



## Developing Applications for Apache Hadoop

Sarah Sproehnle, [sarah@cloudera.com](mailto:sarah@cloudera.com)



Copyright © 2010-2012 Cloudera. All rights reserved. Not to be reproduced without prior written consent.

01-1

### Agenda

---

- **The core concepts of Hadoop (HDFS and MapReduce)**
- **Writing a MapReduce program**
- **Overview of Hive and Pig**
- **Common algorithms that are used with Hadoop**
- **Graph processing in MapReduce**
- **Overview of HBase**



Copyright © 2010-2012 Cloudera. All rights reserved. Not to be reproduced without prior written consent.

01-2

## Hadoop: Basic Concepts

---

In this section you will learn

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

## 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!

## HDFS Basic Concepts

---

- **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*

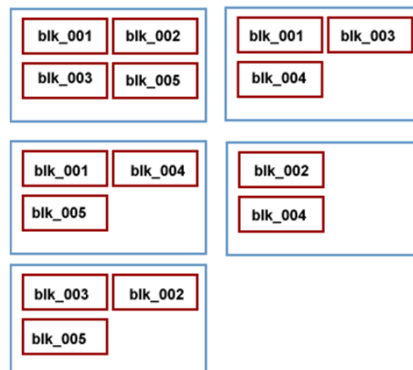
## How Files Are Stored: Example

- **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

### 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!

## 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
```

## What Is MapReduce?

---

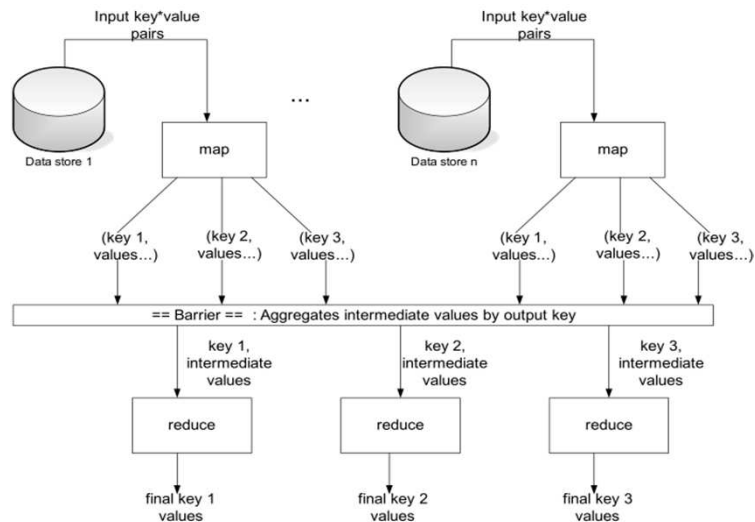
- **MapReduce is a method for distributing a task across multiple nodes**
- **Each node processes data stored on that node**
  - Where possible
- **Consists of two phases:**
  - Map
  - Reduce

## 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)
```

```
('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: 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

## Installing A Hadoop Cluster

---

- **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**
  - Instantiates and monitors individual Map and Reduce tasks

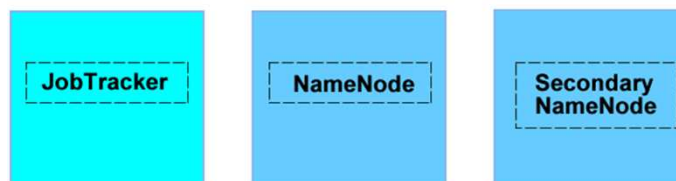


## The Five Hadoop Daemons (cont'd)

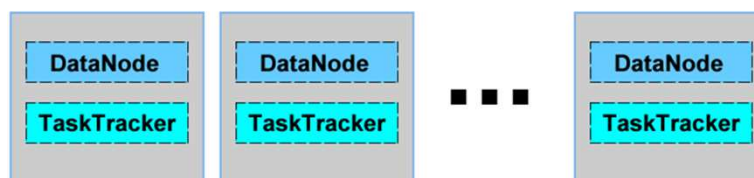
- Each daemon runs in its own Java Virtual Machine (JVM)
- No node on a real cluster will run all five daemons
  - Although this is technically possible
- We can consider nodes to be in two different categories:
  - Master Nodes
    - Run the NameNode, Secondary NameNode, JobTracker daemons
    - Only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

### 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



## Writing a MapReduce Program

## Writing a MapReduce Program

---

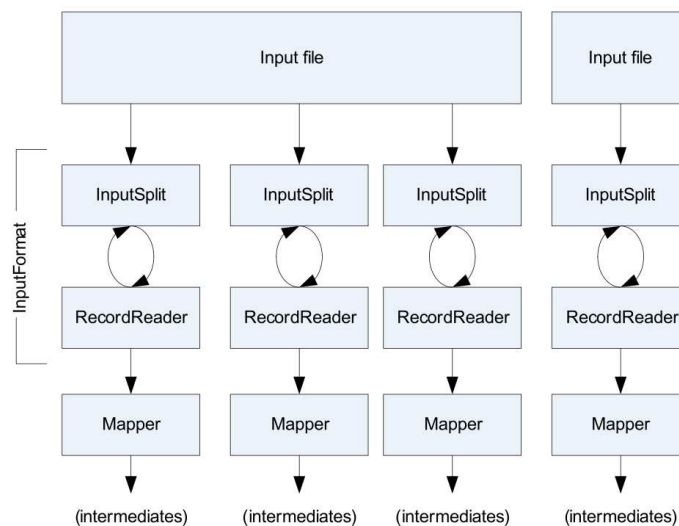
### In this section you will learn

- 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

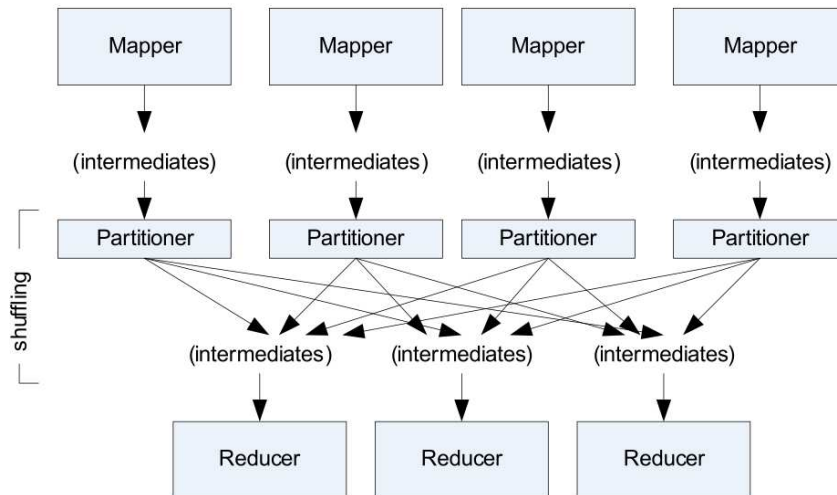
## The MapReduce Flow: Introduction

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

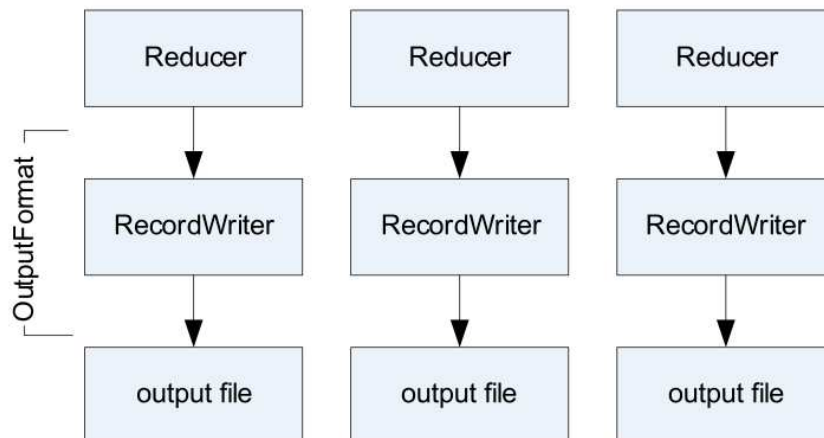
## The MapReduce Flow: The Mapper



## The MapReduce Flow: Shuffle and Sort



## The MapReduce Flow: Reducers to Outputs



## Our MapReduce Program: WordCount

---

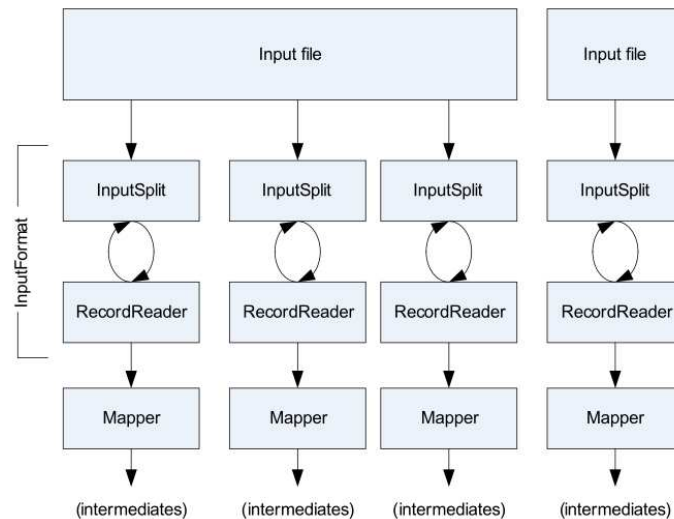
- **To investigate the API, we will dissect the WordCount program**
- **This consists of three portions**
  - The driver code
    - Code that runs on the client to configure and submit the job
  - The Mapper
  - The Reducer
- **Before we look at the code, we need to cover some basic Hadoop API concepts**

## 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  $\backslash n$ -terminated line of a file as a value
  - Key is the byte offset within the file of that line
- **KeyValueTextInputFormat**
  - Maps  $\backslash n$ -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

## 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 class WordCount {
    public static void main(String[] args) throws Exception {
        if (args.length != 2) {
            System.out.println(
                "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);
    }
}
```

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
        )
        JobConf
        conf.setInputPathClass(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);
    }
}
```

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

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.

## 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);

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

## 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);
        conf.setJobName(this.getClass().getName());

        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);
    }
}
```

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

## 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());

        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);
    }
}
```

Give the job a meaningful name.

## 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]));

        JobClient.runJob(conf);
        return 0;
    }

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

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

## 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);
        conf.setMapOutputValueClass(IntWritable.class);

    }

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

    }

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

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

## 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);

        int exitCode = ToolRunner.run(new WordCount(), args);
        System.exit(exitCode);
    }
}
```

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

## 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);

        JobClient.runJob(conf);
        return 0;
    }

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

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,
        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));
            }
        }
    }
}
```

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

## 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
    public void
        Output
        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> {
```

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.collect(key, new IntWritable(wordCount));
    }
}
```

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));
    }
}
```

## 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`





## Hive and Pig

## 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
- **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

## 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 and so on)
  - 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**

## 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 bagname;
```

- 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 grpd 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:**
  - Cloudera Training for Apache Hive and Pig



## Common MapReduce Algorithms

## Common MapReduce Algorithms

---

- **Some typical MapReduce algorithms, including**
  - Sorting
  - Searching
  - Indexing
  - Collaborative filtering, clustering, classification
  - Term Frequency – Inverse Document Frequency

## 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

## Sorting

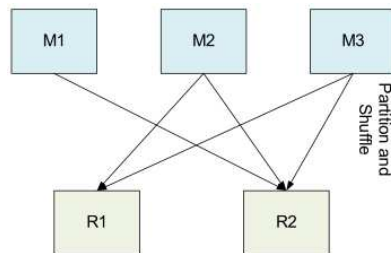
---

- **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  
 $(k, v) \rightarrow (v, \_)$
  - Reducer is the identity function  
 $(k, \_) \rightarrow (k, \_)$

## 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

## Indexing

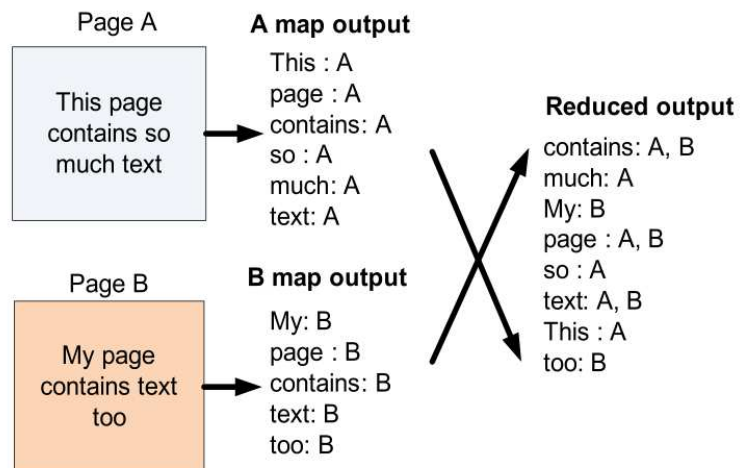
---

- **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



## 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

## 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
    - 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 × 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: ((term, docid),  $tf$ )
  - Output: (term, (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 ((term, docid), ( $tf$ ,  $n$ ))

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

---

- **Mapper**
  - Input:  $((\text{term}, \text{docid}), (tf, n))$
  - Assume  $N$  is known (easy to find)
  - Output  $((\text{term}, \text{docid}), \text{TF} \times \text{IDF})$
- **Reducer**
  - The identity function

## Computing TF-IDF: Working At Scale

---

- **Job 2: We need to buffer  $(\text{docid}, tf)$  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



## Graph Manipulation in MapReduce



## 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
  - Weighted
  - Unweighted
  - 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

## 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

## Graphs and MapReduce

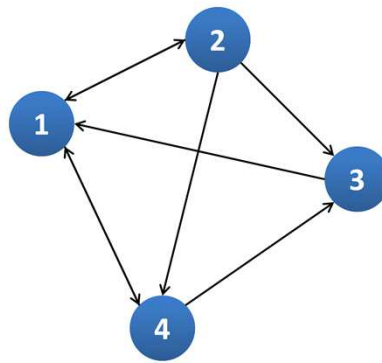
---

- **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?

## Representing Graphs

---

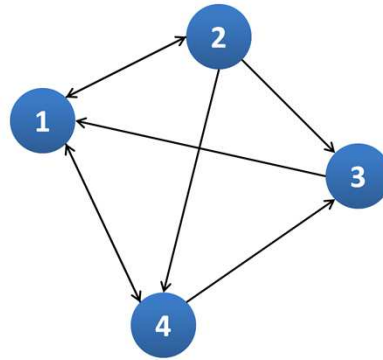
- **Imagine we want to represent this simple graph:**
- **Two approaches:**
  - Adjacency matrices
  - Adjacency lists



## 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

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



$v_1$ :  $v_2, v_4$   
 $v_2$ :  $v_1, v_3, v_4$   
 $v_3$ :  $v_1$   
 $v_4$ :  $v_1, v_3$



## 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 field of type `long` usually suffices

- **Typical encoding format (Writable)**

- `long`: vertex id of the source
- `int`: number of outgoing edges
- Sequence of `longs`: destination vertices

$V_1: V_2, V_4$		1: [2] 2, 4
$V_2: V_1, V_3, V_4$		2: [3] 1, 3, 4
$V_3: V_1$		3: [1] 1
$V_4: V_1, V_3$		4: [2] 1, 3

## Example: 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

---

- **Characters 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 ← N.DISTANCE
4:     EMIT(nid n, N)                                ▷ Pass along graph structure
5:     for all nodeid m ∈ N.ADJACENCYLIST do
6:       EMIT(nid m, d + 1)                          ▷ Emit distances to reachable nodes

1: class REDUCER
2:   method REDUCE(nid m, [d1, d2, ...])
3:     dmin ← ∞
4:     M ← ∅
5:     for all d ∈ counts [d1, d2, ...] do
6:       if ISNODE(d) then
7:         M ← d                                     ▷ Recover graph structure
8:         else if d < dmin then                  ▷ Look for shorter distance
9:           dmin ← d
10:    M.DISTANCE ← dmin                             ▷ Update shortest distance
11:    EMIT(nid m, node M)
```

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



## 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



## Introduction to Apache HBase

### What is HBase?

---

- **HBase is . . .**
  - Open-Source
  - Sparse
  - Multidimensional
  - Persistent
  - Distributed
  - Sorted Map
  - Runs on top of HDFS
  - Modeled after Google's BigTable

## HBase is a Sorted Distributed Map . . .

---

- **HBase is a Map at its core**
  - Much like a PHP/Perl associative array, or JavaScript Object
  - The map is indexed by a row key, column key, and a timestamp; each value in the map is an uninterpreted array of bytes.
- **Sorted**
  - Key/Value pairs are kept in lexicographic sorted order
    - Very important when scanning large amounts of data
  - Ensures like information is located in close proximity
  - Impacts row/key design considerations (discussed later)
- **Distributed**
  - Built upon a distributed filesystem (HDFS)
  - Underlying file storage abstracts away complexities of distributed computing

## . . . that is also Sparse, Multidimensional and Persistent

---

- **Sparse**
  - A given row can have any number of columns
  - May be gaps between keys
- **Multidimensional**
  - All data is versioned using a timestamp (or configurable integer)
  - Data is not updated, instead it is added with a new version number
- **Persistent**
  - Data “persists” after the program that created it is finished

## Hbase is NOT a Traditional RDBMS

	RDBMS	HBase
Data layout	Row or column-oriented	Column Family-oriented
Transactions	Yes	Single row only
Query language	SQL	get/put/scan
Security	Authentication/Authorization	TBD
Indexes	Yes	Row-key only
Max data size	TBs	PB+
Read/write throughput limits	1000s queries/second	Millions of queries/second

## Introduction to HBase

### What is HBase?

#### ➔ HDFS and HBase

### Hands-on-Exercise: Using HDFS

### HBase Usage Scenarios

### Conclusion

## HBase is built on Hadoop

---

- **Hadoop provides:**
  - Fault tolerance
  - Scalability
  - Batch processing with MapReduce
- **HBase provides:**
  - Random reads and writes
  - High throughput
  - Caching

## Usage Scenarios for HBase

---

- **Lots of data**
  - 100s of Gigabytes up to Petabytes
- **High write throughput**
  - 1000s/second per node
- **Scalable cache capacity**
  - Adding nodes adds to available cache
- **Data layout**
  - Excels at key lookup
  - No penalty for sparse columns

## When To Use HBase

---

- **Use HBase if...**
  - You need random write, random read, or both (but not neither)
  - You need to do many thousands of operations per second on multiple TB of data
  - Your access patterns are well-known and simple
- **Don't use HBase if...**
  - You only append to your dataset, and tend to read the whole thing
  - You primarily do ad-hoc analytics (ill-defined access patterns)
  - Your data easily fits on one beefy node

## Overview of the data model

---

- **Tables are made of rows and columns**
- **Every row has a row key (analogous to a primary key)**
  - Rows are stored sorted by row key for fast lookups
- **All columns in HBase belong to a particular column family**
- **A table may have one or more column families**
  - Common to have a small number of column families
  - Column families should rarely change
  - A column family can have any number of columns
- **Table cells are versioned, uninterpreted arrays of bytes**

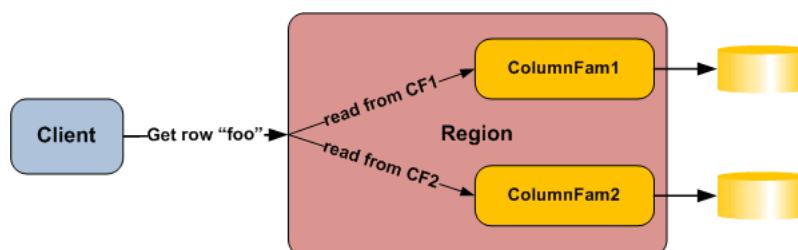
## “User” Table Conceptual View

- Column consist of a column family prefix + qualifier
- Separate column families are useful for
  - Data that is not frequently accessed together
  - Data that uses different column family options (discussed later)
    - e.g., compression

Row Key	Column Family "metadata"	Column Family "photo"
user1	fname: Doug, lname: Cutting	<hadoop.gif>
user2	fname: Tom, lname: White	<tom.job>
user3	fname: Todd, lname: Lipcon	

## Designing tables

- Same row key → same node
  - Lookups by row key talk to a single region server
- Same column family → same set of physical files
  - Retrieving data from a column family is sequential I/O



## Get Operation

---

- **Used to look up a single rowkey**
  - Takes the row key as a byte array
- **HTable's get method retrieves a Result object**
- **Extract the data from the Result object**
  - `getValue` returns the cell value for Column Family + Column
  - Value is returned as a byte array
  - `Bytes.toString(value)` convertst the byte array to a String
- **If rowkey is not found theResult will have size 0**
  - `R.size()` or `r.isEmpty()` to verivy if rows were returned

```
Get g = new Get(Bytes.toBytes("rowkey"));
Result row = table.get(g);
byte[] value = row.getValue(Bytes.toBytes("colfam"), Bytes.toBytes("column"));
```

## Put Operation

---

- **Used to add a row to a table**
  - Takes the row key as a byte array
  - Use `Bytes.toBytes(String)` to convert a string to byte array
- **Add method takes various forms:**
  - `add(byte[] colfam, byte[] columns, byte[] value)`
  - `add(byte[] colfam, byte[] column, long ts, byte[] value)`
  - `add(byte[] colfam_and_column, byte[] value)`

```
Put p = new Put(Bytes.toBytes("rowkey"));
p.add(Bytes.toBytes("colfam"), Bytes.toBytes("column"),
                                           Bytes.toBytes("value"));
table.put(p);
```



## Scan Operation

---

- **Used to scan all the rows in a table**
  - Instantiate a Scan object
  - Invoke the getScanner method
  - Returns a Result Scanner
    - Iterator containing Result objects
- **When iterating over a scanner, one row is retrieved at a time**
  - Caching rows can make scanning faster but requires more RAM
  - `hbase.client.scanner.caching` and `setScannerCaching(int)`

```
Scan s = new Scan();
ResultScanner scanner = table.getScanner(s);
for (Result rr : scanner) {
    ...
}
```

## Non-Java APIs

---

- **A proxy server can marshal non-Java requests to HBase**
- **Advantage**
  - Applications can be written in C++, Python, PHP, Ruby, etc
- **Disadvantages**
  - Requires running and monitoring an extra proxy daemon
  - Slightly slower than native Java access

## REST (Representational State Transfer)

---

- **Stargate**

- a service which exposes HBase objects via REST requests
- Returns JSON or XML
- Start the daemon:

```
$ bin/hbase-daemon.sh start \
org.apache.hadoop.hbase.stargate.Main -p <port>
```
- GET a row:

```
$ curl -H "Accept: text/xml"
http://host:port/table/rowkey/column:qualifier
```
- PUT a row:

```
$ curl -H "Content-Type: text/xml" --data '...'
http://host:port/table/rowkey/column:qualifier
```

## Apache Thrift

---

- **Exposes services (such as HBase) to applications written in other languages such as Python, Perl, C++ and Ruby**
- **Start the daemon:**

```
$ bin/hbase-daemon.sh start thrift
```
- **Generate the language-specific classes**

```
$ thrift --gen py Hbase.thrift
```
- **Look at Hbase.thrift for the set of methods available such as `getTableNames`, `createTable`, `getRow`, `mutateRow`**

## HBase and MapReduce

---

- **HBase tables as input or output of a MapReduce job**
- **TableInputFormat**
  - Splits HBase tables on their Regions
  - Uses a scanner to read each split
- **TableOutputFormat**
  - Uses Put to write records to a table
- **HFileOutputFormat**
  - Writes data in the native HFile format (for bulk load)

# cloudera

Questions?

Visit us @ booth 700

Sarah Sproehnle  
sarah@cloudera.com