problematic
 - If a or b contained commas
- Not always practical
 - Doesn't easily work for binary objects

The Writable Interface

```
public interface Writable {  
    void readFields(DataInput in);  
    void write(DataOutput out);  
}
```

- **DataInput/DataOutput supports**

- boolean
- byte, char (Unicode, 2 bytes)
- double, float, int, long, string
- Line until line terminator
- unsigned byte, short
- UTF string
- byte array

Example: 3D Point Class

```
class Point3d implements Writable {  
    float x, y, z;  
  
    void readFields(DataInput in) {  
        x = in.readFloat();  
        y = in.readFloat();  
        z = in.readFloat();  
    }  
  
    void write(DataOutput out) {  
        out.writeFloat(x);  
        out.writeFloat(y);  
        out.writeFloat(z);  
    }  
}
```

What About Binary Objects?

- **Solution: use byte arrays**
- **Write idiom:**
 - Serialize object to byte array
 - Write byte count
 - Write byte array
- **Read idiom:**
 - Read byte count
 - Create byte array of proper size
 - Read byte array
 - Deserialize object

WritableComparable

- **WritableComparable is a sub-interface of Writable**
 - Must implement compareTo, hashCode, equals methods
- **All keys in MapReduce must be WritableComparable**

Example: 3D Point Class

```
class Point3d implements WritableComparable {
    float x, y, z;

    void readFields(DataInput in) {
        x = in.readFloat();
        y = in.readFloat();
        z = in.readFloat();
    }
    void write(DataOutput out) {
        out.writeFloat(x);
        out.writeFloat(y);
        out.writeFloat(z);
    }

    public int compareTo(Point3d p) {
        // whatever ordering makes sense ... (e.g., closest to origin)
    }

    public boolean equals(Object o) {
        Point3d p = (Point3d) o;
        return this.x == p.x && this.y == p.y && this.z == p.z;
    }

    public int hashCode() {
        // whatever makes sense ...
    }
}
```

Comparators to Speed Up Sort and Shuffle

- Recall that after each Map task, Hadoop must sort the keys
 - Keys are passed to a Reducer in sorted order
- Naïve approach: use the `WritableComparable's compareTo` method for each key
 - This requires deserializing each key, which could be time consuming
- Better approach: use a Comparator
 - A class which can compare two `WritableComparables`, ideally just by looking at the two byte streams
 - Avoids the need for deserialization, if possible
 - Not always possible; depends on the actual key definition

Example Comparator for the Text Class

```
public class Text implements WritableComparable {
    ...
    public static class Comparator extends WritableComparator {
        public Comparator() {
            super(Text.class);
        }
        public int compare(byte[] b1, int s1, int l1, byte[] b2, int s2, int l2)
        {
            int n1 = WritableUtils.decodeVIntSize(b1[s1]);
            int n2 = WritableUtils.decodeVIntSize(b2[s2]);
            return compareBytes(b1, s1+n1, l1-n1, b2, s2+n2, l2-n2);
        }
    }
    ...
    static {
        // register this comparator
        WritableComparator.define(Text.class, new Comparator());
    }
}
```

Using Custom Types in MapReduce Jobs

- Use methods in JobConf to specify your custom key/value types
- For output of Mappers:

```
conf.setMapOutputKeyClass()  
conf.setMapOutputValueClass()
```

- For output of Reducers:

```
conf.setOutputKeyClass()  
conf.setOutputValueClass()
```

- Input types are defined by InputFormat
 - See later

More Advanced MapReduce Programming

A Recap of the MapReduce Flow

Custom Writables and WritableComparables

 **Saving Binary Data**

Creating InputFormats and OutputFormats

Hands-On Exercise

Conclusion

What Are SequenceFiles?

- **SequenceFiles are files containing binary-encoded key-value pairs**
 - Work naturally with Hadoop data types
 - SequenceFiles include metadata which identifies the data type of the key and value
- **Actually, three file types in one**
 - Uncompressed
 - Record-compressed
 - Block-compressed
- **Often used in MapReduce**
 - Especially when the output of one job will be used as the input for another
 - SequenceFileInputFormat
 - SequenceFileOutputFormat

Directly Accessing SequenceFiles

- It is possible to directly access SequenceFiles from your code:

```
Configuration config = new Configuration();
SequenceFile.Reader reader =
    new SequenceFile.Reader(FileSystem.get(config), path, config);

Text key = (Text) reader.getKeyClass().newInstance();
IntWritable value = (IntWritable) reader.getValueClass().newInstance();

while (reader.next(key, value)) {
    // do something here
}
reader.close();
```

Problems With SequenceFiles

- **SequenceFiles are very useful but have some potential problems**
- **They are only typically accessible via the Java API**
 - Some work has been done to allow access from other languages
- **If the definition of the key or value object changes, the file becomes unreadable**

An Alternative to SequenceFiles: Avro

- **Apache Avro is a serialization format which is becoming a popular alternative to SequenceFiles**
 - Project was created by Doug Cutting, the creator of Hadoop
- **Self-describing file format**
 - The schema for the data is included in the file itself
- **Compact file format**
- **Portable across multiple languages**
 - Support for C, C++, Java, Python, Ruby and others
- **Compatible with Hadoop**
 - Via the AvroMapper and AvroReducer classes

More Advanced MapReduce Programming

A Recap of the MapReduce Flow

Custom Writables and WritableComparables

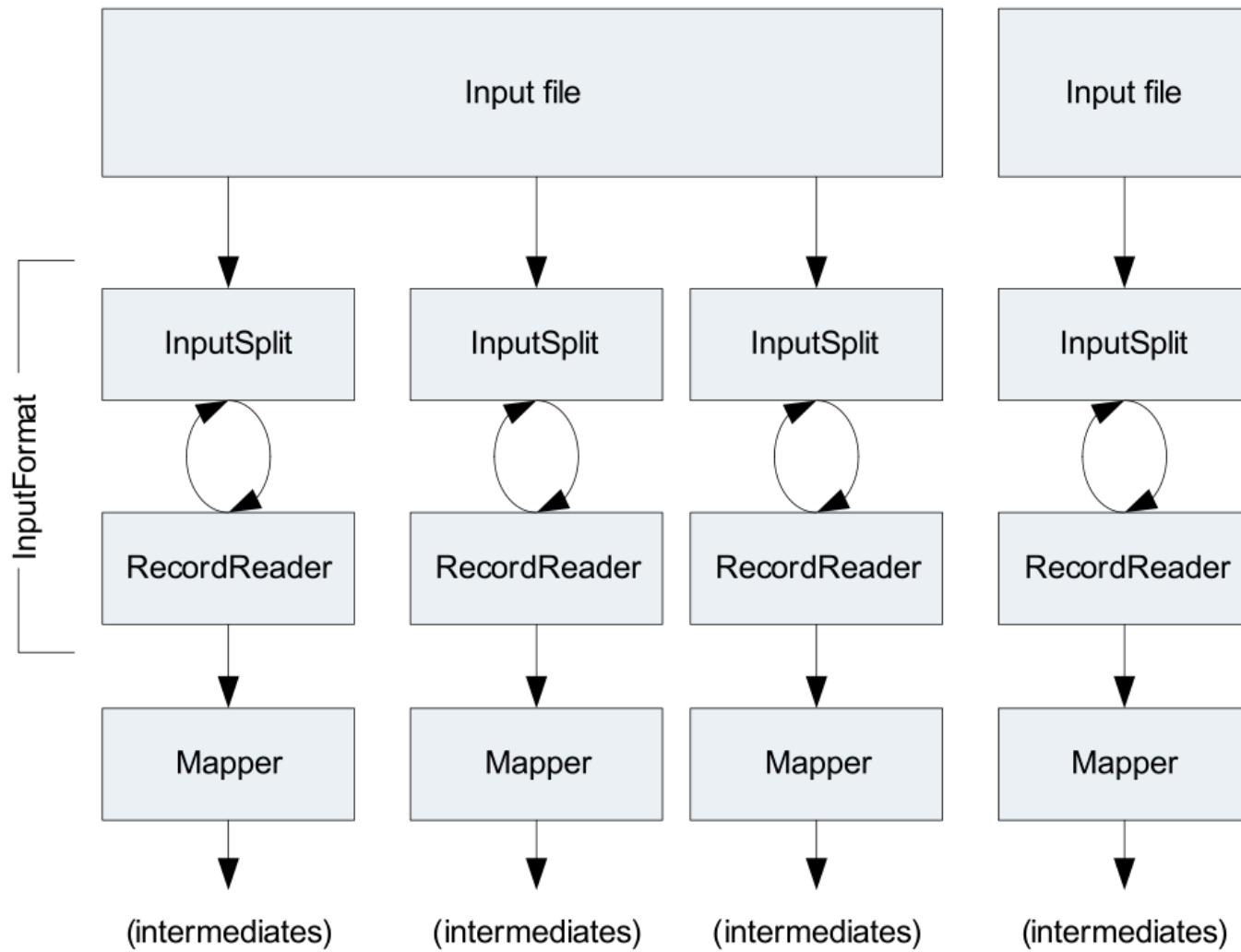
Saving Binary Data

 **Creating InputFormats and OutputFormats**

Hands-On Exercise

Conclusion

Reprise: The Role of the InputFormat



Most Common InputFormats

- **Most common InputFormats:**

- TextInputFormat
- KeyValueTextInputFormat
- SequenceFileInputFormat

- **Others are available**

- NLineInputFormat
 - Every n lines of an input file is treated as a separate InputSplit
 - Configure in the driver code with

```
mapred.line.input.format.linespermap
```
- MultiFileInputFormat
 - Abstract class which manages the use of multiple files in a single task
 - You must supply a getRecordReader() implementation

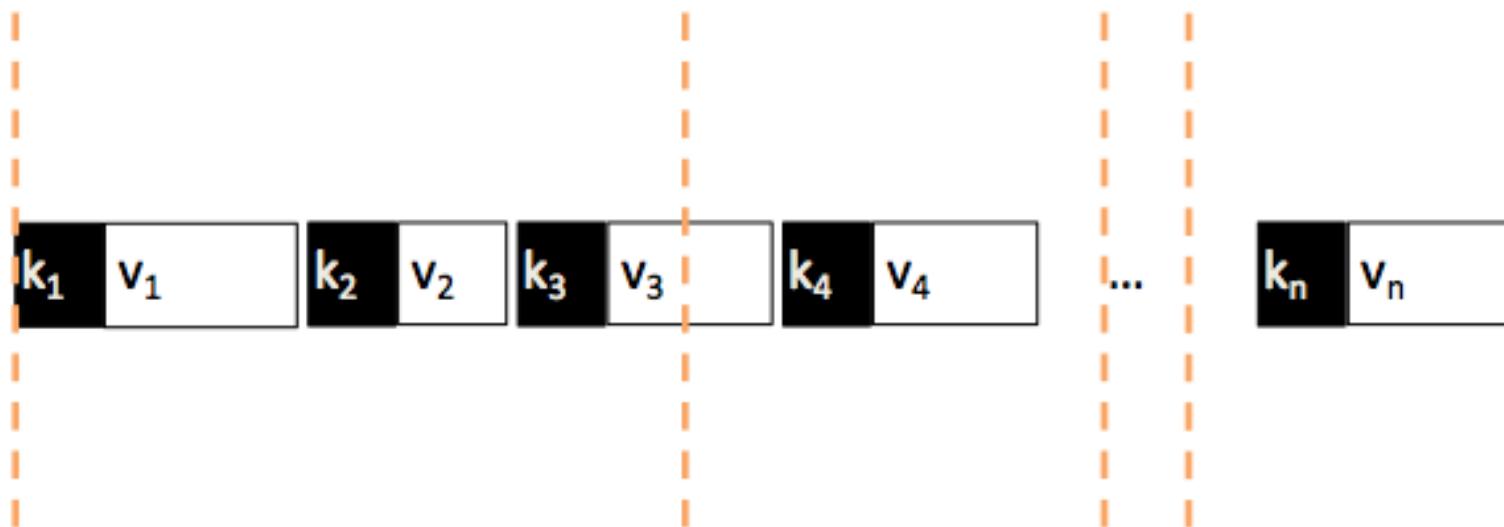
How FileInputFormat Works

- All file-based InputFormats inherit from FileInputFormat
- FileInputFormat computes InputSplits based on the size of each file, in bytes
 - HDFS block size is treated as an upper bound for InputSplit size
 - Lower bound can be specified in your driver code
- Important: InputSplits do not respect record boundaries!

What RecordReaders Do

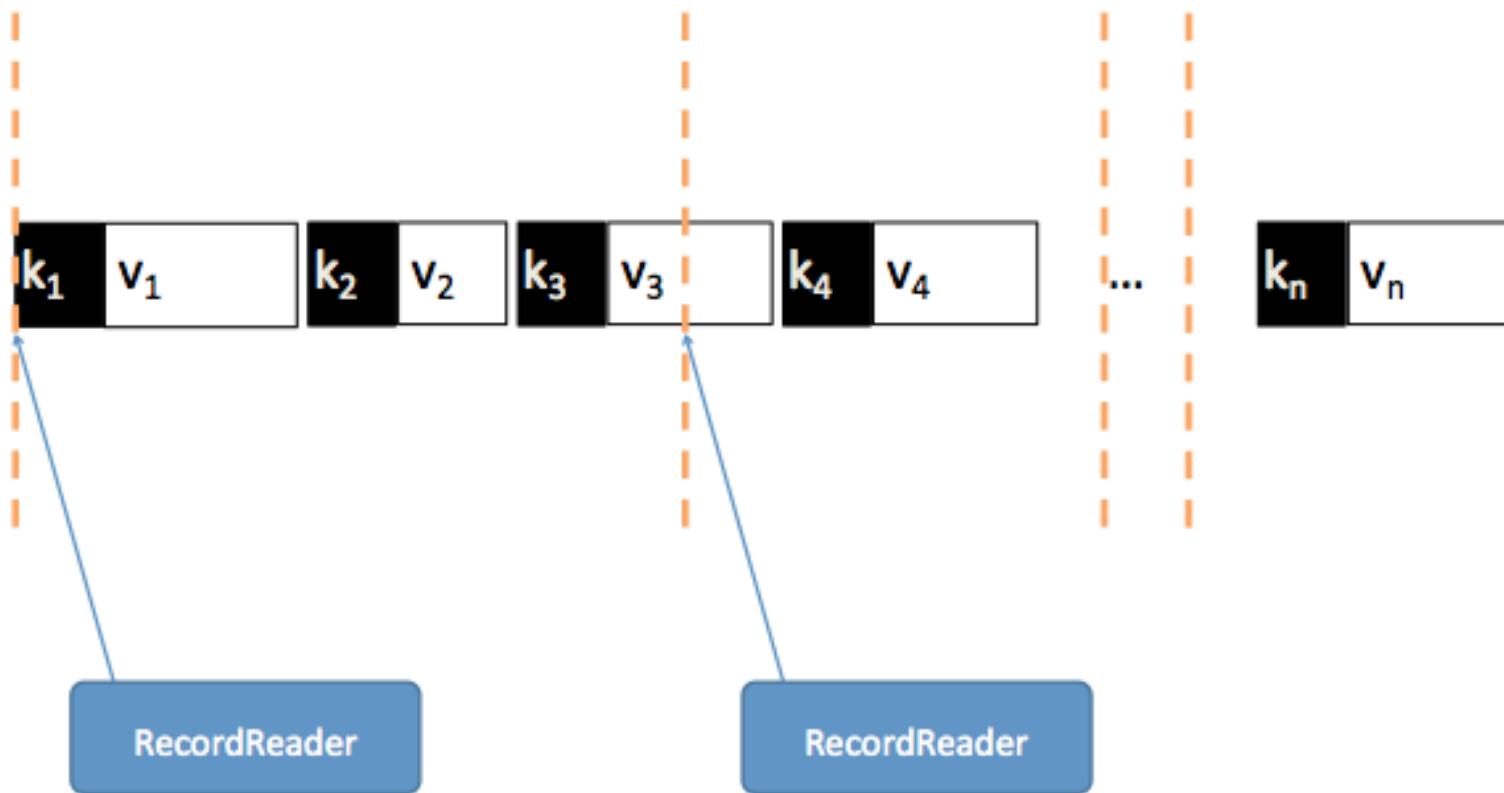
- **InputSplits are handed to the RecordReaders**
 - Specified by the path, starting position offset, length
- **RecordReaders must:**
 - Ensure each (key, value) pair is processed
 - Ensure no (key, value) pair is processed more than once
 - Handle (key, value) pairs which are split across InputSplits

Sample InputSplit



InputSplits

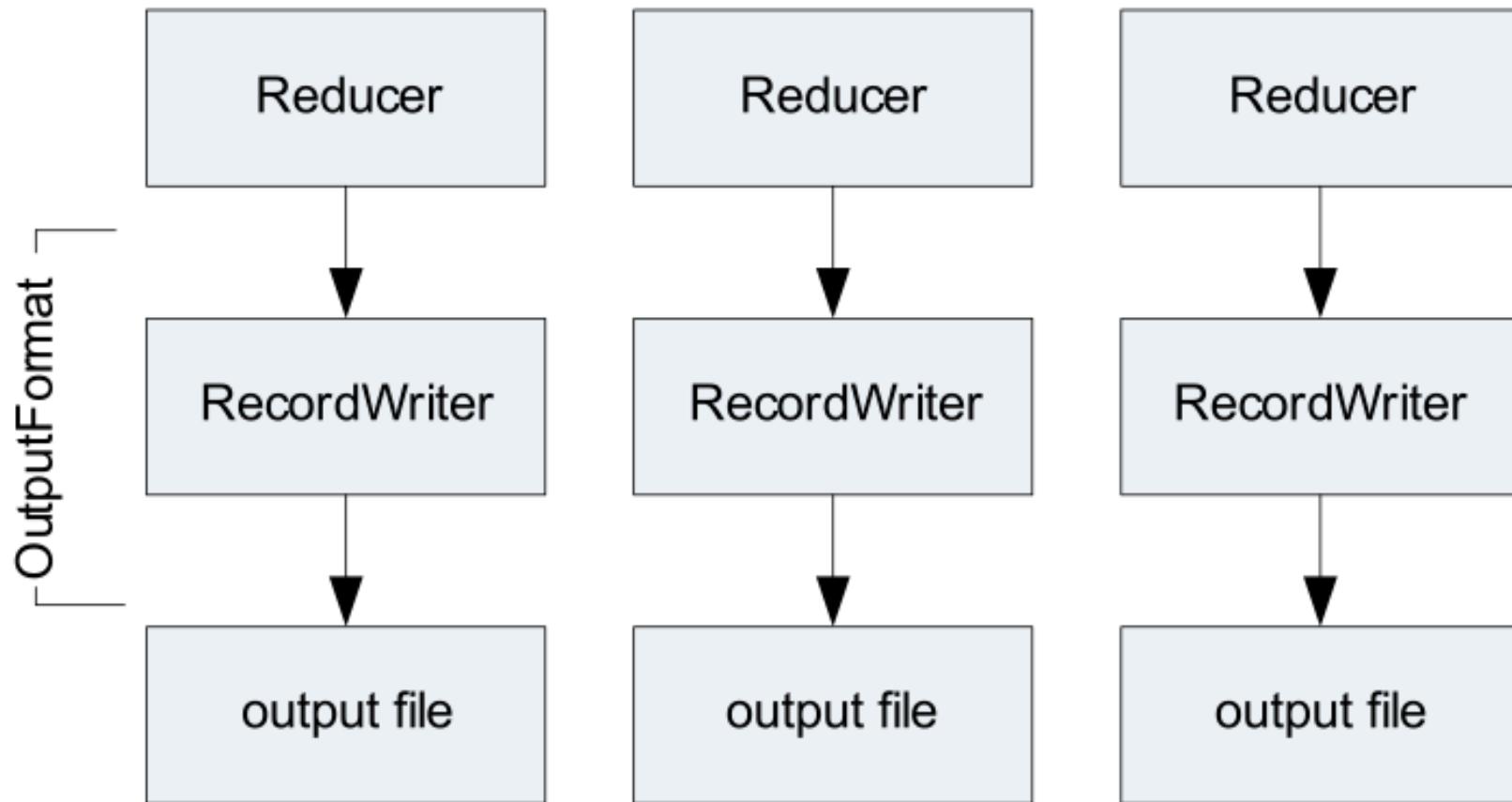
From InputSplits to RecordReaders



Writing Custom InputFormats

- Use `FileInputFormat` as a starting point
 - Extend it
- Write your own custom `RecordReader`
- Override `getRecordReader` method in `FileInputFormat`
- Override `isSplittable` if you don't want input files to be split

Reprise: Role of the OutputFormat



OutputFormat

- **OutputFormats work much like InputFormat classes**
- **Custom OutputFormats must provide a RecordWriter implementation**

More Advanced MapReduce Programming

A Recap of the MapReduce Flow

Custom Writables and WritableComparables

Saving Binary Data

Creating InputFormats and OutputFormats

 **Hands-On Exercise**

Conclusion

Hands-On Exercise

- In this Hands-On Exercise you will create a custom Writable to extend the word co-occurrence program you wrote earlier in the course
- Please refer to the Hands-On Exercise Manual

More Advanced MapReduce Programming

A Recap of the MapReduce Flow

Custom Writables and WritableComparables

Saving Binary Data

Creating InputFormats and OutputFormats

Hands-On Exercise



Conclusion

Conclusion

In this chapter you have learned

- **How to create custom Writables and WritableComparables**
- **How to save binary data using SequenceFiles and Avro**
- **How to build custom InputFormats and OutputFormats**



Chapter 11

Joining Data Sets in MapReduce Jobs

Course Chapters

- Introduction
- The Motivation For Hadoop
- Hadoop: Basic Contents
- Writing a MapReduce Program
- Integrating Hadoop Into The Workflow
- Delving Deeper Into The Hadoop API
- Common MapReduce Algorithms
- An Introduction to Hive and Pig
- Practical Development Tips and Techniques
- More Advanced MapReduce Programming
- Joining Data Sets in MapReduce Jobs**
- Graph Manipulation In MapReduce
- An Introduction to Oozie
- Conclusion
- Appendix: Cloudera Enterprise

Joining Data Sets in MapReduce Jobs

In this chapter you will learn

- How to write a Map-side join
- How to implement the secondary sort
- How to write a Reduce-side join

Joining Data Sets in MapReduce Jobs



Introduction

Map-Side Joins

The Secondary Sort

Reduce-Side Joins

Conclusion

Introduction

- We frequently need to join data together from two sources as part of a MapReduce job
 - Lookup tables
 - Data from database tables
 - Etc
- There are two fundamental approaches: Map-side joins and Reduce-side joins
- Map-side joins are easier to write, but have potential scaling issues
- We will investigate both types of joins in this chapter

But First...

- **But first...**
- **Avoid writing joins in Java MapReduce if you can!**
- **Abstractions such as Pig and Hive are much easier to use**
 - Save hours of programming
- **If you are dealing with text-based data, there really is no reason not to use Pig or Hive**

Joining Data Sets in MapReduce Jobs

Introduction

 **Map-Side Joins**

Reduce-Side Joins

Conclusion

Map-Side Joins: The Algorithm

- **Basic idea for Map-side joins:**

- Load one set of data into memory, stored in an associative array
 - Key of the associative array is the join key
 - Map over the other set of data, and perform a lookup on the associative array using the join key
 - If the join key is found, you have a successful join
 - Otherwise, do nothing

Map-Side Joins: Problems, Possible Solutions

- **Map-side joins have scalability issues**
 - The associative array may become too large to fit in memory
- **Possible solution: break one data set into smaller pieces**
 - Load each piece into memory individually, mapping over the second data set each time
 - Then combine the result sets together

Joining Data Sets in MapReduce Jobs

Introduction

Map-Side Joins

 **The Secondary Sort**

Reduce-Side Joins

Conclusion

Secondary Sort: Motivation

- Recall that keys are passed to the Reducer in sorted order
- The list of values for a particular key is not sorted
 - Order may well change between different runs of the MapReduce job
- Sometimes a job needs to receive the values for a particular key in a sorted order
 - This is known as a *secondary sort*

Implementing the Secondary Sort

- To implement a secondary sort, the intermediate key should be a composite of the ‘actual’ (natural) key and the value
- Define a Partitioner which partitions just on the natural key
- Define a Comparator class which sorts on the entire composite key
 - Orders by natural key and, for the same natural key, on the value portion of the key
 - Ensures that the keys are passed to the Reducer in the desired order
 - Specified in the driver code by

```
conf.setOutputKeyComparatorClass(MyOKCC.class);
```

Implementing the Secondary Sort (cont'd)

- Now we know that all values for the same natural key will go to the same Reducer
 - And they will be in the order we desire
- We must now ensure that all the values for the same natural key are passed in one call to the Reducer
- Achieved by defining a Grouping Comparator class which partitions just on the natural key
 - Determines which keys and values are passed in a single call to the Reducer.
 - Specified in the driver code by

```
conf.setOutputValueGroupingComparator(MyOVGC.class);
```

Secondary Sort: Example

- Assume we have input with (key, value) pairs like this

```
foo 98
foo 101
bar 12
baz 18
foo 22
bar 55
baz 123
```

- We want the Reducer to receive the intermediate data for each key in descending numerical order

Secondary Sort: Example (cont'd)

- Write the Mapper such that they key is a composite of the natural key and value
 - For example, intermediate output may look like this:

```
('foo#98', 98)
('foo#101', 101)
('bar#12', 12)
('baz#18', 18)
('foo#22', 22)
('bar#55', 55)
('baz#123', 123)
```

Secondary Sort: Example (cont'd)

- Write an **OutputKeyComparatorClass** which sorts on natural key, and for identical natural keys sorts on the value portion in descending order
 - Will result in keys being passed to the Reducer in this order:

```
('bar#55', 55)
('bar#12', 12)
('baz#123', 123)
('baz#18', 18)
('foo#101', 101)
('foo#98', 98)
('foo#22', 22)
```

Secondary Sort: Example (cont'd)

- Finally, write an **OutputValueGroupingComparator** which just examines the first portion of the key
 - Ensures that values associated with the same natural key will be sent to the same pass of the Reducer
 - But they're sorted in descending order, as we required

Joining Data Sets in MapReduce Jobs

Introduction

Map-Side Joins

The Secondary Sort

 **Reduce-Side Joins**

Conclusion

Reduce-Side Joins: The Basic Concept

- For a Reduce-side join, the basic concept is:
 - Map over both data sets
 - Emit a (key, value) pair for each record
 - Key is the join key, value is the entire record
 - In the Reducer, do the actual join
 - Because of the Shuffle and Sort, values with the same key are brought together

Reduce-Side Joins: Example

- Example input data:

```
EMP: 42, Aaron, loc(13)  
LOC: 13, New York City
```

- Required output:

```
EMP: 42, Aaron, loc(13), New York City
```

Example Record Data Structure

- A data structure to hold a record could look like this:

```
class Record {  
    enum Typ { emp, loc };  
    Typ type;  
  
    String empName;  
    int empId;  
    int locId;  
    String locationName;  
}
```

Reduce-Side Join: Mapper

```
void map(k, v) {  
    Record r = parse(v);  
    emit (r.locId, r);  
}
```

Reduce-Side Join: Reducer

```
void reduce(k, values) {  
    Record thisLocation;  
    List<Record> employees;  
  
    for (Record v in values) {  
        if (v.type == Typ.loc) {  
            thisLocation = v;  
        } else {  
            employees.add(v);  
        }  
    }  
  
    for (Record e in employees) {  
        e.locationName = thisLocation.locationName;  
        emit(e);  
    }  
}
```

Scalability Problems With Our Reducer

- All employees for a given location must potentially be buffered in the Reducer
 - Could result in out-of-memory errors for large data sets
- Solution: Ensure the location record is the first one to arrive at the Reducer
 - Using a Secondary Sort

A Better Intermediate Key

```
class LocKey {  
    boolean isPrimary;  
    int locId;  
  
    public int compareTo(LocKey k) {  
        if (locId == k.locId) {  
            return Boolean.compare(k.isPrimary, isPrimary);  
        } else {  
            return Integer.compare(locId, k.locId);  
        }  
    }  
  
    public int hashCode() {  
        return locId;  
    }  
}
```

A Better Intermediate Key (cont'd)

```
class LocKey {  
    boolean isPrimary;  
    int locId;  
  
    public int compareTo(LocKey k) {  
        if (locId == k.locId) {  
            return Boolean.compare(k.isPrimary, isPrimary);  
        } else if (isPrimary && !k.isPrimary) {  
            return -1;  
        } else if (!isPrimary && k.isPrimary) {  
            return 1;  
        } else {  
            return Integer.compare(locId, k.locId);  
        }  
    }  
  
    public int hashCode() {  
        return locId;  
    }  
}
```

The hashCode means that all records with the same key will go to the same Reducer

A Better Intermediate Key (cont'd)

```
class LocKey {  
    boolean isPrimary;  
    int locId;  
  
    public int compareTo(LocKey k) {  
        if (locId == k.locId) {  
            return Boolean.compare(k.isPrimary, isPrimary);  
        } else {  
            return Integer.compare(locId, k.locId);  
        }  
    }  
    public String toString() {  
        return "LocKey{" + locId + ", " + isPrimary + '}';  
    }  
}
```

The `compareTo` method ensures that primary keys will sort earlier than non-primary keys for the same location

A Better Mapper

```
void map(k, v) {  
    Record r = parse(v);  
    if (r.type == Typ.emp) {  
        emit (FK(r.locId), r);  
    } else {  
        emit (PK(r.locId), r);  
    }  
}
```

A Better Reducer

```
Record thisLoc;

void reduce(k, values) {
    for (Record v in values) {
        if (v.type == Typ.loc) {
            thisLoc = v;
        } else {
            v.locationName = thisLoc.locationName;
            emit(v);
        }
    }
}
```

Create a Grouping Comparator...

```
Class LocIDComparator implements comparator ()  
    extends WritableComparator {  
    public int compare(Record r1, Record r2) {  
        return Integer.compare(r1.locId, r2.locId);  
    }  
}
```

...And Configure Hadoop To Use It In The Driver

```
conf.setOutputValueGroupingClass(LocIDComparator.class)
```

Joining Data Sets in MapReduce Jobs

Introduction

Map-Side Joins

Reduce-Side Joins



Conclusion

Conclusion

In this chapter you have learned

- **How to join write a Map-side join**
- **How to implement the secondary sort**
- **How to write a Reduce-side join**



Chapter 12

Graph Manipulation

in MapReduce

Course Chapters

- Introduction
- The Motivation For Hadoop
- Hadoop: Basic Contents
- Writing a MapReduce Program
- Integrating Hadoop Into The Workflow
- Delving Deeper Into The Hadoop API
- Common MapReduce Algorithms
- An Introduction to Hive and Pig
- Practical Development Tips and Techniques
- More Advanced MapReduce Programming
- Joining Data Sets in MapReduce Jobs
- Graph Manipulation In MapReduce**
- An Introduction to Oozie
- Conclusion
- Appendix: Cloudera Enterprise

Graph Manipulation in MapReduce

In this chapter you will learn

- Best practices for representing graphs in Hadoop
- How to implement a single source shortest path algorithm in MapReduce

Graph Manipulation in MapReduce



Introduction

Representing Graphs

Implementing Single Source Shortest Path

Conclusion

Introduction: What Is A Graph?

- **Loosely speaking, a graph is a set of vertices, or nodes, connected by edges, or lines**
- **There are many different types of graphs**
 - Directed
 - Undirected
 - Cyclic
 - Acyclic
 - DAG (Directed, Acyclic Graph) is a very common graph type

What Can Graphs Represent?

- **Graphs are everywhere**

- Hyperlink structure of the Web
- Physical structure of computers on a network
- Roadmaps
- Airline flights
- Social networks
- Etc.

Examples of Graph Problems

- **Finding the shortest path through a graph**
 - Routing Internet traffic
 - Giving driving directions
- **Finding the minimum spanning tree**
 - Lowest-cost way of connecting all nodes in a graph
 - Example: telecoms company laying fiber
 - Must cover all customers
 - Need to minimize fiber used
- **Finding maximum flow**
 - Move the most amount of ‘traffic’ through a network
 - Example: airline scheduling

Examples of Graph Problems (cont'd)

- **Finding critical nodes without which a graph would break into disjoint components**
 - Controlling the spread of epidemics
 - Breaking up terrorist cells

Graphs and MapReduce

- **Graph algorithms typically involve:**
 - Performing computations at each vertex
 - Traversing the graph in some manner
- **Key questions:**
 - How do we represent graph data in MapReduce?
 - How do we traverse a graph in MapReduce?

Graph Manipulation in MapReduce

Introduction

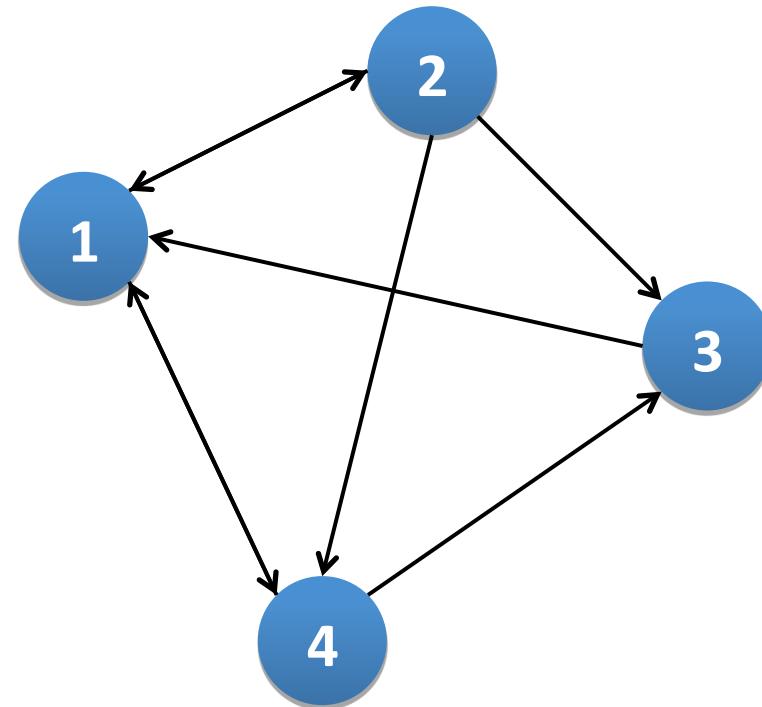
 **Representing Graphs**

Implementing Single Source Shortest Path

Conclusion

Representing Graphs

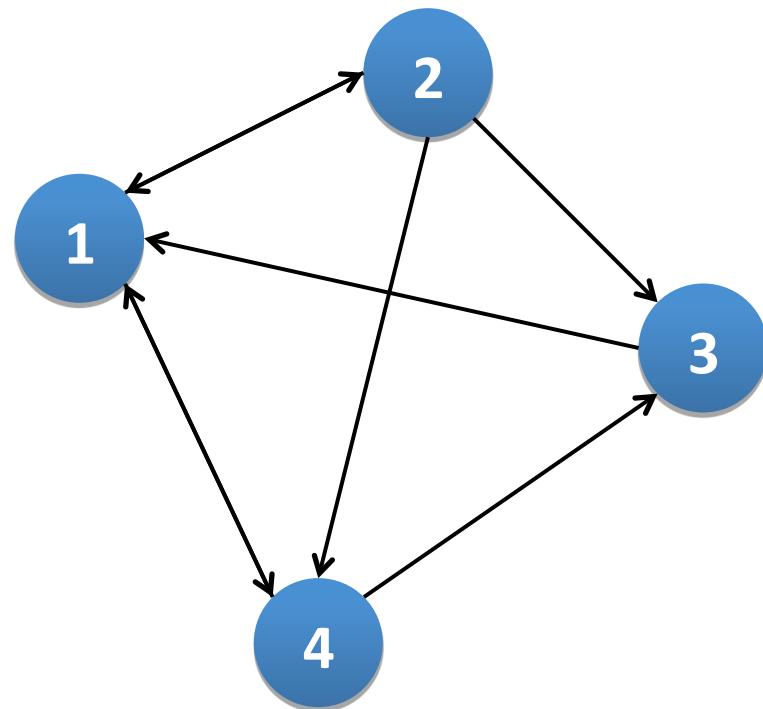
- Imagine we want to represent this simple graph:
- Two approaches:
 - Adjacency matrices
 - Adjacency lists



Adjacency Matrices

- Represent the graph as an $n \times n$ square matrix

	v_1	v_2	v_3	v_4
v_1	0	1	0	1
v_2	1	0	1	1
v_3	1	0	0	0
v_4	1	0	1	0



Adjacency Matrices: Critique

- **Advantages:**

- Naturally encapsulates iteration over nodes
 - Rows and columns correspond to inlinks and outlinks

- **Disadvantages:**

- Lots of zeros for sparse matrices
 - Lots of wasted space

Adjacency Lists

- Take an adjacency matrix... and throw away all the zeros

	v₁	v₂	v₃	v₄
v₁	0	1	0	1
v₂	1	0	1	1
v₃	1	0	0	0
v₄	1	0	1	0



v₁: v₂, v₄
v₂: v₁, v₃, v₄
v₃: v₁
v₄: v₁, v₃

Adjacency Lists: Critique

- **Advantages:**

- Much more compact representation
- Easy to compute outlinks
- Graph structure can be broken up and distributed

- **Disadvantages:**

- More difficult to compute inlinks

Encoding Adjacency Lists

- **Adjacency lists are the preferred way of representing graphs in MapReduce**
 - Typically we represent each vertex (node) with an id number
 - A four-byte int usually suffices
- **Typical encoding format (Writable)**
 - Four-byte int: vertex id of the source
 - Two-byte int: number of outgoing edges
 - Sequence of four-byte ints: destination vertices

$v_1: v_2, v_4$

$v_2: v_1, v_3, v_4$

$v_3: v_1$

$v_4: v_1, v_3$



1: [2] 2, 4

2: [3] 1, 3, 4

3: [1] 1

4: [2] 1, 3

Graph Manipulation in MapReduce

Introduction

Representing Graphs

 Implementing Single Source Shortest Path

Conclusion

Single Source Shortest Path

- **Problem:** find the shortest path from a source node to one or more target nodes
- **Serial algorithm:** Dijkstra's Algorithm
 - Not suitable for parallelization
- **MapReduce algorithm:** parallel breadth-first search

Parallel Breadth-First Search

- **The algorithm, intuitively:**

- Distance to the source = 0
- For all nodes directly reachable from the source, distance = 1
- For all nodes reachable from some node n in the graph, distance from source = $1 + \min(\text{distance to that node})$

Parallel Breadth-First Search: Algorithm

- **Mapper:**

- Input key is some vertex id
 - Input value is D (distance from source), adjacency list
 - Processing: For all nodes in the adjacency list, emit (node id, $D + 1$)
 - If the distance to this node is D , then the distance to any node reachable from this node is $D + 1$

- **Reducer:**

- Receives vertex and list of distance values
 - Processing: Selects the shortest distance value for that node

Iterations of Parallel BFS

- A MapReduce job corresponds to one iteration of parallel breadth-first search
 - Each iteration advances the ‘known frontier’ by one hop
 - Iteration is accomplished by using the output from one job as the input to the next
- How many iterations are needed?
 - Multiple iterations are needed to explore the entire graph
 - As many as the diameter of the graph
 - Graph diameters are surprisingly small, even for large graphs
 - ‘Six degrees of separation’
- Controlling iterations in Hadoop
 - Use counters; when you reach a node, ‘count’ it
 - At the end of each iteration, check the counters
 - When you’ve reached all the nodes, you’re finished

One More Trick: Preserving Graph Structure

- **Characteristics of Parallel BFS**

- Mappers emit distances, Reducers select the shortest distance
 - Output of the Reducers becomes the input of the Mappers for the next iteration

- **Problem: where did the graph structure (adjacency lists) go?**

- **Solution: Mapper must emit the adjacency lists as well**

- Mapper emits two types of key-value pairs
 - Representing distances
 - Representing adjacency lists
 - Reducer recovers the adjacency list and preserves it for the next iteration

Parallel BFS: Pseudo-Code

```
1: class MAPPER
2:   method MAP(nid  $n$ , node  $N$ )
3:      $d \leftarrow N.\text{DISTANCE}$ 
4:     EMIT(nid  $n$ ,  $N$ )                                 $\triangleright$  Pass along graph structure
5:     for all nodeid  $m \in N.\text{ADJACENCYLIST}$  do
6:       EMIT(nid  $m$ ,  $d + 1$ )                          $\triangleright$  Emit distances to reachable nodes

1: class REDUCER
2:   method REDUCE(nid  $m$ , [ $d_1, d_2, \dots$ ])
3:      $d_{min} \leftarrow \infty$ 
4:      $M \leftarrow \emptyset$ 
5:     for all  $d \in \text{counts } [d_1, d_2, \dots]$  do
6:       if ISNODE( $d$ ) then
7:          $M \leftarrow d$                                  $\triangleright$  Recover graph structure
8:       else if  $d < d_{min}$  then
9:          $d_{min} \leftarrow d$                            $\triangleright$  Look for shorter distance
10:     $M.\text{DISTANCE} \leftarrow d_{min}$                  $\triangleright$  Update shortest distance
11:    EMIT(nid  $m$ , node  $M$ )
```

From Lin & Dyer. (2010) Data-Intensive Text Processing with MapReduce

Parallel BFS: Demonstration

- Your instructor will now demonstrate the parallel breadth-first search algorithm

Graph Algorithms: General Thoughts

- **MapReduce is adept at manipulating graphs**
 - Store graphs as adjacency lists
- **Typically, MapReduce graph algorithms are iterative**
 - Iterate until some termination condition is met
 - Remember to pass the graph structure from one iteration to the next

Graph Manipulation in MapReduce

Introduction

Representing Graphs

Implementing Single Source Shortest Path



Conclusion

Conclusion

In this chapter you have learned

- **Best practices for representing graphs in Hadoop**
- **How to implement a single source shortest path algorithm in MapReduce**



Chapter 13

An Introduction To Oozie

Course Chapters

- Introduction
 - The Motivation For Hadoop
 - Hadoop: Basic Contents
 - Writing a MapReduce Program
 - Integrating Hadoop Into The Workflow
 - Delving Deeper Into The Hadoop API
 - Common MapReduce Algorithms
 - An Introduction to Hive and Pig
 - Practical Development Tips and Techniques
 - More Advanced MapReduce Programming
 - Joining Data Sets in MapReduce Jobs
 - Graph Manipulation In MapReduce
- An Introduction to Oozie**
- Conclusion
 - Appendix: Cloudera Enterprise

An Introduction to Oozie

In this chapter you will learn

- **What Oozie is**
- **How to create Oozie workflows**

An Introduction to Oozie



The Motivation for Oozie

Creating Oozie workflows

Hands-On Exercise

Conclusion

The Motivation for Oozie

- Many problems cannot be solved with a single MapReduce job
- Instead, a **workflow** of jobs must be created
- Simple workflow:
 - Run JobA
 - Use output of JobA as input to JobB
 - Use output of JobB as input to JobC
 - Output of JobC is the final required output
- Easy if the workflow is linear like this
 - Can be created as standard Driver code

The Motivation for Oozie (cont'd)

- If the workflow is more complex, Driver code becomes much more difficult to maintain
- Example: running multiple jobs in parallel, using the output from all of those jobs as the input to the next job
- Example: including Hive or Pig jobs as part of the workflow

What is Oozie?

- **Oozie is a ‘workflow engine’**
- **Runs on a server**
 - Typically outside the cluster
- **Runs workflows of Hadoop jobs**
 - Including Pig, Hive, Sqoop jobs
 - Submits those jobs to the cluster based on a workflow definition
- **Workflow definitions are submitted via http**
- **Jobs can be run at specific times**
 - One-off or recurring jobs
- **Jobs can be run when data is present in a directory**

An Introduction to Oozie

The Motivation for Oozie

 **Creating Oozie workflows**

Hands-On Exercise

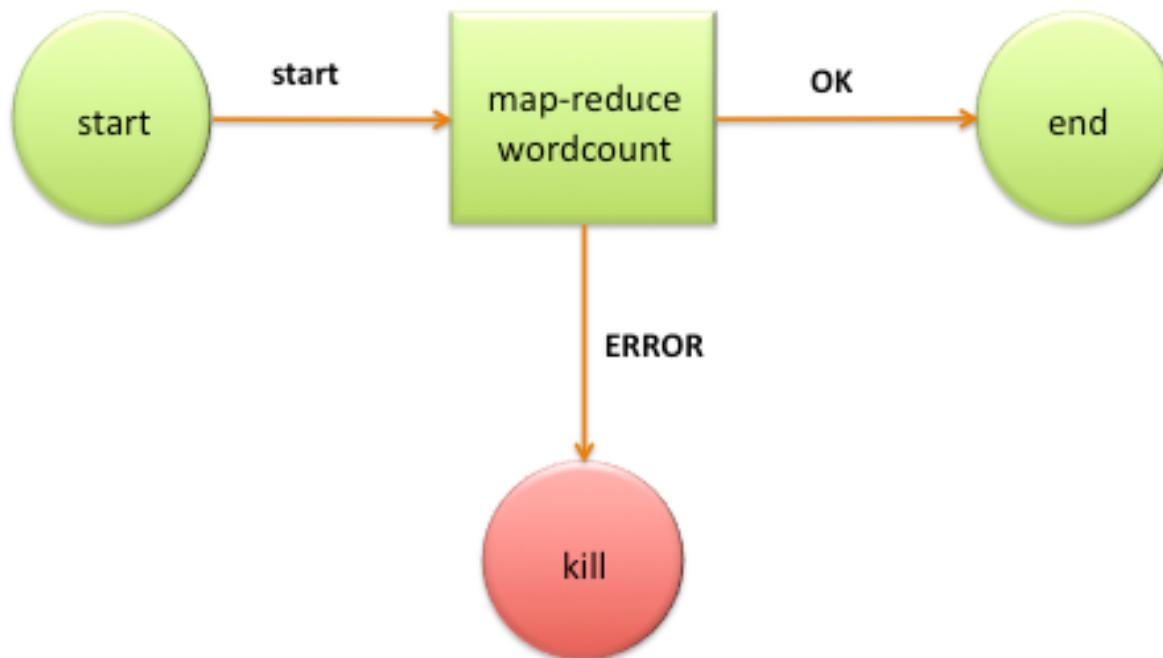
Conclusion

Oozie Workflow Basics

- Oozie workflows are written in XML
- Workflow is a collection of actions
 - MapReduce jobs, Pig jobs, Hive jobs etc.
- A workflow consists of *control flow nodes* and *action nodes*
- Control flow nodes define the beginning and end of a workflow
 - They provide methods to determine the workflow execution path
 - Example: Run multiple jobs simultaneously
- Action nodes trigger the execution of a processing task
 - A MapReduce job
 - A Pig job
 - A Sqoop data import job
 - Etc.

Simple Oozie Example

- Simple example workflow for WordCount:



Simple Oozie Example (cont'd)

```
<workflow-app name='wordcount-wf' xmlns="uri:oozie:workflow:0.1">
  <start to='wordcount'/>
  <action name='wordcount'>
    <map-reduce>
      <job-tracker>${jobTracker}</job-tracker>
      <name-node>${nameNode}</name-node>
      <configuration>
        <property>
          <name>mapred.mapper.class</name>
          <value>org.myorg.WordCount.Map</value>
        </property>
        <property>
          <name>mapred.reducer.class</name>
          <value>org.myorg.WordCount.Reduce</value>
        </property>
        <property>
          <name>mapred.input.dir</name>
          <value>${inputDir}</value>
        </property>
        <property>
          <name>mapred.output.dir</name>
          <value>${outputDir}</value>
        </property>
      </configuration>
    </map-reduce>
    <ok to='end'/>
    <error to='end'/>
  </action>
  <kill name='kill'>
    <message>Something went wrong: ${wf:errorCode('wordcount')}</message>
  </kill/>
  <end name='end' />
</workflow-app>
```

Simple Oozie Example (cont'd)

```
<workflow-app name='wordcount-wf' xmlns="uri:oozie:workflow:0.1">
  <start to='wordcount'/>

  A workflow is wrapped in the workflow entity

  <property>
    <name>mapred.mapper.class</name>
    <value>org.myorg.WordCount.Map</value>
  </property>
  <property>
    <name>mapred.reducer.class</name>
    <value>org.myorg.WordCount.Reduce</value>
  </property>
  <property>
    <name>mapred.input.dir</name>
    <value>${inputDir}</value>
  </property>
  <property>
    <name>mapred.output.dir</name>
    <value>${outputDir}</value>
  </property>
  </configuration>
</map-reduce>
<ok to='end'/>
<error to='end' />
</action>
<kill name='kill'>
  <message>Something went wrong: ${wf:errorCode('wordcount')}</message>
</kill/>
<end name='end' />
</workflow-app>
```

Simple Oozie Example (cont'd)

```
<workflow-app name='wordcount-wf' xmlns="uri:oozie:workflow:0.1">
  <start to='wordcount'/>
  <action name='wordcount'>
```

The start node is the control node which tells Oozie which workflow node should be run first. There must be one start node in an Oozie workflow. In our example, we are telling Oozie to start by transitioning to the wordcount workflow node.

```
    <name>mapred.output.dir</name>
    <value>${outputDir}</value>
  </property>
</configuration>
</map-reduce>
<ok to='end'/>
<error to='end' />
</action>
<kill name='kill'>
  <message>Something went wrong: ${wf:errorCode('wordcount')}</message>
</kill/>
<end name='end' />
</workflow-app>
```

Simple Oozie Example (cont'd)

```
<workflow-app name='wordcount-wf' xmlns="uri:oozie:workflow:0.1">
  <start to='wordcount'/>
  <action name='wordcount'>
    <map-reduce>
      <configuration>
        <property>
          <name>mapred.input.dir</name>
          <value>${inputDir}</value>
        </property>
        <property>
          <name>mapred.output.dir</name>
          <value>${outputDir}</value>
        </property>
      </configuration>
    </map-reduce>
    <ok to='end'/>
    <error to='end'/>
  </action>
  <kill name='kill'>
    <message>Something went wrong: ${wf:errorCode('wordcount')}</message>
  </kill/>
  <end name='end' />
</workflow-app>
```

The wordcount action node defines a map-reduce action – a standard Java MapReduce job.

Simple Oozie Example (cont'd)

```
<workflow-app name='wordcount-wf' xmlns="uri:oozie:workflow:0.1">
  <start to='wordcount'/>
  <action name='wordcount'>
    <map-reduce>
      <job-tracker>${jobTracker}</job-tracker>
      <name-node>${nameNode}</name-node>
      <configuration>
        <property>
          <name>mapred.mapper.class</name>
          <value>org.myorg.WordCount.Map</value>
        </property>
        <property>
          <name>mapred.reducer.class</name>
          <value>org.myorg.WordCount.Reduce</value>
        </property>
        <property>
          <name>mapred.input.dir</name>
          <value>${inputDir}</value>
        </property>
        <property>
          <name>mapred.output.dir</name>
          <value>${outputDir}</value>
        </property>
      </configuration>
    </map-reduce>
  </action>
</workflow-app>
```

Within the action, we define the job's properties.

Simple Oozie Example (cont'd)

```
<workflow-app name='wordcount-wf' xmlns="uri:oozie:workflow:0.1">
  <start to='wordcount'/>
  <action name='wordcount'>
    <map-reduce>
      <job-tracker>${jobTracker}</job-tracker>
      <name-node>${nameNode}</name-node>
      <configuration>
        <property>
          <name>mapred.mapper.class</name>
          <value>org.myorg.WordCount.Map</value>
        </property>
        <property>
```

We specify what to do if the action ends successfully, and what to do if it fails. In this example, in both cases we transition to the end node.

```
</map-reduce>
  <ok to='end'/>
  <error to='end'/>
</action>
<kill name='kill'>
  <message>Something went wrong: ${wf:errorCode('wordcount')}</message>
</kill/>
<end name='end' />
</workflow-app>
```

Simple Oozie Example (cont'd)

```
<workflow-app name='wordcount-wf' xmlns="uri:oozie:workflow:0.1">
  <start to='wordcount'/>
  <action name='wordcount'>
    <map-reduce>
      <job-tracker>${jobTracker}</job-tracker>
      <name-node>${nameNode}</name-node>
      <configuration>
        <property>
          <name>mapred.mapper.class</name>
          <value>org.myorg.WordCount.Map</value>
        </property>
        <property>
          <name>mapred.reducer.class</name>
          <value>org.myorg.WordCount.Reduce</value>
        </property>
        <property>
          <name>mapred.input.dir</name>
          <value>${inputDir}</value>
        </property>
        <property>
          </kill/>
          <end name='end' />
        </workflow-app>
```

Every workflow must have an `end` node. This indicates that the workflow has completed successfully.

Simple Oozie Example (cont'd)

```
<workflow-app name='wordcount-wf' xmlns="uri:oozie:workflow:0.1">
  <start to='wordcount'/>
  <action name='wordcount'>
    <map-reduce>
      <job-tracker>${jobTracker}</job-tracker>
      <name-node>${nameNode}</name-node>
      <configuration>
        <property>
          <name>mapred.mapper.class</name>
          <value>org.myorg.WordCount.Map</value>
        </property>
        <property>
          <name>mapred.reducer.class</name>
          <value>org.myorg.WordCount.Reduce</value>
        </property>
        <property>
          <name>mapred.input.dir</name>
```

If the workflow reaches a kill node, it will kill all running actions and then terminate with an error. A workflow can have zero or more kill nodes.

```
</action>
<kill name='kill'>
  <message>Something went wrong: ${wf:errorCode('wordcount')}</message>
</kill/>
<end name='end' />
</workflow-app>
```

Other Oozie Control Nodes

- A **decision control node** allows Oozie to determine the workflow execution path based on some criteria
 - Similar to a switch-case statement
- **fork and join control nodes split one execution path into multiple execution paths which run concurrently**
 - fork splits the execution path
 - join waits for all concurrent execution paths to complete before proceeding
 - fork and join are used in pairs

Oozie Workflow Action Nodes

- **map-reduce**
- **fs**
 - Create directories, move or delete files or directories
- **java**
 - Runs the main() method in the specified Java class as a single-Map, Map-only job on the cluster
- **pig**
 - Runs a Pig job
- **hive**
 - Runs a Hive job
- **Sqoop**
 - Runs a Sqoop job
- **email**
 - Sends an e-mail message

Submitting an Oozie Workflow

- To submit an Oozie workflow using the command-line tool:

```
$ oozie job -oozie http://<oozie_server>/oozie  
-config config_file -run
```

- Oozie can also be called from within a Java program
 - Via the Oozie client API

More on Oozie

- For full documentation on Oozie, refer to
<http://docs.cloudera.com>

An Introduction to Oozie

The Motivation for Oozie

Creating Oozie workflows

 **Hands-On Exercise**

Conclusion

Hands-On Exercise

- In this Hands-On Exercise you will run Oozie jobs
- Please refer to the Hands-On Exercise Manual

An Introduction to Oozie

The Motivation for Oozie

Creating Oozie workflows

Hands-On Exercise



Conclusion

Conclusion

In this chapter you have learned

- **What Oozie is**
- **How to create Oozie workflows**



Chapter 14

Conclusion

Conclusion

During this course, you have learned:

- **The core technologies of Hadoop**
- **How HDFS and MapReduce work**
- **What other projects exist in the Hadoop ecosystem**
- **How to develop MapReduce jobs**
- **How Hadoop integrates into the datacenter**
- **Algorithms for common MapReduce tasks**
- **How to create large workflows using Oozie**

Conclusion (cont'd)

- How Hive and Pig can be used for rapid application development
- Best practices for developing and debugging MapReduce jobs
- Advanced features of the Hadoop API
- How to handle graph manipulation problems with MapReduce

Certification

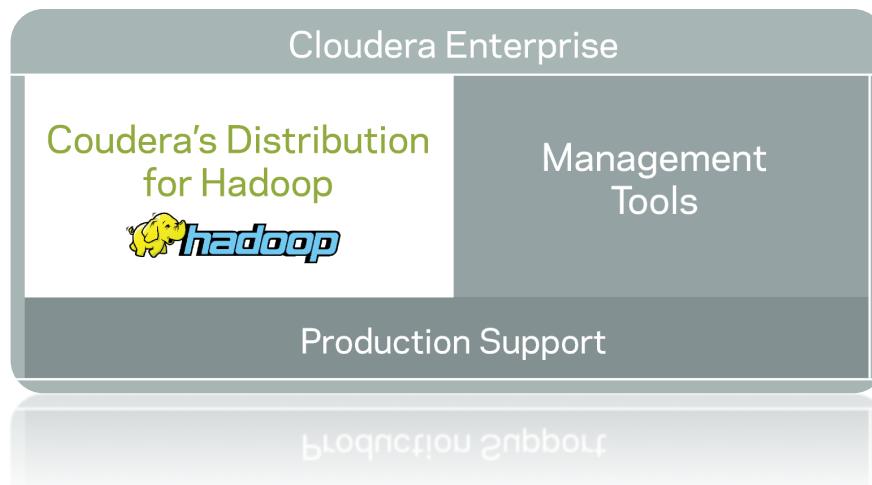
- You can now take the Cloudera Certified Hadoop Developer exam
 - Your instructor will give you information on how to access the exam
-
- Thank you for attending the course!
 - If you have any questions or comments, please contact us via
<http://www.cloudera.com>



Appendix A: Cloudera Enterprise

Cloudera Enterprise

- Reduces the risks of running Hadoop in production
- Improves consistency, compliance and administrative overhead



Management Suite components:

- Service and Configuration Manager (SCM)
- Activity Monitor
- Resource Manager
- Authorization Manager
- The Flume User Interface

- 24x7 Production support for CDH and certified integrations (Oracle, Netezza, Teradata, Greenplum, Aster Data)

Service and Configuration Manager

- Service and Configuration Manager (SCM) is designed to make installing and managing your cluster very easy
- View a ‘dashboard’ of cluster status
- Modify configuration parameters
- Easily start and stop master and slave daemons
- Easily retrieve the configuration files required for client machines to access the cluster

Service and Configuration Manager (cont'd)

The screenshot shows the Cloudera Service and Configuration Manager interface running in Mozilla Firefox. The URL in the address bar is `http://localhost:7180/cmft#/cmf/services/5/status`. The main page displays the status of the HDFS service on node hdfs1. The sidebar on the left shows services like Services, Add Service, HDFS Service hdfs1, MAPREDUCE Service mapreduce1, and Hardware. The central panel has tabs for Status, Instances, Configuration, and History, with Status selected. It shows the following node statuses:

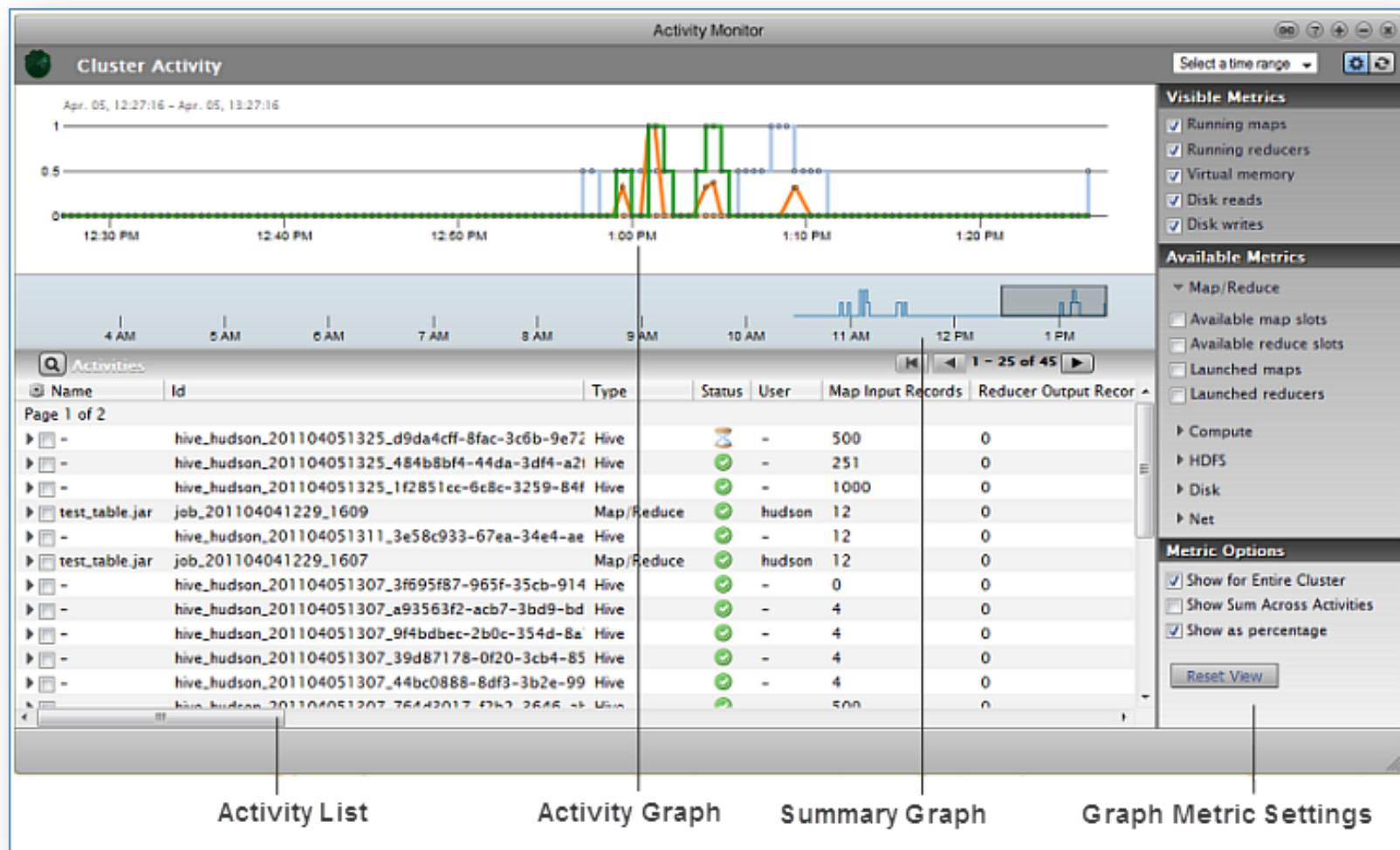
Node Type	Status	Count	Action Buttons
NameNode	✓ 1 Running	1	Start Stop Restart
Secondary NameNode	✓ 1 Running	1	Start Stop Restart
DataNode	✓ 1 Running	1	Start Stop Restart
DataNode	⚠ 1 Running with Concerning Health	1	Start Stop Restart
DataNode	🔴 1 Running with Bad Health	1	Start Stop Restart
Balancer	🌙 1 Stopped	1	

Below the table, there are buttons for Commands (Format, Refresh Node List, CreateTmp, Rebalance, GenerateClient, Delete) and a table for Running Commands. The bottom status bar shows the user is on node1.

Activity Monitor

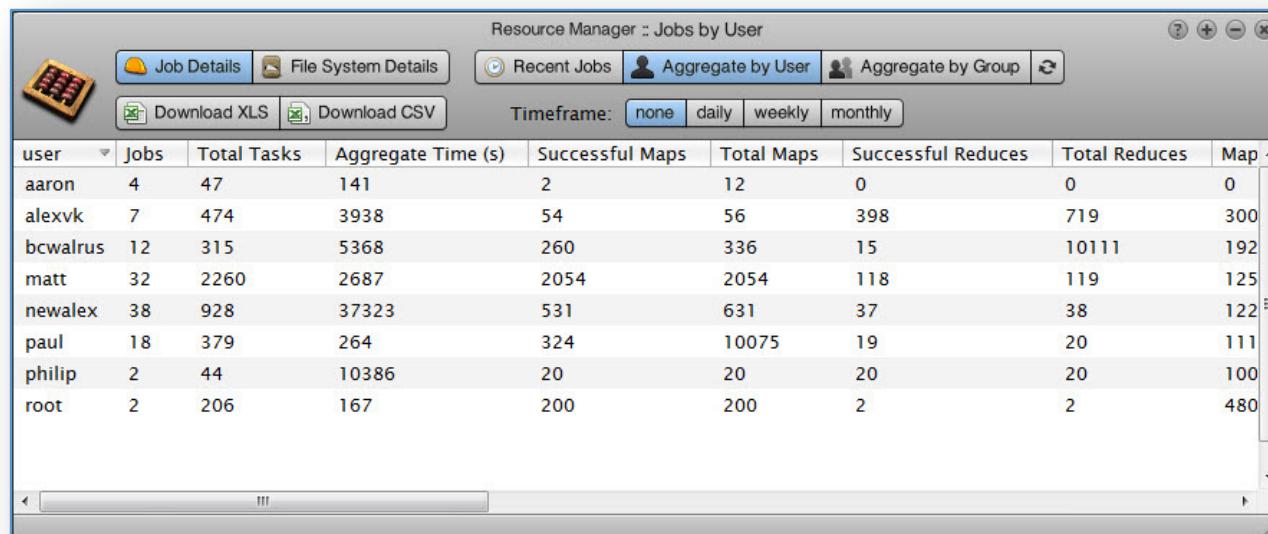
- **Activity Monitor gives an in-depth, comprehensive view of what is happening on the cluster, in real-time**
 - And what has happened in the past
- **Compare the performance of similar jobs**
- **Store historical data**
- **Chart metrics on cluster performance**

Activity Monitor (cont'd)



Resource Manager

- Resource Manager displays the usage of assets within the Hadoop cluster
 - Disk space, processor utilization and more
- Easily configure quotas for disk space and number of files allowed
- Display resource usage history for auditing and internal billing



The screenshot shows a window titled "Resource Manager :: Jobs by User". The window has a toolbar with icons for Job Details, File System Details, Recent Jobs, Aggregate by User, Aggregate by Group, and download options for XLS and CSV. It also includes a timeframe selector for none, daily, weekly, and monthly. The main area is a table with the following data:

user	Jobs	Total Tasks	Aggregate Time (s)	Successful Maps	Total Maps	Successful Reduces	Total Reduces	Map %
aaron	4	47	141	2	12	0	0	0
alexvk	7	474	3938	54	56	398	719	300
bcwalrus	12	315	5368	260	336	15	10111	192
matt	32	2260	2687	2054	2054	118	119	125
newalex	38	928	37323	531	631	37	38	122
paul	18	379	264	324	10075	19	20	111
philip	2	44	10386	20	20	20	20	100
root	2	206	167	200	200	2	2	480

Authorization Manager

- Authorization Manager allows you to provision users, and manage user activity, within the cluster
- Configure permissions based on user or group membership
- Fine-grained control over permissions
- Integrate with Active Directory

The screenshot shows the 'Users' page of the Authorization Manager. The title bar reads 'Users : Authorization Manager'. The top navigation bar includes links for 'Users', 'Groups', 'Permissions', 'Recent Access', 'LDAP', and a search icon. Below the navigation is a toolbar with buttons for 'Import LDAP Users', 'Add User', 'XLS', and 'CSV'. The main content area displays a table of users:

Username	First Name	Last Name	Primary Group	E-mail	Active	Last Login
admin	-	-	-	-	✓	Mon, May 30 2011
ian	Ian	Wrigley	users	-	✓	Mon, May 30 2011
mike	Mike	Smith	-	mike@example.com	✓	Mon, May 30 2011
sample	-	-	-	-	-	Fri, Sep 18 2009
training	-	-	users	-	✓	Mon, May 30 2011

The Flume User Interface

- Flume is designed to collect large amounts of data as that data is created, and ingest it into HDFS
- Flume features:
 - Reliability
 - Scalability
 - Manageability
 - Extensibility
- The Flume User Interface in the Cloudera Management System allows you to configure and monitor Flume via a graphical user interface
 - Create data flows
 - Monitor node usage

Conclusion

- **Cloudera Enterprise makes it easy to run open source Hadoop in production**
- **Includes**
 - Cloudera Distribution including Apache Hadoop (CDH)
 - Cloudera Management Suite (CMS)
 - Production Support
- **Cloudera Management Suite enables you to:**
 - Simplify and accelerate Hadoop deployment
 - Reduce the costs and risks of adopting Hadoop in production
 - Reliably operate Hadoop in production with repeatable success
 - Apply SLAs to Hadoop
 - Increase control over Hadoop cluster provisioning and management