CSC590: Selected Topics
BIG DATA & DATA MINING

Lecture 4

Mar 19, 2014

Dr. Esam A. Alwagait

Apache Hadoop

An Introduction to HDFS and MapReduce

Training Chapters

Training Chapters

Training Chapters

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

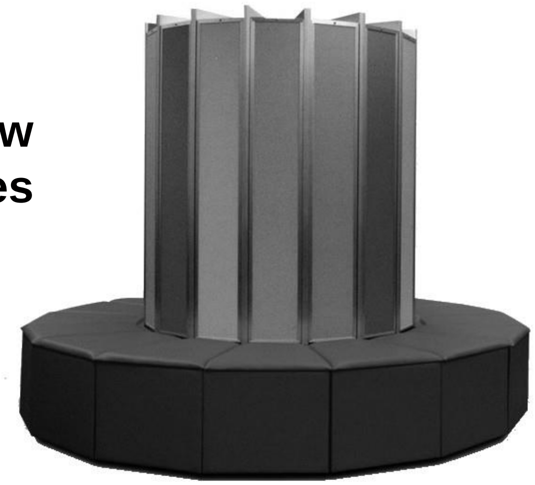
Requirements for a New Approach

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 (Message Passing Interface)
  - PVM (Parallel Virtual Machine)
  - Condor



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 and scalability
- **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 need not worry about network programming, temporal dependencies or low-level infrastructure
- **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**

Training Chapters

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**

Hadoop: Basic Concepts

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
- **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

HDFS Basic Concepts

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

HDFS Basic Concepts (cont'd)

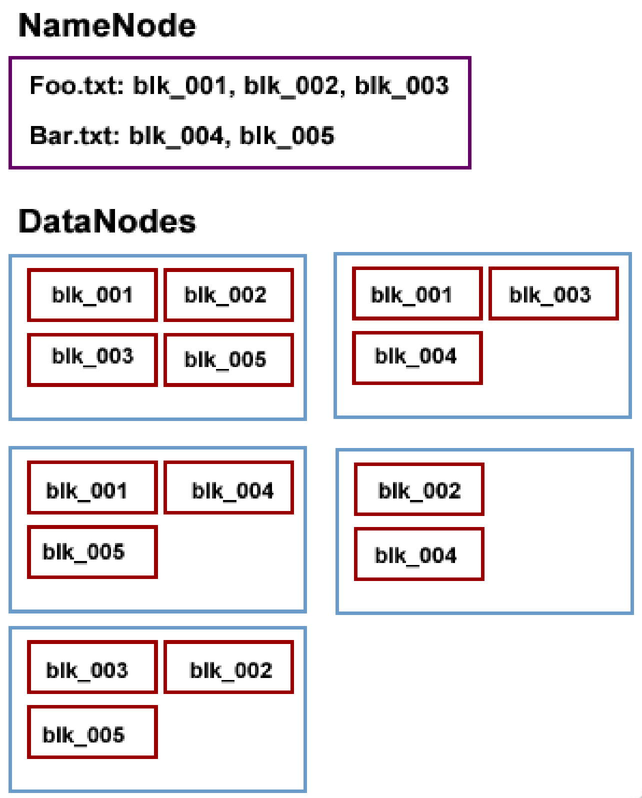
- **HDFS performs best with a 'modest' number of large files**
  - Millions, rather than billions, of files
  - Each file typically 100MB or more
- **Files in HDFS are 'write once'**
  - No random writes to files are allowed
  - Append support is included in Cloudera's Distribution including Apache Hadoop (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
    - Meaning that 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



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**
- **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. Hence, 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 -rmr 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

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

Hadoop: Basic Concepts

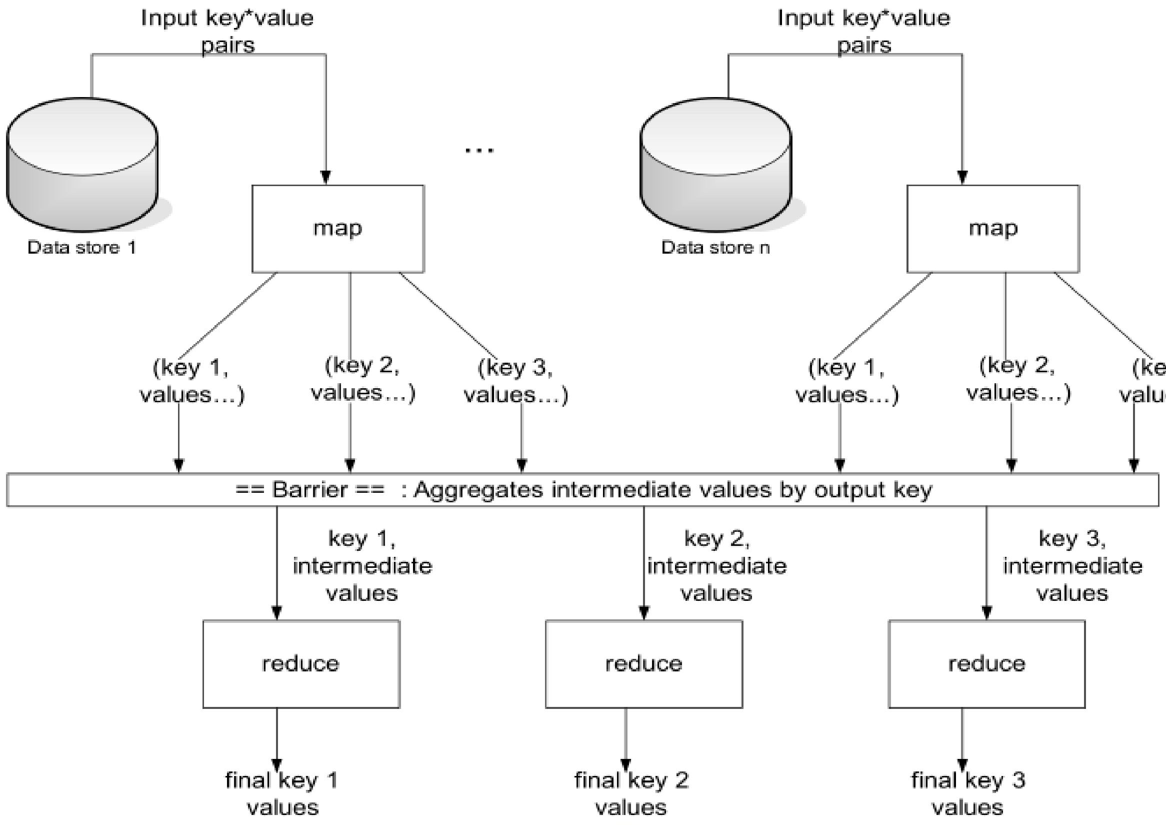
What Is MapReduce?

- **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.toUpper(), v.toUpper())
```

```
('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)
```

```
('bar', [9, 3, -17, 44]) ->    ('bar', 39)

('foo', [123, 100, 77]) -> ('foo', 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 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)
```

MapReduce: Data Locality

- **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
  - Note that this behavior is configurable
    - The developer can specify the percentage of Mappers which should finish before Reducers start retrieving data
  - The developer's `reduce` method still does not start until all intermediate data has been transferred and sorted

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

Hadoop: Basic Concepts

Hands-On Exercise: Running A MapReduce Job

- **In this Hands-On Exercise, you will run a MapReduce job on your pseudo-distributed Hadoop cluster**

Hadoop: Basic Concepts

Installing A Hadoop Cluster

- **Cluster installation is usually performed by the system administrator, and is outside the scope of this talk**
- **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

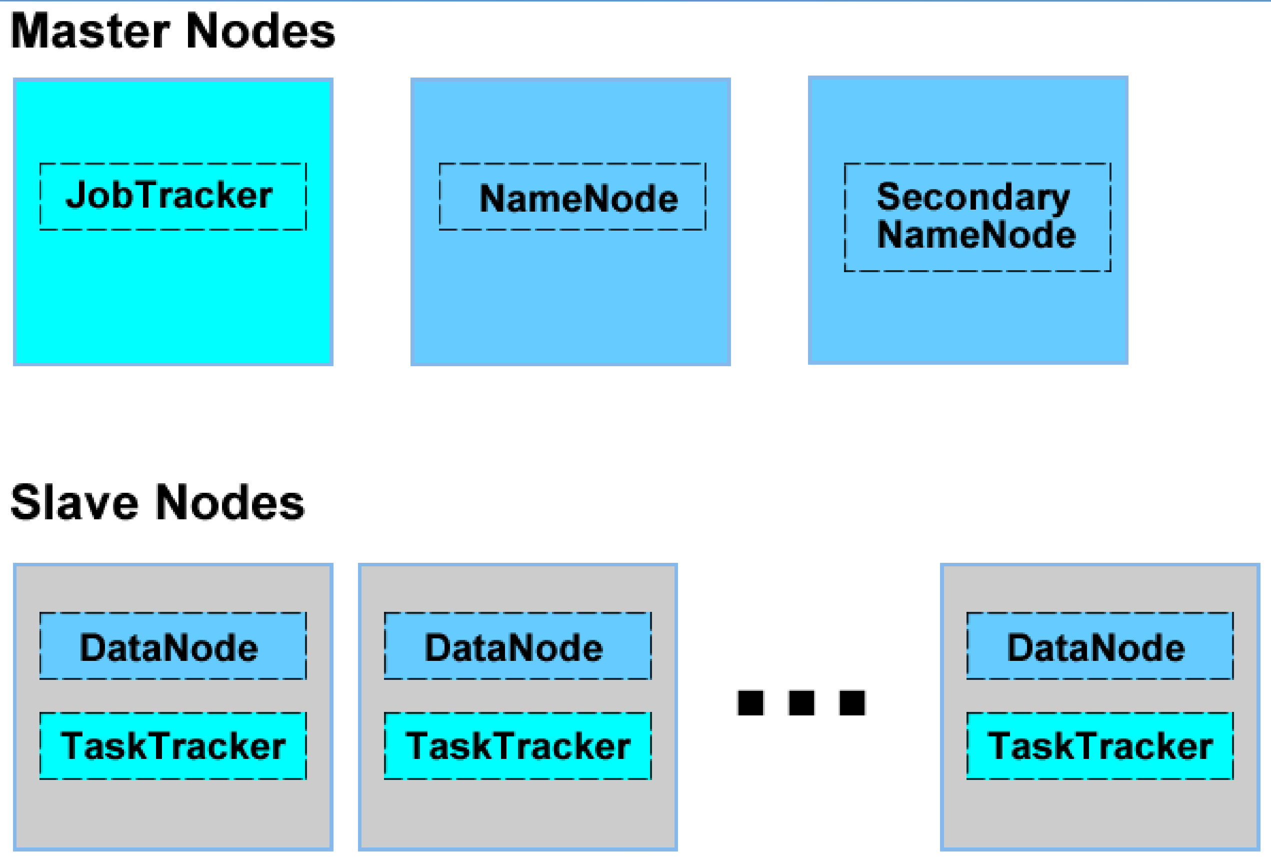
The Five Hadoop Daemons

- **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**
  - Instantiates and monitors 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



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

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

Training Chapters

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

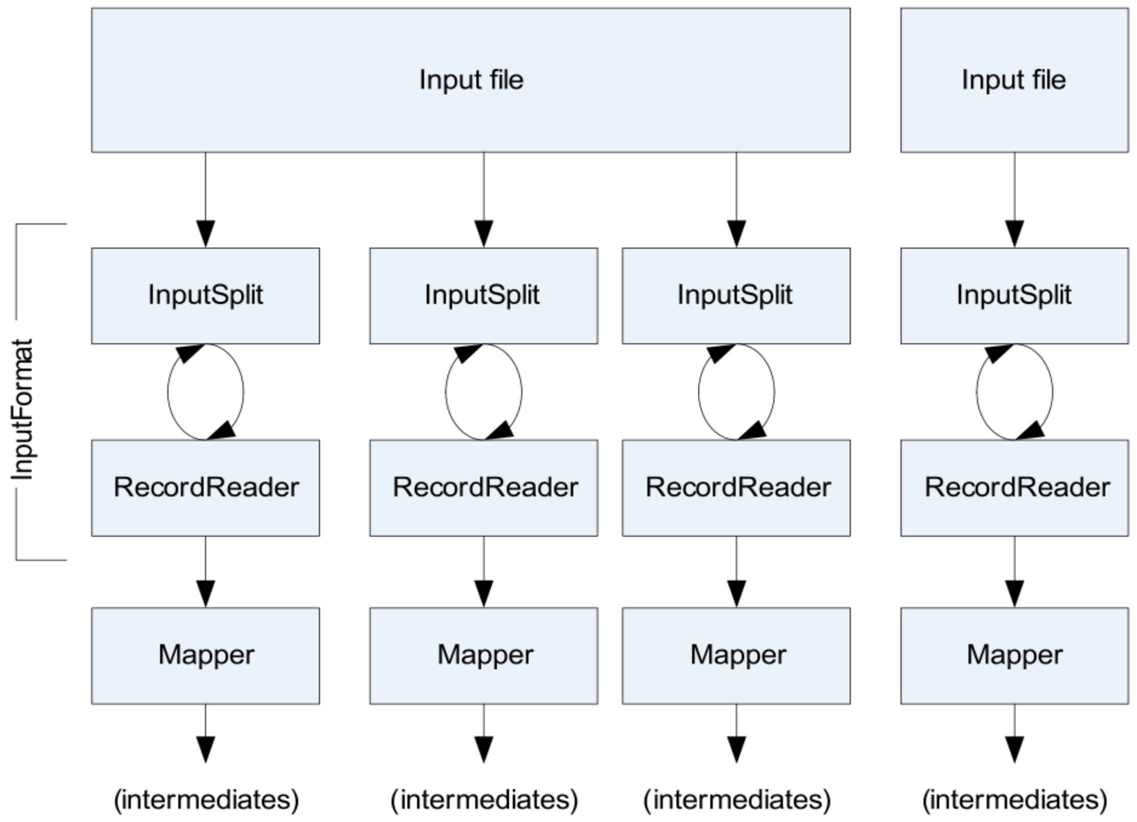
A Sample MapReduce Program: Introduction

- **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

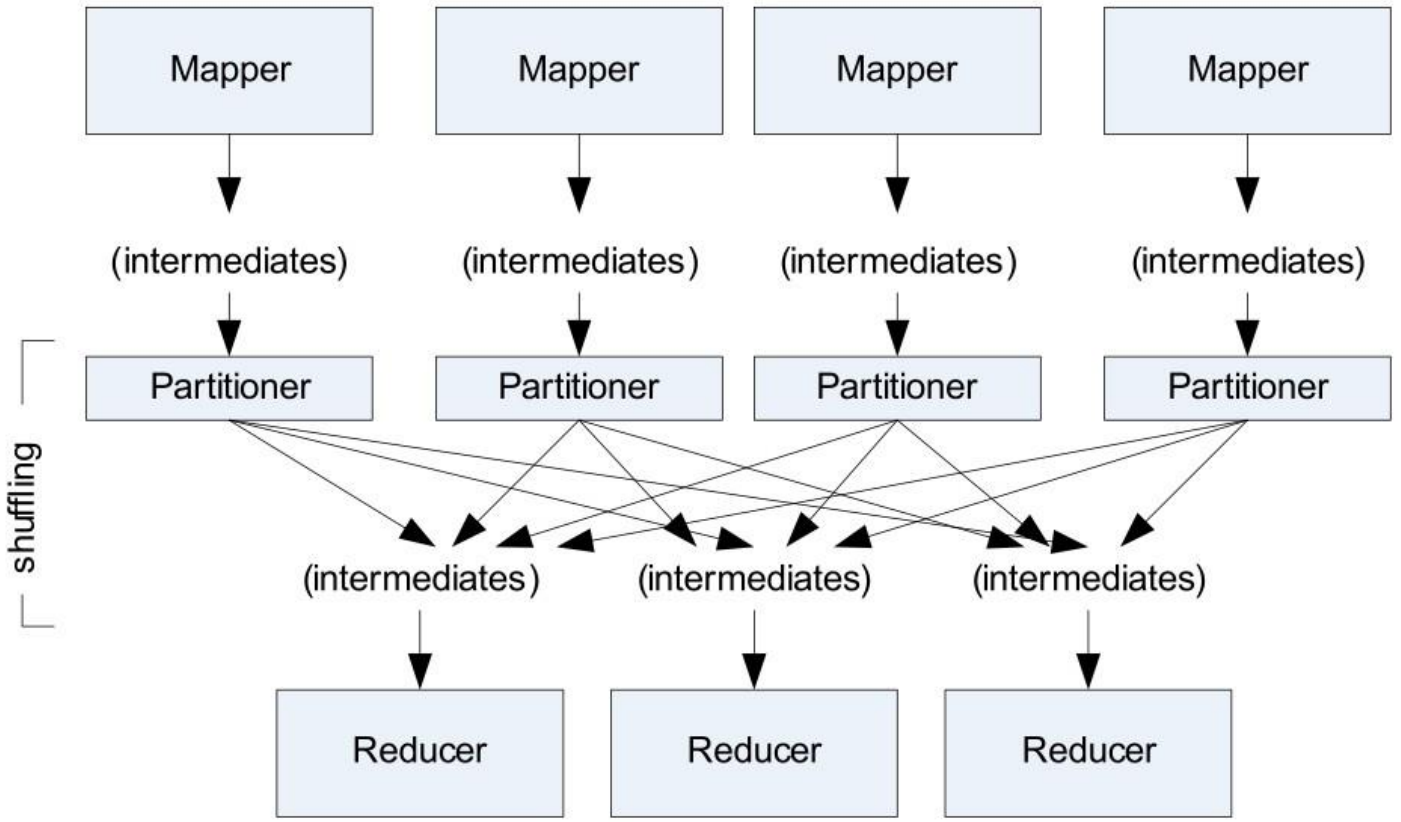
The MapReduce Flow: Introduction

- **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

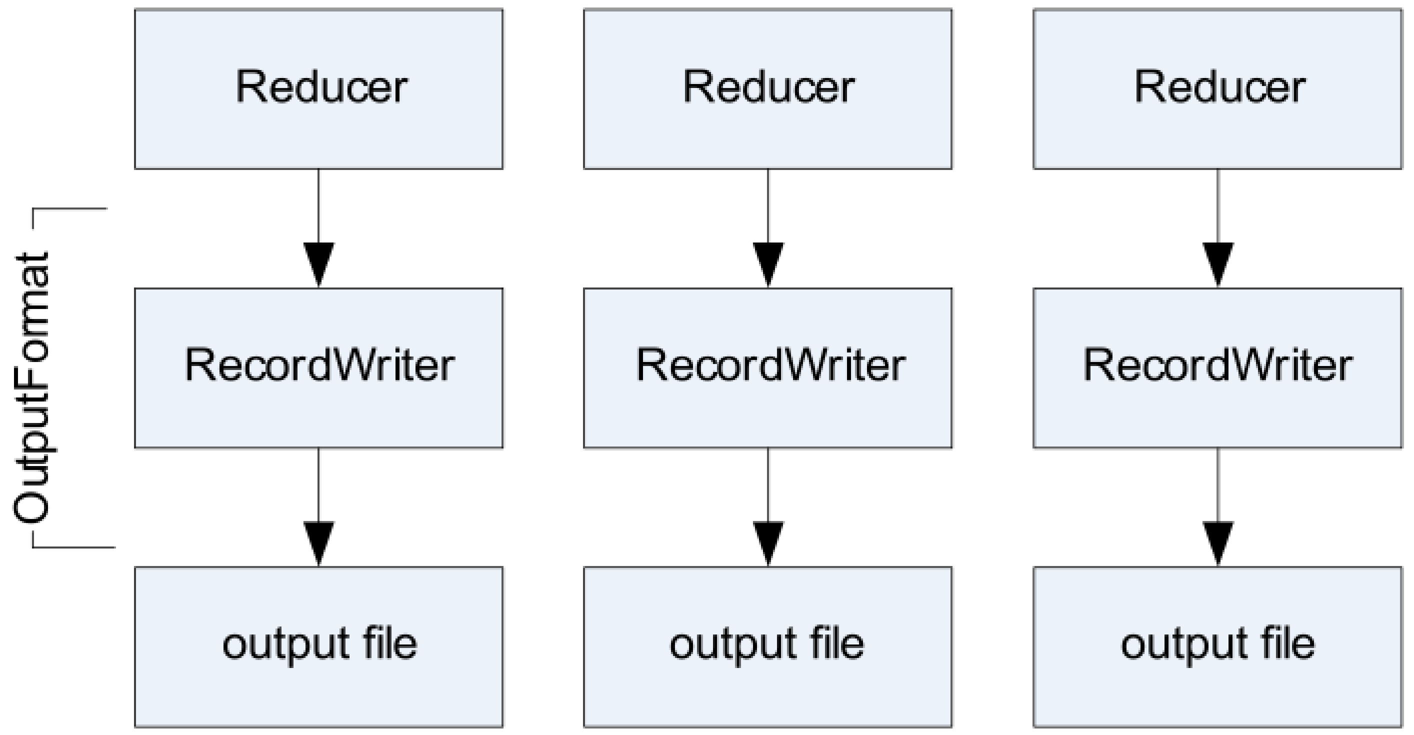
The MapReduce Flow: The Mapper



The MapReduce Flow: Shuffle and Sort



The MapReduce Flow: Reducers to Outputs



Writing a MapReduce Program

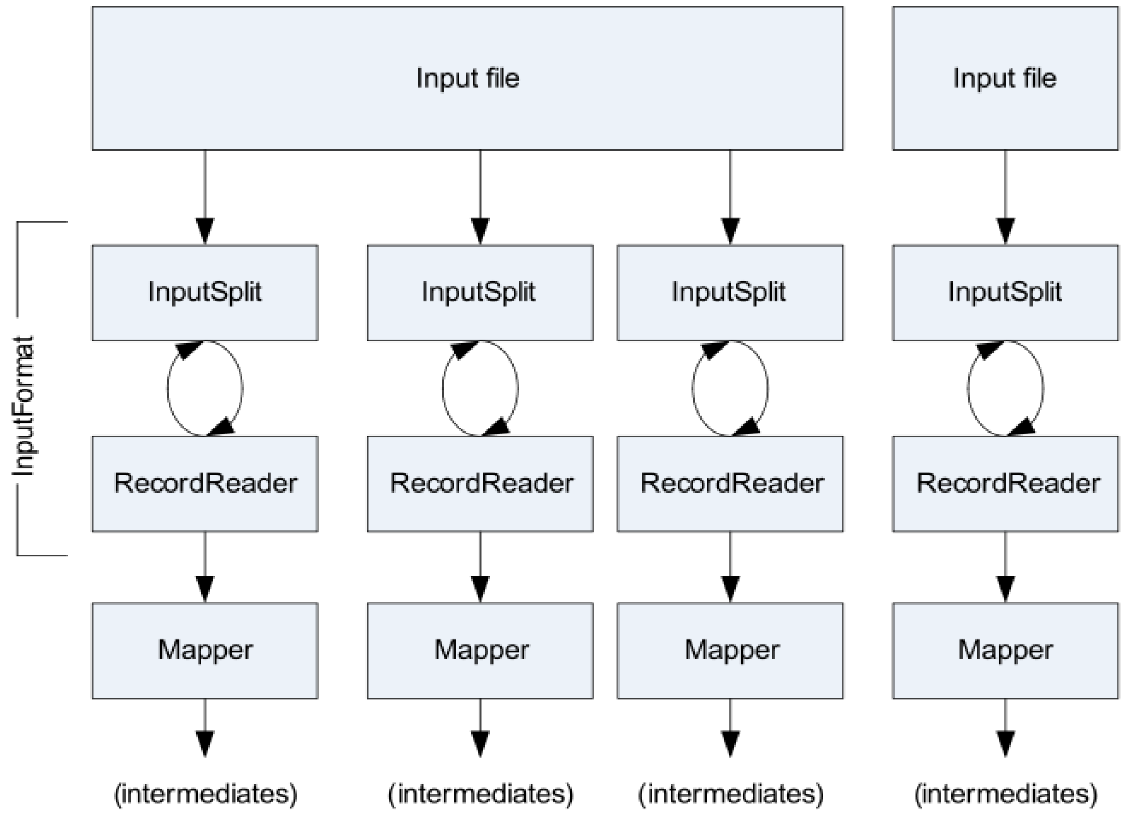
Our MapReduce Program: WordCount

- **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 `WritableComparable`s can be compared against each other to determine their ‘order’
  - Keys must be `WritableComparable`s 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 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 WordCount 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(), WordCount.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 WordCount(),
      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 c
  public

    if (a
      Sys


      Too
      ret
    }
    JobCo
    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);
```

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 WordCount 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(), WordCount.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 WordCount(), args);
    System.exit(exitCode);
  }
}
```

The Driver Class: Using ToolRunner

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

    if

    }
     o
    Jo
    c
    F
     i
     i
    F
    c
     o
     o
    c
    conf.setMapOutputValueClass(IntWritable.class);

    c
    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 WordCount(), args);
    System.exit(exitCode);
  }
}
```

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.)

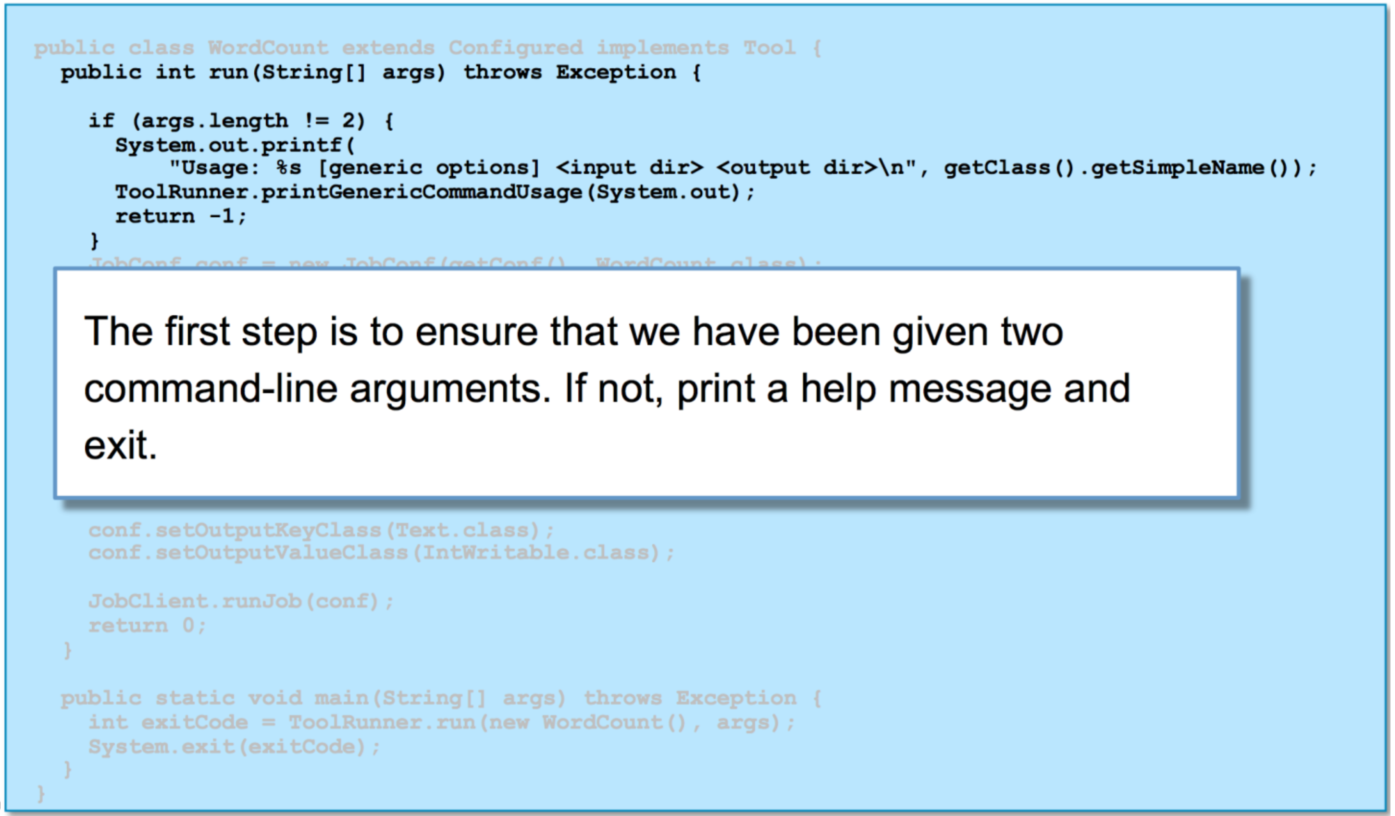
The Driver Class: Using ToolRunner (cont’d)

```
public class WordCount 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(), WordCount.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 WordCount(), args);
    System.exit(exitCode);
  }
}
```

Sanity Checking The Job’s Invocation

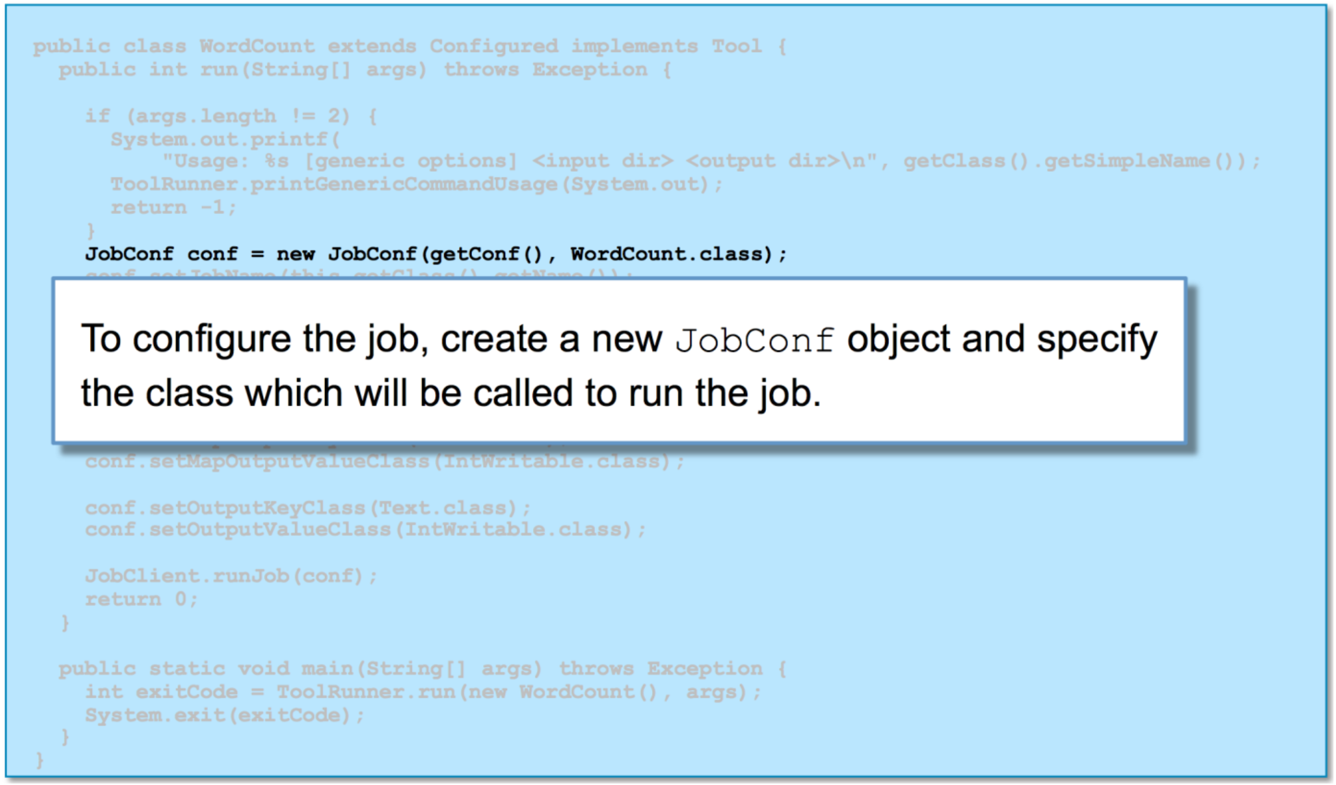


```
public class WordCount extends Configured implements Tool {
  public int run(String[] args) throws Exception {

    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(), WordCount.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 WordCount(), args);
    System.exit(exitCode);
  }
}
```

Configuring The Job With `JobConf`



```
public class WordCount 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(), WordCount.class);
```

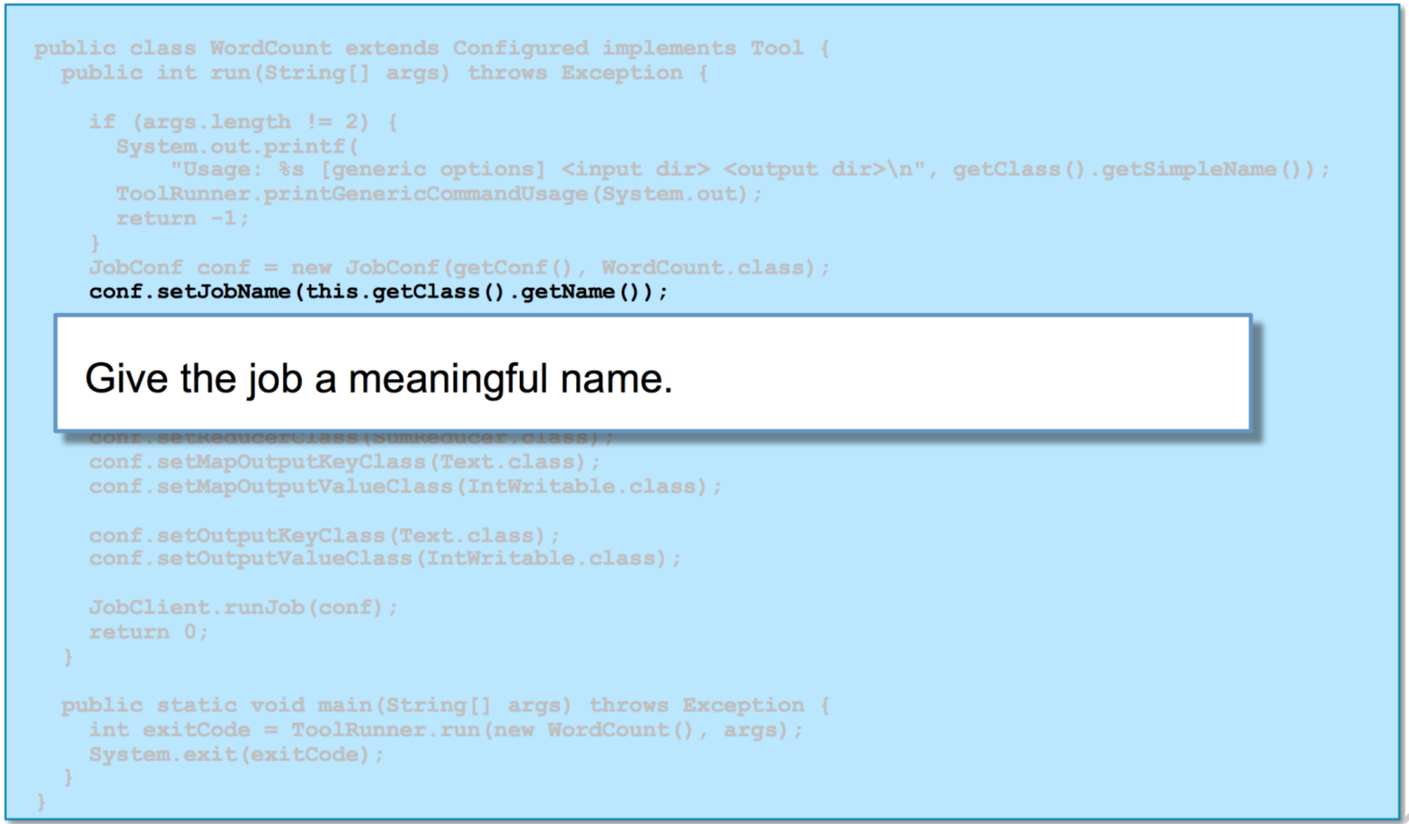
To configure the job, create a new `JobConf` object and specify the class which will be called to run the job.

```
    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 WordCount(), 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 WordCount 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(), WordCount.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 WordCount(), args);
    System.exit(exitCode);
  }
}
```

Specifying Input and Output Directories

```
public class WordCount 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(), WordCount.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 WordCount(), 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)
```

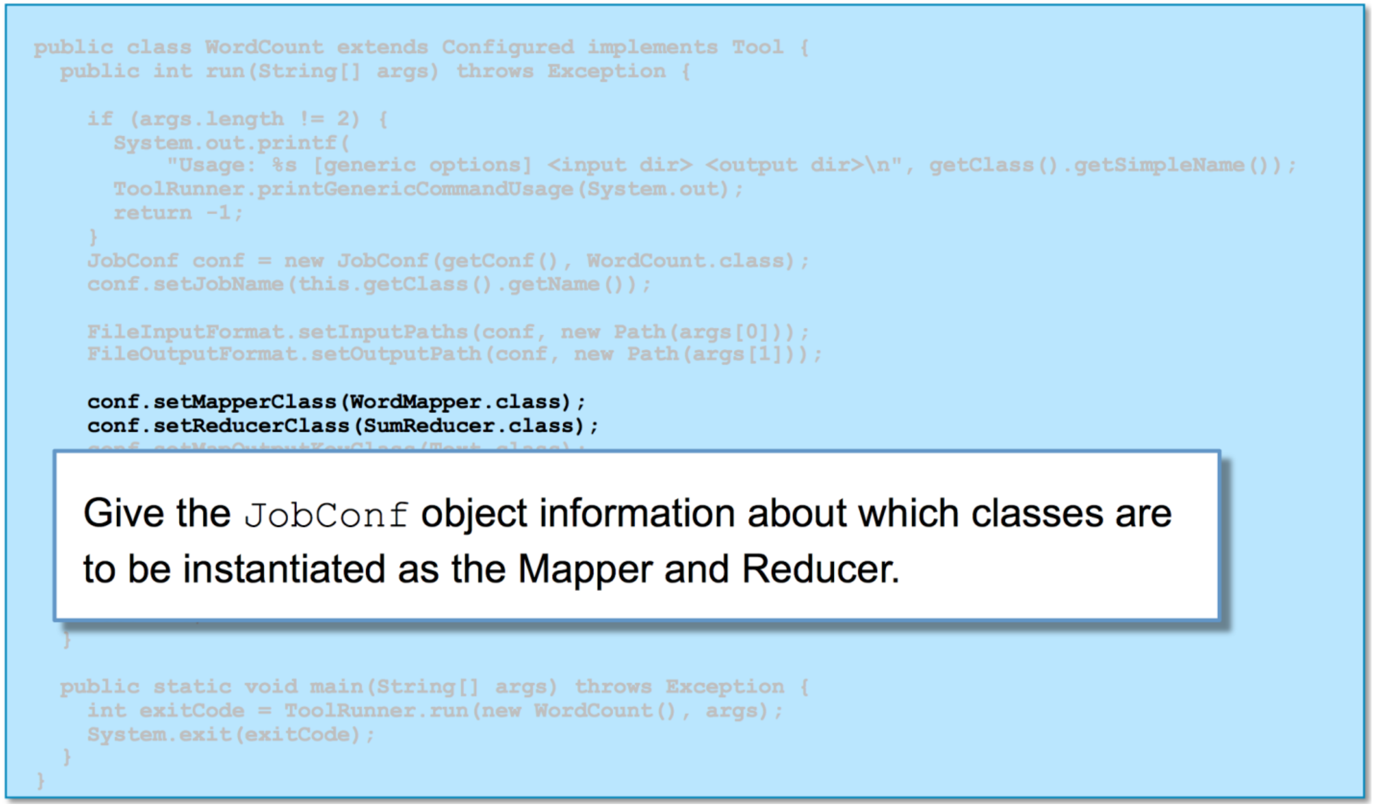
Determining Which Files To Read

- **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 WordCount 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(), WordCount.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);
```

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

```
  }

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

Specify The Intermediate Data Types

```
public class WordCount 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(), WordCount.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 WordCount(), args);
    System.exit(exitCode);
  }
}
```

Specify The Final Output Data Types

```
public class WordCount 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(), WordCount.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 WordCount(), args);
    System.exit(exitCode);
  }
}
```

Running The Job

```
public class WordCount 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(), WordCount.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);
```

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

```
    JobClient.runJob(conf);
    return 0;
  }

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

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 WordCount 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(), WordCount.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 WordCount(), args);
    System.exit(exitCode);
  }
}
```

Writing a MapReduce Program

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 cla
    Mapper

  public v
      Outp
      thro
    String
    for (S
      if (
        output.collect(new Text(word), 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> {

   publi                                                   orter)
       (
       t
     Str
     for
       i

       }
     }
   }
}
```

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();
    fo



    }
  }
}
```

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+")) {
                                                                 ));
   }
}
```

`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 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 clas
    Reducer

  public vo
      Outpu
      throw

    int wor
    while (
      IntWr
      wordC
    }
    output.
  }
}
```

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> {
   publ                                               porter)

     int
     wh

     }
     out
   }
}
```

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 `WritableComparable`s, the values are `Writable`s.

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
        wordC
        while
          (va
      IntWrit
      wordCou
  } }
}   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 Streaming API: Motivation

- **Many organizations have developers skilled in languages other than Java, such as**
  - Ruby
  - Python
  - Perl
- **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**
  - For example: `awk`, `grep`, `sed`, or `wc`

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

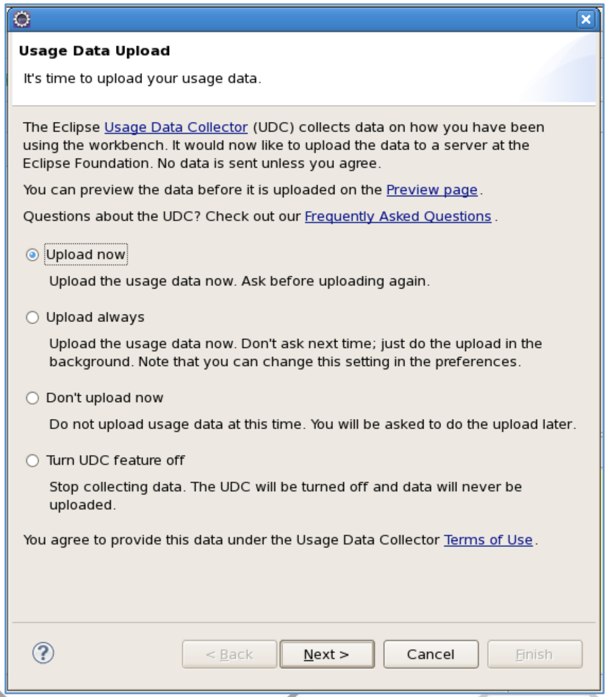
Conclusion

Integrated Development Environments

- **There are many Integrated Development Environments (IDEs) available**
- **Eclipse is one such IDE**
  - Open source
  - Very popular among 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**



Usage Data Upload

It's time to upload your usage data.

The Eclipse Usage Data Collector (UDC) collects data on how you have been using the workbench. It would now like to upload the data to a server at the Eclipse Foundation. No data is sent unless you agree.

You can preview the data before it is uploaded on the Preview page.

Questions about the UDC? Check out our Frequently Asked Questions.

[✓ Upload now]

Upload the usage data now. Ask before uploading again.

Upload always

Upload the usage data now. Don't ask next time; just do the upload in the background. Note that you can change this setting in the preferences.

Don't upload now

Do not upload usage data at this time. You will be asked to do the upload later.

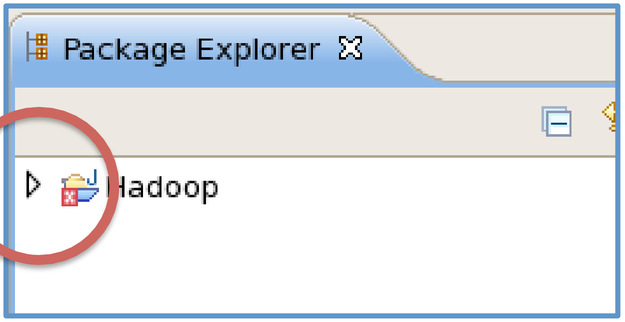
Turn UDC feature off

Stop collecting data. The UDC will be turned off and data will never be uploaded.

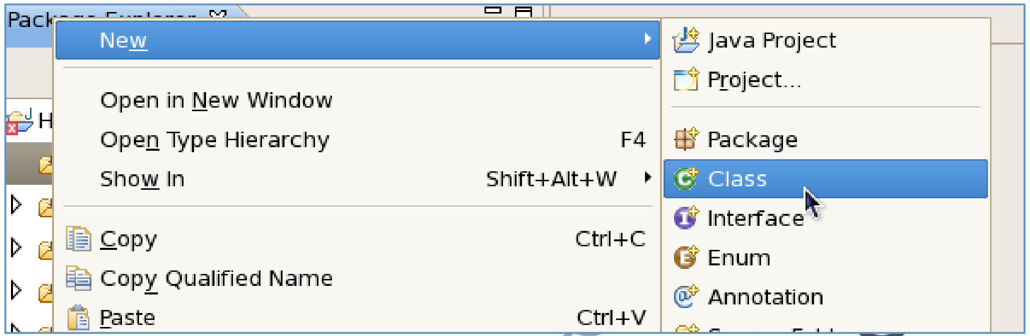
You agree to provide this data under the Usage Data Collector Terms of Use.

Using Eclipse (cont'd)

- Expand the 'Hadoop' project

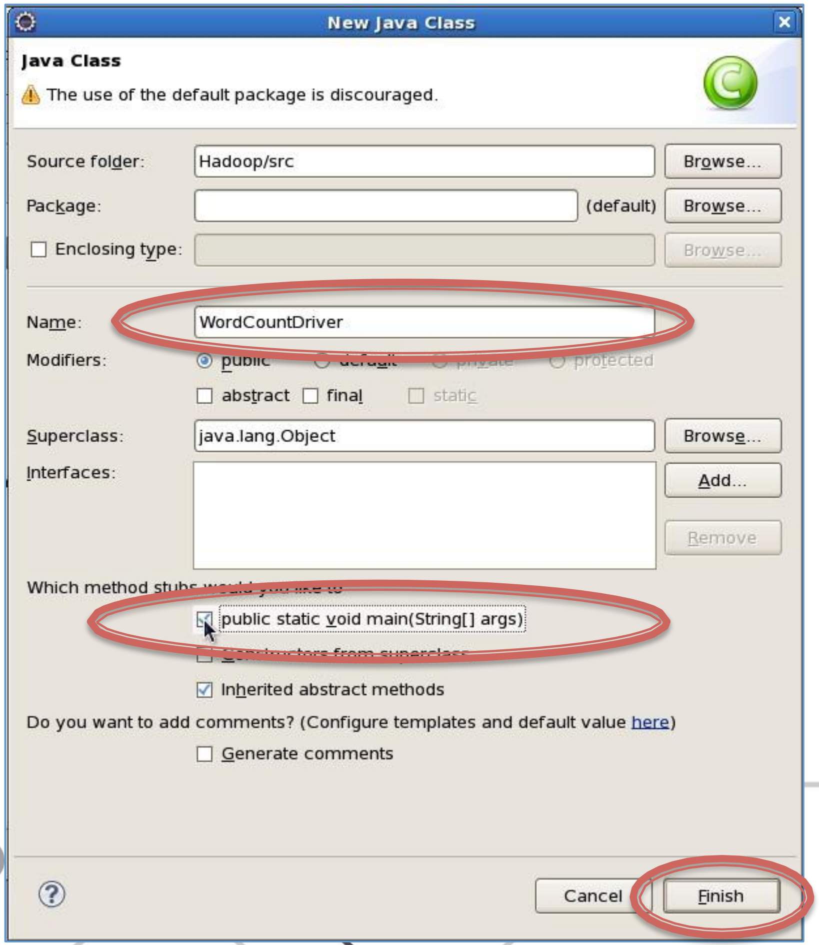


- Right-click on '`src`', and choose New ➔ Class



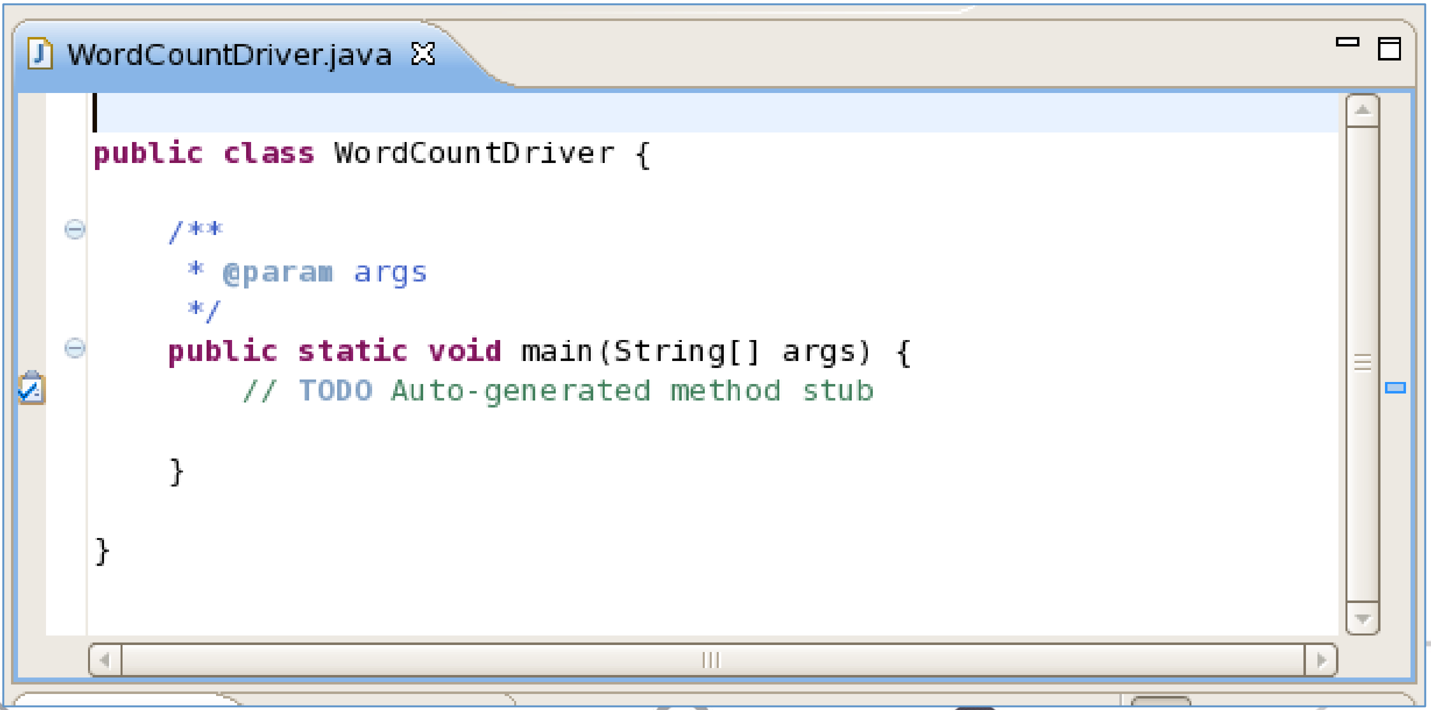
Using Eclipse (cont'd)

- **Enter a class name, check the box to generate the main method stub, and then click Finish**


New Java Class
Java Class
The use of the default package is discouraged.
Source folder: Hadoop/src
Browse...
Package: (default)
Browse...
Enclosing type:
Browse...
Name: WordCountDriver
Modifiers: public default private protected
abstract final static
Superclass: java.lang.Object
Browse...
Interfaces:
Add...
Remove
Which method stubs would you like to
public static void main(String[] args)
constructors from superclass
Inherited abstract methods
Do you want to add comments? (Configure templates and default value here)
Generate comments
Cancel
Finish

Using Eclipse (cont’d)

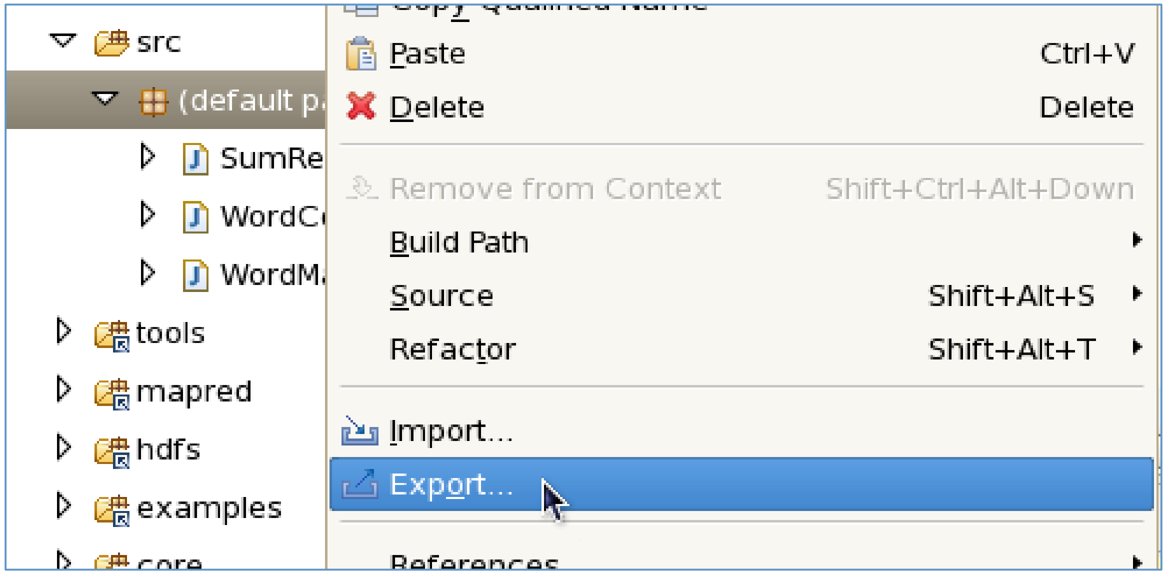
- **You can now edit your class**



```
public class WordCountDriver {

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

    }

}
```

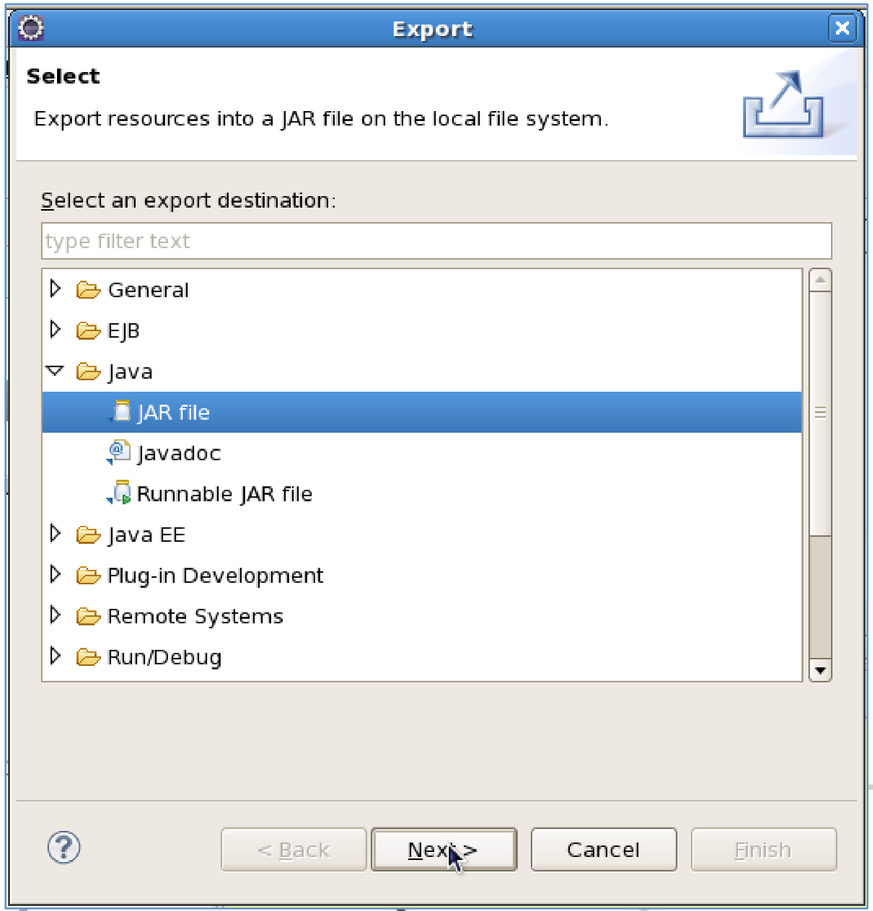
Using Eclipse (cont'd)

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



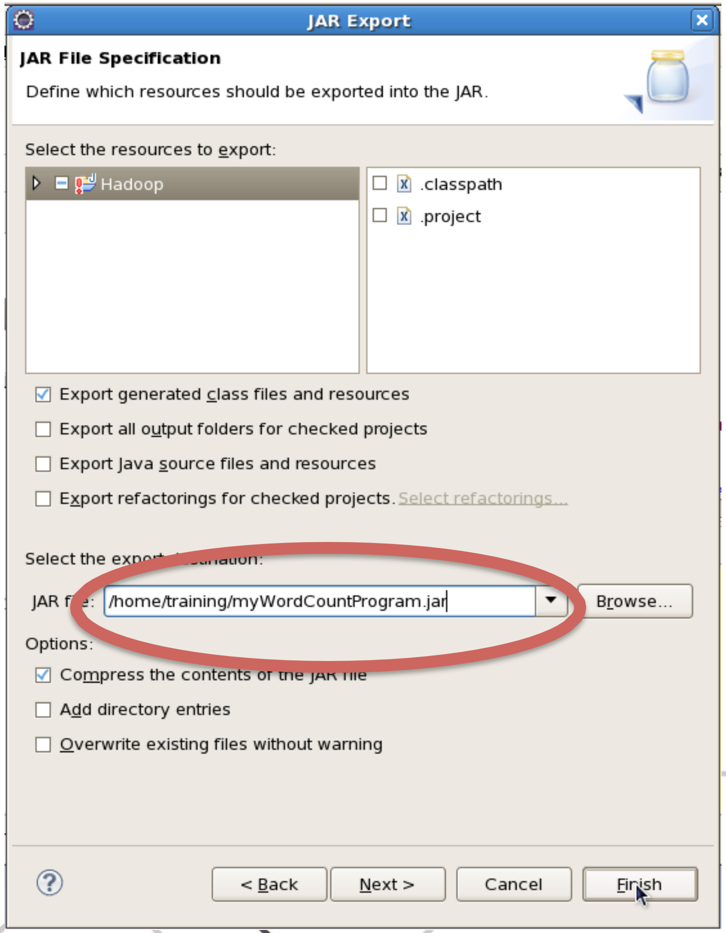
Using Eclipse (cont’d)

- **Expand ‘Java’, select the ‘JAR file’ item, and then click Next.**



Using Eclipse (cont'd)

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



☑ Export generated class files and resources
☐ Export all output folders for checked projects
☐ Export Java source files and resources
☐ Export refactorings for checked projects.

JAR file: /home/training/myWordCountProgram.jar

☑ Compress the contents of the JAR file
☐ Add directory entries
☐ Overwrite existing files without warning

Writing a MapReduce Program

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 Hands-On Exercise Instructions

Writing a MapReduce Program

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
`import org.apache.hadoop.mapred.*`	`import org.apache.hadoop.mapreduce.*`

Driver code:
`JobConf conf = new JobConf(conf,`
`Driver.class);`
`conf.setSomeProperty(...);`
`...`
`JobClient.runJob(conf);`
	Driver code:
	`Configuration conf = new Configuration();`
	`Job job = new Job(conf);`
	`job.setJarByClass(Driver.class);`
	`job.setSomeProperty(...);`
	`...`
	`job.waitForCompletion(true);`

Mapper:
`public class MyMapper extends MapReduceBase`
`implements`
`Mapper {`
`public void map(Keytype k, Valuetype v,`
`OutputCollector o, Reporter`
`r) {`
`...`
`o.collect(key, val);`
`}`
`}`
	Mapper:
	`public class MyMapper extends Mapper {`
	`public void map(Keytype k, Valuetype`
	`v,`
	`Context c) {`
	`...`
	`c.write(key, val);`
	`}`
	`}`

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

Old API

Reducer:

```
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);
      }
   }

}
```

New API

Reducer:

```
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);
      }
   }

}
```

Old API	New API
`configure(JobConf job)` **(See later)**	`setup(Context c)`
`close()` **(See later)**	`cleanup(Context c)`

Writing a MapReduce Program

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