Map-Reduce using Hadoop

Marissa Hollingsworth and *Amit Jain*[*]

Department of Computer Science
College of Engineering
Boise State University

[*]Chief Science Officer
Boise Computing Partners

Big Data, Big Disks, Cheap Computers

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

Big Data, Big Disks, Cheap Computers

- *"In pioneer days they used oxen for heavy pulling, and when one ox couldn't budge a log, they didn't try to grow a larger ox. We shouldn't be trying for bigger computers, but for more systems of computers."* Rear Admiral Grace Hopper.
- *"More data usually beats better algorithms."* Anand Rajaraman.

Big Data, Big Disks, Cheap Computers

- *"In pioneer days they used oxen for heavy pulling, and when one ox couldn't budge a log, they didn't try to grow a larger ox. We shouldn't be trying for bigger computers, but for more systems of computers."* Rear Admiral Grace Hopper.
- *"More data usually beats better algorithms."* Anand Rajaraman.
- *"The good news is that Big Data is here. The bad news is that we are struggling to store and analyze it."* Tom White.

Introduction

- MapReduce is a programming model and an associated implementation for processing and generating large data sets.

Introduction

- MapReduce is a programming model and an associated implementation for processing and generating large data sets.
- Users specify a map function that processes a key/value pair to generate a set of intermediate key/value pairs, and a reduce function that merges all intermediate values associated with the same intermediate key.

Introduction

- MapReduce is a programming model and an associated implementation for processing and generating large data sets.
- Users specify a map function that processes a key/value pair to generate a set of intermediate key/value pairs, and a reduce function that merges all intermediate values associated with the same intermediate key.
- Allows programmers without any experience with parallel and distributed systems to easily utilize the resources of a large distributed system.

Introduced by Google. Used internally for all major computations on over 100k servers. Yahoo is running on over 36,000 Linux servers with 5 PBs of data. Facebook has over 2 PB of data and growing at 10TB per day. Amazon is leasing servers to run map reduce computations (EC2 and S3 programs). Microsoft is developing Dryad (a super set of Map-Reduce).

MapReduce Programming Model

- Map method, written by the user, takes an input pair and produces a set of intermediate key/value pairs. The MapReduce library groups together all intermediate values associated with the same intermediate key I and passes them to the Reduce function.

MapReduce Programming Model

- Map method, written by the user, takes an input pair and produces a set of intermediate key/value pairs. The MapReduce library groups together all intermediate values associated with the same intermediate key I and passes them to the Reduce function.
- Reduce method, also written by the user, accepts an intermediate key I and a set of values for that key. It merges together these values to form a possibly smaller set of values. Typically just zero or one output value is produced per Reduce invocation. The intermediate values are supplied to the users reduce function via an iterator. This allows us to handle lists of values that are too large to fit in memory.

MapReduce Programming Model

- Map method, written by the user, takes an input pair and produces a set of intermediate key/value pairs. The MapReduce library groups together all intermediate values associated with the same intermediate key I and passes them to the Reduce function.
- Reduce method, also written by the user, accepts an intermediate key I and a set of values for that key. It merges together these values to form a possibly smaller set of values. Typically just zero or one output value is produced per Reduce invocation. The intermediate values are supplied to the users reduce function via an iterator. This allows us to handle lists of values that are too large to fit in memory.
- MapReduce Specification Object. Contains names of input/output files and optional tuning parameters. The user then invokes the MapReduce function, passing it the specification object. The users code is linked together with the MapReduce library.

A MapReduce Example

Consider the problem of counting the number of occurrences of each word in a large collection of documents.

```
map(String key, String value):
  // key: document name
  // value: document contents
  for each word w in value:
     EmitIntermediate(w, "1");

reduce(String key, Iterator values):
  // key: a word
  // values: a list of counts
  int result = 0;
  for each v in values:
     result += ParseInt(v);
  Emit(key, AsString(result));
```

MapReduce Examples

- **Distributed Grep**: The map function emits a line if it matches a supplied pattern. The reduce function is an identity function that just copies the supplied intermediate data to the output.

MapReduce Examples

- **Distributed Grep**: The map function emits a line if it matches a supplied pattern. The reduce function is an identity function that just copies the supplied intermediate data to the output.
- **Count of URL Access Frequency**: The map function processes logs of web page requests and outputs <URL, 1>. The reduce function adds together all values for the same URL and emits a <URL, total count> pair.

MapReduce Examples

- **Distributed Grep**: The map function emits a line if it matches a supplied pattern. The reduce function is an identity function that just copies the supplied intermediate data to the output.
- **Count of URL Access Frequency**: The map function processes logs of web page requests and outputs <URL, 1>. The reduce function adds together all values for the same URL and emits a <URL, total count> pair.
- **Reverse Web-Link Graph**: The map function outputs <target, source> pairs for each link to a target URL found in a page named source. The reduce function concatenates the list of all source URLs associated with a given target URL and emits the pair: <target, list(source)>

MapReduce Examples (contd.)

- **Inverted Index**: The map function parses each document, and emits a sequence of <word, document ID> pairs. The reduce function accepts all pairs for a given word, sorts the corresponding document IDs and emits a <word, list(document ID)> pair. The set of all output pairs forms a simple inverted index. It is easy to augment this computation to keep track of word positions.

MapReduce Examples (contd.)

- **Inverted Index**: The map function parses each document, and emits a sequence of <word, document ID> pairs. The reduce function accepts all pairs for a given word, sorts the corresponding document IDs and emits a <word, list(document ID)> pair. The set of all output pairs forms a simple inverted index. It is easy to augment this computation to keep track of word positions.
- **Distributed Sort**: The map function extracts the key from each record, and emits a <key, record> pair. The reduce function emits all pairs unchanged. This computation depends on the partitioning and ordering facilities that are provided in a MapReduce implementation.

MapReduce Examples (contd.)

- **Capitalization Probability**: In a collection of text documents, find the percentage capitalization for each letter of the alphabet.

MapReduce Examples (contd.)

- **Capitalization Probability**: In a collection of text documents, find the percentage capitalization for each letter of the alphabet.

  'a' $\rightarrow$ ("A", 0)
  'A' $\rightarrow$ ("A", 1)

MapReduce Examples (contd.)

- **Capitalization Probability**: In a collection of text documents, find the percentage capitalization for each letter of the alphabet.

  'a' → ("A", 0)
  'A' → ("A", 1)

- **Track Statistics at Last.fm**: We will cover this in detail as our case study in the second half of the talk.

MapReduce versus Relational Databases

	Traditional RDBMS	*MapReduce*
Data size	GB-TBs	TBs-PBs
Access	Interactive and batch	Batch
Updates	Read/write many times	Write once, read many times
Structure	Static Schema	Dynamic Schema
Integrity	High	Low
Scaling	Nonlinear	Linear

Hadoop is a software platform that lets one easily write and run parallel-distributed applications that process vast amounts of data. Features of Hadoop:

- Scalable: Hadoop can reliably store and process Petabytes.
- Economical: It distributes the data and processing across clusters of commonly available computers. These clusters can number into the thousands of nodes.
- Efficient: By distributing the data, Hadoop can process it in parallel on the nodes where the data is located. This makes it extremely efficient.
- Reliable: Hadoop automatically maintains multiple copies of data and automatically redeploys computing tasks based on failures.

Hadoop subprojects

The core subprojects:

- **Hadoop Common**: The common utilities that support the other Hadoop subprojects.
- **HDFS**: A distributed file system that provides high throughput access to application data.
- **MapReduce**: A software framework for distributed processing of large data sets on compute clusters.

Hadoop subprojects

The core subprojects:

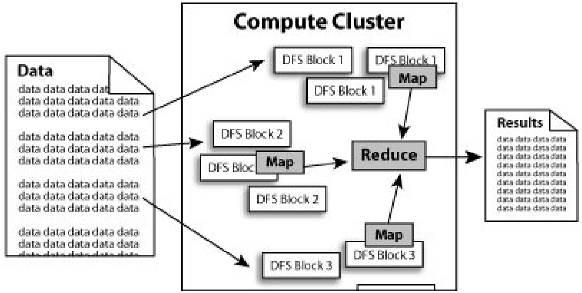
- **Hadoop Common**: The common utilities that support the other Hadoop subprojects.
- **HDFS**: A distributed file system that provides high throughput access to application data.
- **MapReduce**: A software framework for distributed processing of large data sets on compute clusters.

Other Hadoop-related projects at Apache include:

- *Avro*: A data serialization system.
- *Chukwa*: A data collection system for managing large distributed systems.
- *HBase*: A scalable, distributed database.
- *Hive*: A data warehouse infrastructure with SQL ad hoc querying.
- *Mahout*: A scalable machine learning and data mining library.
- *Pig*: A high-level parallel data-flow language and execution framework.
- *ZooKeeper*: A high-performance distributed coordination service.
- *Sqoop*: A tool for efficiently moving data between relational databases and HDFS.

Hadoop Implementation

Hadoop implements MapReduce, using the Hadoop Distributed File System (HDFS) (see figure below.) MapReduce divides applications into many small blocks of work. HDFS creates multiple replicas of data blocks for reliability, placing them on compute nodes around the cluster. MapReduce can then process the data where it is located.



Hadoop History

- Hadoop is sub-project of the Apache foundation. Receives sponsorship from Google, Yahoo, Microsoft, HP and others.
- Hadoop is written in Java. Hadoop MapReduce programs can be written in Java as well as several other languages.
- Hadoop grew out of the Nutch Web Crawler project.

Hadoop History

- Hadoop is sub-project of the Apache foundation. Receives sponsorship from Google, Yahoo, Microsoft, HP and others.
- Hadoop is written in Java. Hadoop MapReduce programs can be written in Java as well as several other languages.
- Hadoop grew out of the Nutch Web Crawler project.
- Hadoop programs can be developed using Eclipse/NetBeans on Linux or MS Windows. To use MS Windows requires Cygwin package. MS Windows is recommended for development but not as a full production Hadoop cluster.

Hadoop History

- Hadoop is sub-project of the Apache foundation. Receives sponsorship from Google, Yahoo, Microsoft, HP and others.
- Hadoop is written in Java. Hadoop MapReduce programs can be written in Java as well as several other languages.
- Hadoop grew out of the Nutch Web Crawler project.
- Hadoop programs can be developed using Eclipse/NetBeans on Linux or MS Windows. To use MS Windows requires Cygwin package. MS Windows is recommended for development but not as a full production Hadoop cluster.
- Used by Yahoo, Facebook, Amazon, RackSpace, Twitter, eBay, LinkedIn, New York Times, Last.fm, E-Harmony, Microsoft (via acquisition of Powerset) and many others. Several cloud consulting companies like Cloudera.
- New York Times article on Hadoop. `http://www.nytimes.com/2009/03/17/technology/business-computing/17cloud.html`

Hadoop Map-Reduce Inputs and Outputs

- The Map/Reduce framework operates exclusively on $<key, value>$ pairs, that is, the framework views the input to the job as a set of $<key, value>$ pairs and produces a set of $<key, value>$ pairs as the output of the job, conceivably of different types.

Hadoop Map-Reduce Inputs and Outputs

- The Map/Reduce framework operates exclusively on $< key, value >$ pairs, that is, the framework views the input to the job as a set of $< key, value >$ pairs and produces a set of $< key, value >$ pairs as the output of the job, conceivably of different types.
- The key and value classes have to be serializable by the framework and hence need to implement the Writable interface. Additionally, the key classes have to implement the WritableComparable interface to facilitate sorting by the framework.

Hadoop Map-Reduce Inputs and Outputs

- The Map/Reduce framework operates exclusively on $<key, value>$ pairs, that is, the framework views the input to the job as a set of $<key, value>$ pairs and produces a set of $<key, value>$ pairs as the output of the job, conceivably of different types.
- The key and value classes have to be serializable by the framework and hence need to implement the Writable interface. Additionally, the key classes have to implement the WritableComparable interface to facilitate sorting by the framework.
- The user needs to implement a `Mapper` class as well as a `Reducer` class. Optionally, the user can also write a `Combiner` class.

$$\text{(input)} < k1, v1 > \rightarrow \text{map} \rightarrow < k2, v2 > \rightarrow \text{combine} \rightarrow < k2, v2 >$$

$$\rightarrow \text{reduce} \rightarrow < k3, v3 > \text{(output)}$$

How to write a Hadoop Map class

Subclass from `MapReduceBase` and implement the `Mapper` interface.

```
public class MyMapper<K extends WritableComparable, V extends Writable>
 extends MapReduceBase implements Mapper<K, V, K, V> { ... }
```

The `Mapper` interface provides a single method:

```
public void map(K key, V val, OutputCollector<K, V> output,
Reporter reporter)
```

- `WriteableComparable` key:
- `Writeable` value:
- `OutputCollector` output: this has the collect method to output a <key, value> pair
- `Reporter` reporter: allows the application code to permit alteration of status

The Hadoop system divides the input data into logical "records" and then calls `map()` once for each record. For text files, a record is one line of text. The key then is the byte-offset and the value is a line from the text file. For other input types, it can be defined differently. The main method is responsible for setting output key values and value types.

How to write a Hadoop Reduce class

Subclass from `MapReduceBase` and implement the `Reducer` interface.

```
public class MyReducer<K extends WritableComparable, V extends Writable>
 extends MapReduceBase implements Reducer<K, V, K, V> {...}
```

The `Reducer` interface provides a single method:

```
public void reduce(K key, Iterator<V> values,
OutputCollector<K, V> output, Reporter reporter)
```

- `WriteableComparable` key:
- `Iterator` values:
- `OutputCollector` output:
- `Reporter` reporter:

Given all the values for the key, the Reduce code typically iterates over all the values and either concatenates the values together in some way to make a large summary object, or combines and reduces the values in some way to yield a short summary value.

WordCount example in Java with Hadoop

Problem: To count the number of occurrences of each word in a large collection of documents.

```
/**
 * Counts the words in each line.
 * For each line of input, break the line into words and emit them as (word, 1).
 */
public static class Map extends MapReduceBase
        implements Mapper<LongWritable, Text, Text, IntWritable>
{
    private final static IntWritable one = new IntWritable(1);
    private Text word = new Text();
    public void map(LongWritable key, Text value,
                    OutputCollector<Text, IntWritable> output,
            Reporter reporter) throws IOException
    {
        String line = value.toString();
        StringTokenizer tokenizer = new StringTokenizer(line);
        while (tokenizer.hasMoreTokens()) {
            word.set(tokenizer.nextToken());
            output.collect(word, one);
        }
    }
}
```

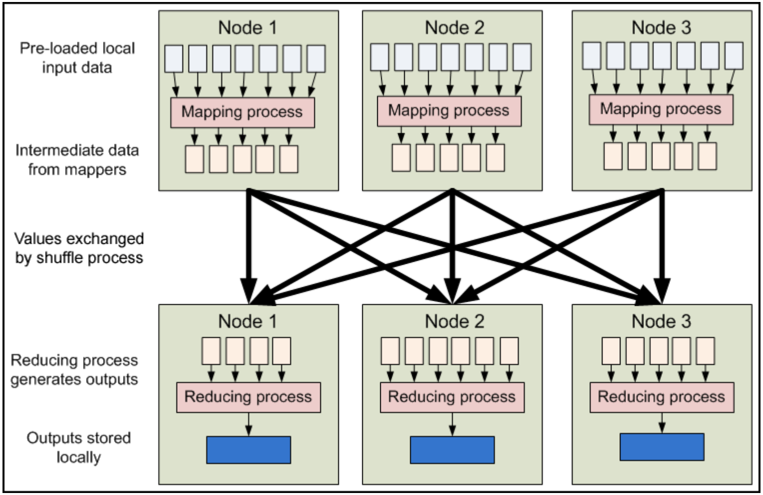
WordCount Example continued

```
/**
 * A reducer class that just emits the sum of the input values.
 */
public static class Reduce 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 sum = 0;
        while (values.hasNext()) {
            sum += values.next().get();
        }
        output.collect(key, new IntWritable(sum));
    }
}
```

WordCount Example continued

```
public static void main(String[] args) throws Exception
{
    if (args.length != 2) {
        printUsage();
        System.exit(1);
    }
    JobConf conf = new JobConf(WordCount.class);
    conf.setJobName("wordcount");

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

    conf.setMapperClass(Map.class);
    conf.setCombinerClass(Reduce.class);
    conf.setReducerClass(Reduce.class);

    conf.setInputFormat(TextInputFormat.class);
    conf.setOutputFormat(TextOutputFormat.class);

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

    JobClient.runJob(conf);
}
```

MapReduce: High-Level Data Flow



Case Analysis Example

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

  private final static IntWritable one = new IntWritable(1);
  private final static IntWritable zero = new IntWritable(0);
  private Text word = new Text();

  public void map(LongWritable key, Text value,
          OutputCollector<Text, IntWritable> output, Reporter reporter)
          throws IOException {
      String line = value.toString();

      for (int i = 0; i < line.length(); i++) {
          if (Character.isLowerCase(line.charAt(i))) {
              word.set(String.valueOf(line.charAt(i)).toUpperCase());
              output.collect(word, zero);
          } else if (Character.isUpperCase(line.charAt(i))) {
              word.set(String.valueOf(line.charAt(i)));
              output.collect(word, one);
          } else {
              word.set("other");
              output.collect(word, one);
          }
      }
    }
```

Case Analysis Example (contd.)

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

    private Text result = new Text();

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

      long total = 0;
      int upper = 0;

      while (values.hasNext()) {
          upper += values.next().get();
          total++;
      }
      result.set(String.format("%16d %16d %16d %16.2f", total, upper,
                                (total - upper),((double)upper / total)));
      output.collect(key, result);
    }
}
```

Case Analysis Example (contd.)

```
public static void main(String[] args) throws Exception {
    if (args.length != 2) {
        printUsage();
        System.exit(1);
    }
    JobConf conf = new JobConf(CaseAnalysis.class);
    conf.setJobName("caseanalysis");

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

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

    conf.setMapperClass(Map.class);
    // conf.setCombinerClass(Reduce.class);
    conf.setReducerClass(Reduce.class);

    conf.setInputFormat(TextInputFormat.class);
    conf.setOutputFormat(TextOutputFormat.class);

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

    JobClient.runJob(conf);
}
```

MapReduce Optimizations

- Overlap of maps, shuffle, and sort
- Mapper locality
  - Schedule mappers close to the data.
- Combiner
  - Mappers may generate duplicate keys
  - Side-effect free reducer can be run on mapper node
  - Minimizes data size before transfer
  - Reducer is still run
- Speculative execution to help with load-balancing
  - Some nodes may be slower
  - Run duplicate task on another node, take first answer as correct and abandon other duplicate tasks
  - Only done as we start getting toward the end of the tasks

Setting up Hadoop

- Download the latest stable version of Hadoop from http://hadoop.apache.org/.
- Unpack the tarball that you downloaded in previous step.
  `tar xzvf hadoop-0.20.2.tar.gz`
- Edit `conf/hadoop-env.sh` and at least set `JAVA_HOME` to point to Java installation folder. You will need Java version 1.6 or higher.
- Now run `bin/hadoop` to test Java setup. You should get output similar to shown below.

```
[amit@kohinoor hadoop-0.20.2]$ bin/hadoop
Usage: hadoop [--config confdir] COMMAND
where COMMAND is one of:
...
```

Setting up Hadoop

- Download the latest stable version of Hadoop from `http://hadoop.apache.org/`.
- Unpack the tarball that you downloaded in previous step. `tar xzvf hadoop-0.20.2.tar.gz`
- Edit `conf/hadoop-env.sh` and at least set `JAVA_HOME` to point to Java installation folder. You will need Java version 1.6 or higher.
- Now run `bin/hadoop` to test Java setup. You should get output similar to shown below.

```
[amit@kohinoor hadoop-0.20.2]$ bin/hadoop
Usage: hadoop [--config confdir] COMMAND
where COMMAND is one of:
...
```

- We can use hadoop in three modes:
  - *Standalone mode*: Everything runs in a single process. Useful for debugging.
  - *Pseudo-distributed mode*: Multiple processes as in distributed mode but they all run on one host. Again useful for debugging distributed mode of operation before unleashing it on a real cluster.
  - *Distributed mode*: "The real thing!" Multiple processes running on multiple machines.

Standalone and Pseudo-Distributed Mode

- Hadoop comes ready to run in standalone mode out of the box. Try the following with the wordcount jar file to test Hadoop.

```
mkdir input
cp conf/*.xml input
bin/hadoop jar wordcount.jar input output
cat output/*
```

Standalone and Pseudo-Distributed Mode

- Hadoop comes ready to run in standalone mode out of the box. Try the following with the wordcount jar file to test Hadoop.

  ```
  mkdir input
  cp conf/*.xml input
  bin/hadoop jar wordcount.jar input output
  cat output/*
  ```

- To run in pseudo-distributed mode, we need to specify the following:
  - The NameNode (Distributed Filesystem master) host and port. This is specified with the configuration property `fs.default.name`.
  - The JobTracker (MapReduce master) host and port. This is specified with the configuration property `mapred.job.tracker`.
  - The Replication Factor should be set to 1 with the property `dfs.replication`.
  - A slaves file that lists the names of all the hosts in the cluster. The default slaves file is `conf/slaves` it should contain just one hostname: `localhost`.

Standalone and Pseudo-Distributed Mode

- Hadoop comes ready to run in standalone mode out of the box. Try the following with the wordcount jar file to test Hadoop.

  ```
  mkdir input
  cp conf/*.xml input
  bin/hadoop jar wordcount.jar input output
  cat output/*
  ```

- To run in pseudo-distributed mode, we need to specify the following:
  - The NameNode (Distributed Filesystem master) host and port. This is specified with the configuration property `fs.default.name`.
  - The JobTracker (MapReduce master) host and port. This is specified with the configuration property `mapred.job.tracker`.
  - The Replication Factor should be set to 1 with the property `dfs.replication`.
  - A slaves file that lists the names of all the hosts in the cluster. The default slaves file is `conf/slaves` it should contain just one hostname: `localhost`.
- Make sure that you can run `ssh localhost` command without a password. If you cannot, then set it up as follows:

  ```
  ssh-keygen -t dsa -P '' -f ~/.ssh/id_dsa
  cat ~/.ssh/id_dsa.pub >> ~/.ssh/authorized_keys
  ```

Pseudo-Distributed Mode

- To run everything in one node, edit three config files so that they contain the configuration shown below:

```
--> conf/core-site.xml
<configuration>
<property> <name>fs.default.name</name>
           <value>hdfs://localhost:9000</value> </property>
</configuration>
```

Pseudo-Distributed Mode

- To run everything in one node, edit three config files so that they contain the configuration shown below:

```
--> conf/core-site.xml
<configuration>
<property> <name>fs.default.name</name>
           <value>hdfs://localhost:9000</value> </property>
</configuration>

--> conf/hdfs-site.xml
<configuration>
<property> <name>dfs.replication</name>
           <value>1</value> </property>
</configuration>
```

Pseudo-Distributed Mode

- To run everything in one node, edit three config files so that they contain the configuration shown below:

```
--> conf/core-site.xml
<configuration>
<property> <name>fs.default.name</name>
           <value>hdfs://localhost:9000</value> </property>
</configuration>

--> conf/hdfs-site.xml
<configuration>
<property> <name>dfs.replication</name>
           <value>1</value> </property>
</configuration>

--> conf/mapred-site.xml
<configuration>
<property> <name>mapred.job.tracker</name>
           <value>localhost:9001</value> </property>
</configuration>
```

Pseudo-Distributed Mode

- Now create a new Hadoop distributed file system (HDFS) with the command:

  ```
  bin/hadoop namenode -format
  ```

Pseudo-Distributed Mode

- Now create a new Hadoop distributed file system (HDFS) with the command:

  ```
  bin/hadoop namenode -format
  ```

- Start the Hadoop daemons.

  ```
  bin/start-all.sh
  ```

Pseudo-Distributed Mode

- Now create a new Hadoop distributed file system (HDFS) with the command:
  ```
  bin/hadoop namenode -format
  ```
- Start the Hadoop daemons.
  ```
  bin/start-all.sh
  ```
- Put input files into the Distributed file system.
  ```
  bin/hadoop dfs -put input input
  ```

Pseudo-Distributed Mode

- Now create a new Hadoop distributed file system (HDFS) with the command:
  ```
  bin/hadoop namenode -format
  ```
- Start the Hadoop daemons.
  ```
  bin/start-all.sh
  ```
- Put input files into the Distributed file system.
  ```
  bin/hadoop dfs -put input input
  ```
- Now run the distributed job and copy output back to local file system.
  ```
  bin/hadoop jar wordcount.jar input output
  bin/hadoop dfs -get output output
  ```

Pseudo-Distributed Mode

- Now create a new Hadoop distributed file system (HDFS) with the command:

  ```
  bin/hadoop namenode -format
  ```
- Start the Hadoop daemons.

  ```
  bin/start-all.sh
  ```
- Put input files into the Distributed file system.

  ```
  bin/hadoop dfs -put input input
  ```
- Now run the distributed job and copy output back to local file system.

  ```
  bin/hadoop jar wordcount.jar input output
  bin/hadoop dfs -get output output
  ```
- Point your web browser to `localhost:50030` to watch the Hadoop job tracker and to `localhost:50070` to be able to browse the Hadoop DFS and get its status.

Pseudo-Distributed Mode

- Now create a new Hadoop distributed file system (HDFS) with the command:
  `bin/hadoop namenode -format`
- Start the Hadoop daemons.
  `bin/start-all.sh`
- Put input files into the Distributed file system.
  `bin/hadoop dfs -put input input`
- Now run the distributed job and copy output back to local file system.
  ```
  bin/hadoop jar wordcount.jar input output
  bin/hadoop dfs -get output output
  ```
- Point your web browser to `localhost:50030` to watch the Hadoop job tracker and to `localhost:50070` to be able to browse the Hadoop DFS and get its status.
- When you are done, stop the Hadoop daemons as follows.
  `bin/stop-all.sh`

last.fm A Case Study

- Internet radio and community-driven music discovery service founded in 2002.

last.fm A Case Study

- Internet radio and community-driven music discovery service founded in 2002.
- Users transmit information to Last.fm servers indicating which songs they are listening to.

last.fm A Case Study

- Internet radio and community-driven music discovery service founded in 2002.
- Users transmit information to Last.fm servers indicating which songs they are listening to.
- The received data is processed and stored so the user can access it in the form of charts and so Last.fm can make intelligent taste and compatibility decisions for generating recommendations and radio stations.

last.fm A Case Study

- Internet radio and community-driven music discovery service founded in 2002.
- Users transmit information to Last.fm servers indicating which songs they are listening to.
- The received data is processed and stored so the user can access it in the form of charts and so Last.fm can make intelligent taste and compatibility decisions for generating recommendations and radio stations.
- The track listening data is obtained from one of two sources:
  - The listen is a scrobble when a user plays a track of his or her own and sends the information to Last.fm through a client application.
  - The listen is a radio listen when the user tunes into a Last.fm radio station and streams a song.
  - Last.fm applications allow users to love, skip or ban each track they listen to. This track listening data is also transmitted to the server.

Big Data at Last.fm

- Over 40M unique visitors and 500M pageviews each month

Big Data at Last.fm

- Over 40M unique visitors and 500M pageviews each month
- **Scrobble stats:**
  - Up to 800 scrobbles per second
  - More than 40 million scrobbles per day
  - Over 40 billion scrobbles so far

Big Data at Last.fm

- Over 40M unique visitors and 500M pageviews each month
- **Scrobble stats:**
  - Up to 800 scrobbles per second
  - More than 40 million scrobbles per day
  - Over 40 billion scrobbles so far
- **Radio stats:**
  - Over 10 million streaming hours per month
  - Over 400 thousand unique stations per day
- Each scrobble and radio listen generates at least one log line.

Big Data at Last.fm

- Over 40M unique visitors and 500M pageviews each month
- **Scrobble stats:**
  - Up to 800 scrobbles per second
  - More than 40 million scrobbles per day
  - Over 40 billion scrobbles so far
- **Radio stats:**
  - Over 10 million streaming hours per month
  - Over 400 thousand unique stations per day
- Each scrobble and radio listen generates at least one log line.
- In other words...**lots of data!!**

hadoop @ last.fm

- Started using Hadoop in 2006 as users grew from thousands to millions.

hadoop @ last.fm

- Started using Hadoop in 2006 as users grew from thousands to millions.
- Hundreds of daily, monthly, and weekly jobs including,
  - Site stats and metrics
  - Chart generation (track statistics)
  - Metadata corrections (e.g. misspellings of artists)
  - Indexing for search and combining/formatting data for recommendations
  - Data insights, evaluations, reporting

hadoop @ last.fm

- Started using Hadoop in 2006 as users grew from thousands to millions.
- Hundreds of daily, monthly, and weekly jobs including,
  - Site stats and metrics
  - Chart generation (track statistics)
  - Metadata corrections (e.g. misspellings of artists)
  - Indexing for search and combining/formatting data for recommendations
  - Data insights, evaluations, reporting
- This case study will focus on the chart generation (Track Statistics) job, which was the first Hadoop implementation at Last.fm.

hadoop @ last.fm

- Last.FM's *"Herd of Elephants"*
    - 44 Nodes
    - 8 cores per node
    - 16GB memory per node
    - 4 disks of 1 TB spinning at 7200 RPM per node

hadoop @ last.fm

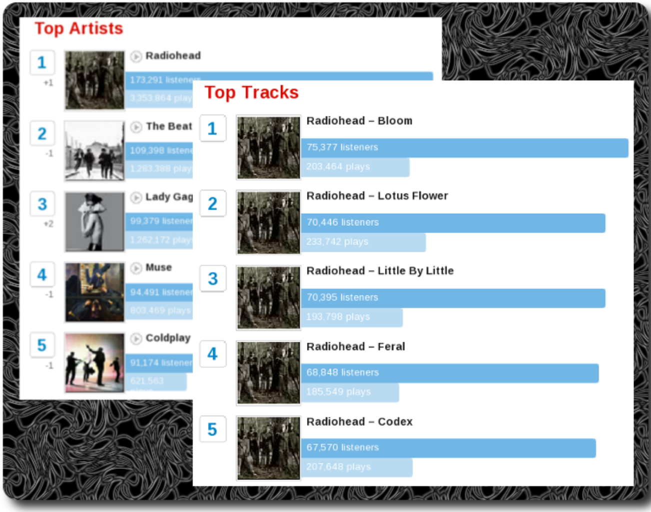
- Last.FM's *"Herd of Elephants"*
  - 44 Nodes
  - 8 cores per node
  - 16GB memory per node
  - 4 disks of 1 TB spinning at 7200 RPM per node
- Hive integration to run optimized SQL queries for analysis.

Last.fm Chart Generation (Track Statistics)

The goal of the Track Statistics program is to take incoming listening data and summarize it into a format that can be used to display on the website or used as input to other Hadoop programs.

Last.fm Chart Generation (Track Statistics)

The goal of the Track Statistics program is to take incoming listening data and summarize it into a format that can be used to display on the website or used as input to other Hadoop programs.



Top Artists

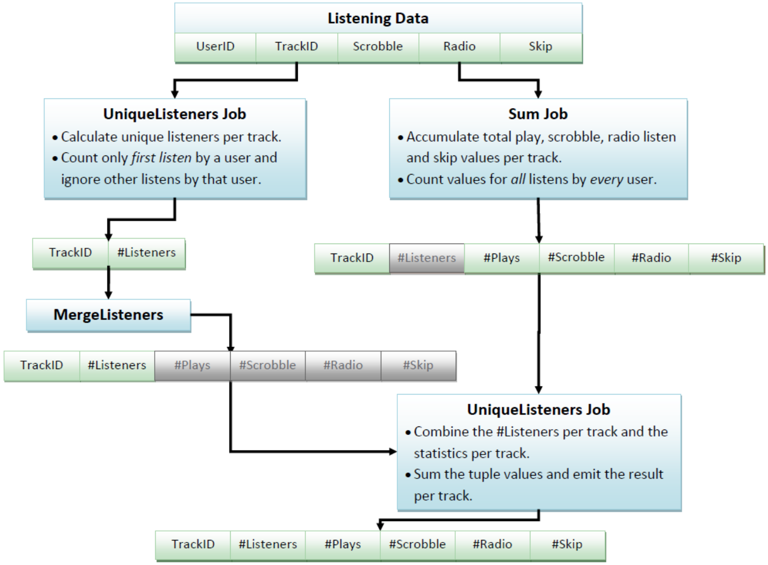
1	+1	Radiohead	173,291 listene / 3,353,864 play
2	-1	The Beat	109,398 listene / 1,283,388 play
3	+2	Lady Gag	99,379 listener / 1,262,172 play
4	-1	Muse	94,491 listener / 803,469 plays
5	-1	Coldplay	91,174 listener / 621,563

Top Tracks

1	Radiohead – Bloom	75,377 listeners	203,464 plays
2	Radiohead – Lotus Flower	70,446 listeners	233,742 plays
3	Radiohead – Little By Little	70,395 listeners	193,798 plays
4	Radiohead – Feral	68,848 listeners	185,549 plays
5	Radiohead – Codex	67,570 listeners	207,648 plays

Track Statistics Program

Two jobs to calculate values from the data, and a third job to merge the results.



Listening Data
UserID
TrackID
Scrobble
Radio
Skip
UniqueListeners Job
• Calculate unique listeners per track.
• Count only *first listen* by a user and ignore other listens by that user.
Sum Job
• Accumulate total play, scrobble, radio listen and skip values per track.
• Count values for *all* listens by *every* user.
TrackID
#Listeners
TrackID
#Listeners
#Plays
#Scrobble
#Radio
#Skip
MergeListeners
TrackID
#Listeners
#Plays
#Scrobble
#Radio
#Skip
UniqueListeners Job
• Combine the #Listeners per track and the statistics per track.
• Sum the tuple values and emit the result per track.
TrackID
#Listeners
#Plays
#Scrobble
#Radio
#Skip

Track Statistics Jobs

- **Input:** Gigabytes of space-delimited text files of the form (userID trackID scrobble radioPlay skip), where the last three fields are 0 or 1.

Track Statistics Jobs

- **Input:** Gigabytes of space-delimited text files of the form (userID trackID scrobble radioPlay skip), where the last three fields are 0 or 1.

UserId	TrackId	Scrobble	Radio	Skip
111115	222	0	1	0
111113	225	1	0	0
111117	223	0	1	1
111115	225	1	0	0

Track Statistics Jobs

- **Input:** Gigabytes of space-delimited text files of the form (userID trackID scrobble radioPlay skip), where the last three fields are 0 or 1.

UserId	TrackId	Scrobble	Radio	Skip
111115	222	0	1	0
111113	225	1	0	0
111117	223	0	1	1
111115	225	1	0	0

- **Output:** Charts require the following statistics per track:
  - Number of unique listeners
  - Number of scrobbles
  - Number of radio listens
  - Total number of listens
  - Number of radio skips

Track Statistics Jobs

- **Input:** Gigabytes of space-delimited text files of the form (userID trackID scrobble radioPlay skip), where the last three fields are 0 or 1.

UserId	TrackId	Scrobble	Radio	Skip
111115	222	0	1	0
111113	225	1	0	0
111117	223	0	1	1
111115	225	1	0	0

- **Output:** Charts require the following statistics per track:
  - Number of unique listeners
  - Number of scrobbles
  - Number of radio listens
  - Total number of listens
  - Number of radio skips

TrackId	numListeners	numPlays	numScrobbles	numRadioPlays	numSkips
`IntWritable`	`IntWritable`	`IntWritable`	`IntWritable`	`IntWritable`	`IntWritable`
222	0	1	0	1	0
223	1	1	0	1	1
225	2	2	2	0	0

UniqueListeners Job

Calculates the number of *unique* listeners per track.

- UniqueListenersMapper **input**:
  - key is the line number of the current log entry.
  - value is the space-delimited log entry.

UniqueListeners Job

Calculates the number of *unique* listeners per track.

- UniqueListenersMapper **input**:
  - key is the line number of the current log entry.
  - value is the space-delimited log entry.

LineOfFile	UserId	TrackId	Scrobble	Radio	Skip
LongWritable	IntWritable	IntWritable	Boolean	Boolean	Boolean
0	111115	222	0	1	0
1	111113	225	1	0	0
2	111117	223	0	1	1
3	111115	225	1	0	0

UniqueListeners Job

Calculates the number of *unique* listeners per track.

- UniqueListenersMapper **input**:
  - key is the line number of the current log entry.
  - value is the space-delimited log entry.

LineOfFile	**UserId**	**TrackId**	**Scrobble**	**Radio**	**Skip**
LongWritable	IntWritable	IntWritable	Boolean	Boolean	Boolean
0	111115	222	0	1	0
1	111113	225	1	0	0
2	111117	223	0	1	1
3	111115	225	1	0	0

- UniqueListenersMapper **function**:

UniqueListeners Job

Calculates the number of *unique* listeners per track.

- UniqueListenersMapper **input**:
  - key is the line number of the current log entry.
  - value is the space-delimited log entry.

LineOfFile	UserId	TrackId	Scrobble	Radio	Skip
LongWritable	IntWritable	IntWritable	Boolean	Boolean	Boolean
0	111115	222	0	1	0
1	111113	225	1	0	0
2	111117	223	0	1	1
3	111115	225	1	0	0

- UniqueListenersMapper **function**:
  - `if(scrobbles <= 0 && radioListens <=0) output nothing; else output(trackId, userId)`
  - `map(0, ‘111115 222 0 1 0’)` $\rightarrow$ `<222, 111115>`

UniqueListeners Job

Calculates the number of *unique* listeners per track.

- `UniqueListenersMapper` **input**:
  - `key` is the line number of the current log entry.
  - `value` is the space-delimited log entry.

LineOfFile	UserId	TrackId	Scrobble	Radio	Skip
LongWritable	IntWritable	IntWritable	Boolean	Boolean	Boolean
0	111115	222	0	1	0
1	111113	225	1	0	0
2	111117	223	0	1	1
3	111115	225	1	0	0

- `UniqueListenersMapper` **function**:
  - `if(scrobbles <= 0 && radioListens <=0) output nothing; else output(trackId, userId)`
  - `map(0, ‘111115 222 0 1 0’)` → `<222,  111115>`
- `UniqueListenersMapper` **output**:

UniqueListeners Job

Calculates the number of *unique* listeners per track.

- `UniqueListenersMapper` **input**:
  - `key` is the line number of the current log entry.
  - `value` is the space-delimited log entry.

LineOfFile	UserId	TrackId	Scrobble	Radio	Skip
`LongWritable`	`IntWritable`	`IntWritable`	`Boolean`	`Boolean`	`Boolean`
0	111115	222	0	1	0
1	111113	225	1	0	0
2	111117	223	0	1	1
3	111115	225	1	0	0

- `UniqueListenersMapper` **function**:
  - `if(scrobbles <= 0 && radioListens <=0) output nothing; else output(trackId, userId)`
  - `map(0, '111115 222 0 1 0')` $\rightarrow$ `<222,  111115>`
- `UniqueListenersMapper` **output**:

TrackId	UserId
`IntWritable`	`IntWritable`
222	111115
225	111113
223	111117
225	111115

UniqueListenersMapper Class

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

  @Override
  public void map(LongWritable offset, Text line,
    OutputCollector<IntWritable, IntWritable> output, Reporter reporter)
    throws IOException {

   String[] parts = (line.toString()).split(" ");

   int scrobbles = Integer.parseInt(parts[COL_SCROBBLE]);
   int radioListens = Integer.parseInt(parts[COL_RADIO]);

   /* Ignore track if marked with zero plays */
   if (scrobbles <= 0 && radioListens <= 0) return;

   /* Output user id against track id */
   IntWritable trackId = new IntWritable(Integer.parseInt(parts[COL_TRACKID]));
   IntWritable userId = new IntWritable(Integer.parseInt(parts[COL_USERID]));

   output.collect(trackId, userId);
  }
}
```

UniqueListenersReducer

- UniqueListenersReducer **input**:
  - key is the TrackID output by UniqueListenersMapper.
  - value is the iterator over the list of all UserIDs who listened to the track.

UniqueListenersReducer

- UniqueListenersReducer **input**:
  - key is the TrackID output by UniqueListenersMapper.
  - value is the iterator over the list of all UserIDs who listened to the track.

TrackId	**Iterator<UserIds>**
IntWritable	Iterator<IntWritable>
222	111115
225	111113 111115
223	111117

UniqueListenersReducer

- `UniqueListenersReducer` **input**:
  - `key` is the TrackID output by `UniqueListenersMapper`.
  - `value` is the iterator over the list of all UserIDs who listened to the track.

TrackId	**Iterator<UserIds>**
`IntWritable`	`Iterator<IntWritable>`
222	111115
225	111113 111115
223	111117

- `UniqueListenersReducer` **function**:

UniqueListenersReducer

- UniqueListenersReducer **input**:
  - key is the TrackID output by UniqueListenersMapper.
  - value is the iterator over the list of all UserIDs who listened to the track.

TrackId	**Iterator<UserIds>**
IntWritable	Iterator<IntWritable>
222	111115
225	111113 111115
223	111117

- UniqueListenersReducer **function**:
  - Add userIds to a HashSet as you iterate the list. Since a HashSet doesn't store duplicates, the size of the set will be the number of unique listeners.
  - reduce(225, '111115 111113') $\rightarrow$ <225, 2>

UniqueListenersReducer

- UniqueListenersReducer **input**:
  - key is the TrackID output by UniqueListenersMapper.
  - value is the iterator over the list of all UserIDs who listened to the track.

TrackId	**Iterator<UserIds>**
IntWritable	Iterator<IntWritable>
222	111115
225	111113 111115
223	111117

- UniqueListenersReducer **function**:
  - Add userIds to a HashSet as you iterate the list. Since a HashSet doesn't store duplicates, the size of the set will be the number of unique listeners.
  - reduce(225, '111115 111113') $\rightarrow$ <225, 2>
- UniqueListenersReducer **output**:

UniqueListenersReducer

- UniqueListenersReducer **input**:
  - key is the TrackID output by UniqueListenersMapper.
  - value is the iterator over the list of all UserIDs who listened to the track.

TrackId	**Iterator<UserIds>**
IntWritable	Iterator<IntWritable>
222	111115
225	111113 111115
223	111117

- UniqueListenersReducer **function**:
  - Add userIds to a HashSet as you iterate the list. Since a HashSet doesn't store duplicates, the size of the set will be the number of unique listeners.
  - reduce(225, '111115 111113') $\rightarrow$ <225, 2>
- UniqueListenersReducer **output**:

TrackId	**numListeners**
IntWritable	IntWritable
222	1
223	1
225	2

UniqueListenersReducer Class

```
/**
 * Receives a list of user IDs per track ID and puts them into a
 * set to remove duplicates. The size of this set (the number of
 * unique listeners) is emitted for each track.
 */
public static class UniqueListenersReducer extends MapReduceBase
  implements Reducer<IntWritable, IntWritable, IntWritable, IntWritable> {

  @Override
  public void reduce(IntWritable trackId, Iterator<IntWritable> values,
    OutputCollector<IntWritable, IntWritable> output, Reporter reporter)
    throws IOException {

    Set<Integer> userIds = new HashSet<Integer>();

    /* Add all users to set, duplicates automatically removed */
    while (values.hasNext()) {
      IntWritable userId = values.next();
      userIds.add(Integer.valueOf(userId.get()));
    }

    /* Output trackId -> number of unique listeners per track */
    output.collect(trackId, new IntWritable(userIds.size()));
  }
}
```

SumTrackStats Job

Adds up the scrobble, radio and skip values for each track.

- `SumTrackStatsMapper` **input**: The same as `UniqueListeners`.

SumTrackStats Job

Adds up the scrobble, radio and skip values for each track.

- SumTrackStatsMapper **input**: The same as UniqueListeners.

LineOfFile	UserId	TrackId	Scrobble	Radio	Skip
LongWritable	IntWritable	IntWritable	Boolean	Boolean	Boolean
0	111115	222	0	1	0
1	111113	225	1	0	0
2	111117	223	0	1	1
3	111115	225	1	0	0

SumTrackStats Job

Adds up the scrobble, radio and skip values for each track.

- `SumTrackStatsMapper` **input**: The same as `UniqueListeners`.

LineOfFile	UserId	TrackId	Scrobble	Radio	Skip
LongWritable	IntWritable	IntWritable	Boolean	Boolean	Boolean
0	111115	222	0	1	0
1	111113	225	1	0	0
2	111117	223	0	1	1
3	111115	225	1	0	0

- `SumTrackStatsMapper` **function**:

SumTrackStats Job

Adds up the scrobble, radio and skip values for each track.

- `SumTrackStatsMapper` **input**: The same as `UniqueListeners`.

LineOfFile	UserId	TrackId	Scrobble	Radio	Skip
`LongWritable`	`IntWritable`	`IntWritable`	`Boolean`	`Boolean`	`Boolean`
0	111115	222	0	1	0
1	111113	225	1	0	0
2	111117	223	0	1	1
3	111115	225	1	0	0

- `SumTrackStatsMapper` **function**:
  - Simply parse input and output the values as a new `TrackStats` object (see next slide).
  - `map(0, ‘111115 222 0 1 0’)` →
    `<222, new TrackStats(0, 1, 0, 1, 0)>`

SumTrackStats Job

Adds up the scrobble, radio and skip values for each track.

- `SumTrackStatsMapper` **input**: The same as `UniqueListeners`.

LineOfFile	UserId	TrackId	Scrobble	Radio	Skip
`LongWritable`	`IntWritable`	`IntWritable`	`Boolean`	`Boolean`	`Boolean`
0	111115	222	0	1	0
1	111113	225	1	0	0
2	111117	223	0	1	1
3	111115	225	1	0	0

- `SumTrackStatsMapper` **function**:
  - Simply parse input and output the values as a new `TrackStats` object (see next slide).
  - `map(0, ‘111115 222 0 1 0’)` $\rightarrow$
    `<222, new TrackStats(0, 1, 0, 1, 0)>`
- `SumTrackStatsMapper` **output**:

SumTrackStats Job

Adds up the scrobble, radio and skip values for each track.

- SumTrackStatsMapper **input**: The same as UniqueListeners.

LineOfFile	UserId	TrackId	Scrobble	Radio	Skip
LongWritable	IntWritable	IntWritable	Boolean	Boolean	Boolean
0	111115	222	0	1	0
1	111113	225	1	0	0
2	111117	223	0	1	1
3	111115	225	1	0	0

- SumTrackStatsMapper **function**:
  - Simply parse input and output the values as a new TrackStats object (see next slide).
  - `map(0, ‘111115 222 0 1 0’)` $\longrightarrow$
    `<222, new TrackStats(0, 1, 0, 1, 0)>`
- SumTrackStatsMapper **output**:

TrackId	numListeners	numPlays	numScrobbles	numRadioPlays	numSkips
IntWritable	IntWritable	IntWritable	IntWritable	IntWritable	IntWritable
222	0	1	0	1	0
225	0	1	1	0	0
223	0	1	0	1	1
225	0	1	1	0	0

TrackStats object

```
public class TrackStats implements WritableComparable<TrackStats> {

   private IntWritable listeners;
   private IntWritable plays;
   private IntWritable scrobbles;
   private IntWritable radioPlays;
   private IntWritable skips;

   public TrackStats(int numListeners, int numPlays, int numScrobbles,
                     int numRadio, int numSkips) {
        this.listeners = new IntWritable(numListeners);
        this.plays = new IntWritable(numPlays);
        this.scrobbles = new IntWritable(numScrobbles);
        this.radioPlays = new IntWritable(numRadio);
        this.skips = new IntWritable(numSkips);
   }

   public TrackStats() {
        this(0, 0, 0, 0, 0);
   }

   ...

}
```

SumTrackStatsMapper Class

```
public static class SumTrackStatsMapper extends MapReduceBase
    implements Mapper<LongWritable, Text, IntWritable,
    TrackStats> {

  @Override
  public void map(LongWritable offset, Text line,
    OutputCollector<IntWritable, TrackStats> output,
    Reporter reporter) throws IOException {

      String[] parts = (line.toString()).split(" ");
      int trackId = Integer.parseInt(parts[COL_TRACKID]);
      int scrobbles = Integer.parseInt(parts[COL_SCROBBLE]);
      int radio = Integer.parseInt(parts[COL_RADIO]);
      int skip = Integer.parseInt(parts[COL_SKIP]);

      TrackStats track = new TrackStats(0, scrobbles + radio,
                                       scrobbles, radio, skip);
      output.collect(new IntWritable(trackId), track);
   }
}
```

SumTrackStatsReducer

- `SumTrackStatsReducer` **input**:
  - `key` is the TrackID output by the mapper.
  - `value` is the iterator over the list of TrackStats associated with the track.

SumTrackStatsReducer

- SumTrackStatsReducer **input**:
  - key is the TrackID output by the mapper.
  - value is the iterator over the list of TrackStats associated with the track.

TrackId	**Iterator<TrackStats>**
IntWritable	Iterator<TrackStats>
222	(0,1,0,1,0)
225	(0,1,1,0,0) (0,1,1,0,0)
223	(0,1,0,1,1)

SumTrackStatsReducer

- SumTrackStatsReducer **input**:
  - key is the TrackID output by the mapper.
  - value is the iterator over the list of TrackStats associated with the track.

TrackId	**Iterator$<$TrackStats$>$**
IntWritable	Iterator$<$TrackStats$>$
222	(0,1,0,1,0)
225	(0,1,1,0,0) (0,1,1,0,0)
223	(0,1,0,1,1)

- SumTrackStatsReducer **function**:
  - Create new TrackStats object to hold totals for current track.

SumTrackStatsReducer

- `SumTrackStatsReducer` **input**:
  - `key` is the TrackID output by the mapper.
  - `value` is the iterator over the list of TrackStats associated with the track.

TrackId	**Iterator<TrackStats>**
`IntWritable`	`Iterator<TrackStats>`
222	(0,1,0,1,0)
225	(0,1,1,0,0) (0,1,1,0,0)
223	(0,1,0,1,1)

- `SumTrackStatsReducer` **function**:
  - Create new `TrackStats` object to hold totals for current track.
  - Iterate through values and add the stats of each value to the stats of the object we created.

SumTrackStatsReducer

- SumTrackStatsReducer **input**:
  - key is the TrackID output by the mapper.
  - value is the iterator over the list of TrackStats associated with the track.

TrackId	Iterator<TrackStats>
IntWritable	Iterator<TrackStats>
222	(0,1,0,1,0)
225	(0,1,1,0,0) (0,1,1,0,0)
223	(0,1,0,1,1)

- SumTrackStatsReducer **function**:
  - Create new TrackStats object to hold totals for current track.
  - Iterate through values and add the stats of each value to the stats of the object we created.
  - `reduce(225,‘(0,1,1,0,0) (0,1,1,0,0)’)` $\rightarrow$ `<225, (0,2,2,0,0)>`

SumTrackStatsReducer

- SumTrackStatsReducer **input**:
  - key is the TrackID output by the mapper.
  - value is the iterator over the list of TrackStats associated with the track.

TrackId	Iterator<TrackStats>
IntWritable	Iterator<TrackStats>
222	(0,1,0,1,0)
225	(0,1,1,0,0) (0,1,1,0,0)
223	(0,1,0,1,1)

- SumTrackStatsReducer **function**:
  - Create new TrackStats object to hold totals for current track.
  - Iterate through values and add the stats of each value to the stats of the object we created.
  - `reduce(225,‘(0,1,1,0,0) (0,1,1,0,0)’)` $\rightarrow$ `<225, (0,2,2,0,0)>`
- SumTrackStatsReducer **output**:

TrackId	numListeners	numPlays	numScrobbles	numRadioPlays	numSkips
IntWritable	IntWritable	IntWritable	IntWritable	IntWritable	IntWritable
222	0	1	0	1	0
223	0	1	0	1	1
225	0	2	2	0	0

SumTrackStatsReducer

```
public static class SumTrackStatsReducer extends MapReduceBase
    implements Reducer<IntWritable, TrackStats, IntWritable,
    TrackStats> {

  @Override
  public void reduce(IntWritable trackId, Iterator<TrackStats> values,
      OutputCollector<IntWritable, TrackStats> output, Reporter
      reporter) throws IOException {

      /* Sum totals for this track */
      TrackStats sum = new TrackStats();

      while(values.hasNext()) {
          TrackStats trackStats = (TrackStats) values.next();
          sum.setListeners(sum.getListeners() + trackStats.getListeners());
          sum.setPlays(sum.getPlays() + trackStats.getPlays());
          sum.setSkips(sum.getSkips() + trackStats.getSkips());
          sum.setScrobbles(sum.getScrobbles() + trackStats.getScrobbles());
          sum.setRadioPlays(sum.getRadioPlays() + trackStats.getRadioPlays());
      }
      output.collect(trackId, sum);
   }
}
```

Merging the Results: `MergeResults` Job

Merges the output from the `UniqueListeners` and `SumTrackStats` jobs.

Merging the Results: `MergeResults` Job

Merges the output from the `UniqueListeners` and `SumTrackStats` jobs.

- Specify a mapper for each input type:

Merging the Results: `MergeResults` Job

Merges the output from the `UniqueListeners` and `SumTrackStats` jobs.

- Specify a mapper for each input type:

```
MultipleInputs.addInputPath(conf, sumInputDir,
    SequenceFileInputFormat.class, IdentityMapper.class);

MultipleInputs.addInputPath(conf, listenersInputDir,
    SequenceFileInputFormat.class, MergeListenersMapper.class);
```

Merging the Results: MergeResults Job

Merges the output from the UniqueListeners and SumTrackStats jobs.

- Specify a mapper for each input type:

```
MultipleInputs.addInputPath(conf, sumInputDir,
    SequenceFileInputFormat.class, IdentityMapper.class);

MultipleInputs.addInputPath(conf, listenersInputDir,
    SequenceFileInputFormat.class, MergeListenersMapper.class);
```

- IdentityMapper simply emits the trackId and TrackStats object output by the SumTrackStats job.

Merging the Results: MergeResults Job

Merges the output from the UniqueListeners and SumTrackStats jobs.

- Specify a mapper for each input type:

```
MultipleInputs.addInputPath(conf, sumInputDir,
    SequenceFileInputFormat.class, IdentityMapper.class);

MultipleInputs.addInputPath(conf, listenersInputDir,
    SequenceFileInputFormat.class, MergeListenersMapper.class);
```

- IdentityMapper simply emits the trackId and TrackStats object output by the SumTrackStats job.
- IdentityMapper **input** and **output**:

TrackId	**numListeners**	**numPlays**	**numScrobbles**	**numRadioPlays**	**numSkips**
IntWritable	IntWritable	IntWritable	IntWritable	IntWritable	IntWritable
222	0	1	0	1	0
223	0	1	0	1	1
225	0	2	2	0	0

`MergeListenersMapper`

Prepares the data for input to the final reducer function by mapping the `TrackId` to a `TrackStats` object with the number of unique listeners set.

- `MergeListenersMapper` **input**: The `UniqueListeners` job output.

MergeListenersMapper

Prepares the data for input to the final reducer function by mapping the `TrackId` to a `TrackStats` object with the number of unique listeners set.

- `MergeListenersMapper` **input**: The `UniqueListeners` job output.

TrackId	**numListeners**
`IntWritable`	`IntWritable`
222	1
223	1
225	2

MergeListenersMapper

Prepares the data for input to the final reducer function by mapping the TrackId to a TrackStats object with the number of unique listeners set.

- MergeListenersMapper **input**: The UniqueListeners job output.

TrackId	numListeners
IntWritable	IntWritable
222	1
223	1
225	2

- MergeListenersMapper **function**:

MergeListenersMapper

Prepares the data for input to the final reducer function by mapping the `TrackId` to a `TrackStats` object with the number of unique listeners set.

- `MergeListenersMapper` **input**: The `UniqueListeners` job output.

TrackId	numListeners
`IntWritable`	`IntWritable`
222	1
223	1
225	2

- `MergeListenersMapper` **function**:
  - Create a new `TrackStats` object per track and set the `numListeners` attribute.
  - `map(225, 2)` $\longrightarrow$ `<225, new TrackStats(2, 0, 0, 0, 0)>`

MergeListenersMapper

Prepares the data for input to the final reducer function by mapping the TrackId to a TrackStats object with the number of unique listeners set.

- MergeListenersMapper **input**: The UniqueListeners job output.

TrackId	**numListeners**
IntWritable	IntWritable
222	1
223	1
225	2

- MergeListenersMapper **function**:
  - Create a new TrackStats object per track and set the `numListeners` attribute.
  - `map(225, 2)` $\rightarrow$ `<225, new TrackStats(2, 0, 0, 0, 0)>`
- MergeListenersMapper **output**:

MergeListenersMapper

Prepares the data for input to the final reducer function by mapping the TrackId to a TrackStats object with the number of unique listeners set.

- MergeListenersMapper **input**: The UniqueListeners job output.

TrackId	numListeners
IntWritable	IntWritable
222	1
223	1
225	2

- MergeListenersMapper **function**:
  - Create a new TrackStats object per track and set the numListeners attribute.
  - `map(225, 2)` $\rightarrow$ `<225, new TrackStats(2, 0, 0, 0, 0)>`
- MergeListenersMapper **output**:

TrackId	numListeners	numPlays	numScrobbles	numRadioPlays	numSkips
222	1	0	0	0	0
223	1	0	0	0	0
225	2	0	0	0	0

MergeListenersMapper

```
public static class MergeListenersMapper extends MapReduceBase
    implements Mapper<IntWritable, IntWritable, IntWritable,
    TrackStats>
{

    @Override
    public void map(IntWritable trackId, IntWritable listenerCount,
        OutputCollector<IntWritable, TrackStats> output,
        Reporter reporter) throws IOException {

      TrackStats trackStats = new TrackStats();
      trackStats.setListeners(listenerCount.get());

      output.collect(trackId, trackStats);
    }
}
```

Final Reduce Stage: `SumTrackStatsReducer`

Finally, we have two partially defined `TrackStats` objects for each track. We can reuse the `SumTrackStatsReducer` to combine them and emit the final result.

- `SumTrackStatsReducer` **input**:
  - `key` is the TrackID output by the two mappers.
  - `value` is the iterator over the list of TrackStats associated with the track (in this case, one contains the unique listener count and the other contains the play, scrobble, radio listen, and skip counts).

Final Reduce Stage: SumTrackStatsReducer

Finally, we have two partially defined TrackStats objects for each track. We can reuse the SumTrackStatsReducer to combine them and emit the final result.

- SumTrackStatsReducer **input**:
  - key is the TrackID output by the two mappers.
  - value is the iterator over the list of TrackStats associated with the track (in this case, one contains the unique listener count and the other contains the play, scrobble, radio listen, and skip counts).

TrackId	**Iterator<TrackStats>**
IntWritable	Iterator<TrackStats>
222	(1,0,0,0,0) (0,1,0,1,0)
223	(1,0,0,0,0) (0,1,0,1,1)
225	(2,0,0,0,0) (0,2,2,0,0)

Final Reduce Stage: SumTrackStatsReducer

Finally, we have two partially defined TrackStats objects for each track. We can reuse the SumTrackStatsReducer to combine them and emit the final result.

- SumTrackStatsReducer **input**:
  - key is the TrackID output by the two mappers.
  - value is the iterator over the list of TrackStats associated with the track (in this case, one contains the unique listener count and the other contains the play, scrobble, radio listen, and skip counts).

TrackId	**Iterator<TrackStats>**
IntWritable	Iterator<TrackStats>
222	(1,0,0,0,0) (0,1,0,1,0)
223	(1,0,0,0,0) (0,1,0,1,1)
225	(2,0,0,0,0) (0,2,2,0,0)

- SumTrackStatsReducer **function**:
  - `reduce(225,‘(2,0,0,0,0) (0,2,2,0,0)’)` → `<225, (2,2,2,0,0)>`

Final Reduce Stage: SumTrackStatsReducer

Finally, we have two partially defined TrackStats objects for each track. We can reuse the SumTrackStatsReducer to combine them and emit the final result.

- SumTrackStatsReducer **input**:
  - key is the TrackID output by the two mappers.
  - value is the iterator over the list of TrackStats associated with the track (in this case, one contains the unique listener count and the other contains the play, scrobble, radio listen, and skip counts).

TrackId	**Iterator<TrackStats>**
IntWritable	Iterator<TrackStats>
222	(1,0,0,0,0) (0,1,0,1,0)
223	(1,0,0,0,0) (0,1,0,1,1)
225	(2,0,0,0,0) (0,2,2,0,0)

- SumTrackStatsReducer **function**:
  - `reduce(225,‘(2,0,0,0,0) (0,2,2,0,0)’)` → `<225, (2,2,2,0,0)>`
- SumTrackStatsReducer **output**:

TrackId	**numListeners**	**numPlays**	**numScrobbles**	**numRadioPlays**	**numSkips**
IntWritable	IntWritable	IntWritable	IntWritable	IntWritable	IntWritable
222	0	1	0	1	0
223	1	1	0	1	1
225	2	2	2	0	0

Final SumTrackStatsReducer

From the SumTrackStats job.

```
public static class SumTrackStatsReducer extends MapReduceBase
    implements Reducer<IntWritable, TrackStats, IntWritable,
    TrackStats> {

  @Override
  public void reduce(IntWritable trackId, Iterator<TrackStats> values,
      OutputCollector<IntWritable, TrackStats> output, Reporter
      reporter) throws IOException {

      /* Sum totals for this track */
      TrackStats sum = new TrackStats();

      while(values.hasNext()) {
          TrackStats trackStats = (TrackStats) values.next();
          sum.setListeners(sum.getListeners() + trackStats.getListeners());
          sum.setPlays(sum.getPlays() + trackStats.getPlays());
          sum.setSkips(sum.getSkips() + trackStats.getSkips());
          sum.setScrobbles(sum.getScrobbles() + trackStats.getScrobbles());
          sum.setRadioPlays(sum.getRadioPlays() + trackStats.getRadioPlays());
      }
      output.collect(trackId, sum);
   }
}
```

Putting it all together

Hadoop provides a `ToolRunner` to simplify “job chaining“.

- All job classes should extend `Configured` and implement `Tool`.

  ```
  public class MergeResults extends Configured implements Tool
  ```

- Then override the `run` method with specific job configuration.

  ```
  public int run(String[] args) throws Exception {
    Path sums = new Path("sums");
    Path listeners = new Path("listeners");
    Path output = new Path("output");
    JobConf conf = new JobConf(UniqueListeners.class).setJobName("Merge");

    MultipleInputs.addInputPath(conf, sums,
            SequenceFileInputFormat.class, IdentityMapper.class);
    MultipleInputs.addInputPath(conf, listeners,
            SequenceFileInputFormat.class, MergeListenersMapper.class);
    FileOutputFormat.setOutputPath(conf, output);

    conf.setReducerClass(SumTrackStats.SumTrackStatsReducer.class);
    conf.setOutputKeyClass(IntWritable.class);
    conf.setOutputValueClass(TrackStats.class);

    try { JobClient.runJob(conf); } catch (Exception e) { return 1; }
    return 0;
  }
  ```

Putting it all together

The driver class simply runs each job in the correct order.

```
public class GenerateTrackStatistics {
    public static void main(String[] args) throws Exception {

      if(args.length != 2) {
         System.out.println("Usage: GenerateTrackStatistics "
                 + "<input path> <output path>");
         System.exit(0);
      }

      /* Run UniqueListeners job */
      int exitCode = ToolRunner.run(new UniqueListeners(), args);

      /* Run SumTrackStats job */
      if(exitCode == 0)
          exitCode = ToolRunner.run(new SumTrackStats(), args);

      /* Run Merge job */
      if(exitCode == 0)
          exitCode = ToolRunner.run(new MergeResults(), args);

      System.exit(exitCode);
    }
}
```

If you are interested in the complete code, email me at:

marissahollingsworth@u.boisestate.edu

References

- *MapReduce: Simplified Data Processing on Large Clusters*. Jeffrey Dean and Sanjay Ghemawat, Google Inc. OSDI 2004.
- *Hadoop: An open source implementation of MapReduce*. http://hadoop.apache.org/.
- *Can Your Programming Language Do This?* Joel Spolsky. http://www.joelonsoftware.com/items/2006/08/01.html
- *Hadoop: The Definitive Guide (2nd ed.)*. Tom White, October 2010, O'Reilly.
- *MapReduce tutorial at Yahoo*. http://developer.yahoo.com/hadoop/tutorial/
- *Hadoop Eclipse Plugin*: Jar file packaged with Hadoop in `contrib/eclipse-plugin` folder.
- *Data Clustering using MapReduce*. Makho Ngazimbi (supervised by Amit Jain). Masters in Computer Science project report, Boise State University, 2009.
- Last.fm. The main website: http://www.last.fm/.

New Hadoop API

The biggest change in 0.20 onwards is a large refactoring of the core MapReduce classes.

- All of the methods take *Context* objects that allow us to add new methods without breaking compatibility.
- *Mapper* and *Reducer* now have a "run" method that is called once and contains the control loop for the task, which lets applications replace it.
- *Mapper* and *Reducer* by default are Identity Mapper and Reducer.
- The *FileOutputFormats* use part-r-00000 for the output of reduce 0 and part-m-00000 for the output of map 0.
- The reduce grouping comparator now uses the raw compare instead of object compare.
- The number of maps in *FileInputFormat* is controlled by min and max split size rather than min size and the desired number of maps.

Hadoop added the classes in a new package at *org.apache.hadoop.mapreduce*.

WordCount example with new Hadoop API

Problem: To count the number of occurrences of each word in a large collection of documents.

```
/**
 * Counts the words in each line.
 * For each line of input, break the line into words
 * and emit them as (word, 1).
 */
public static class TokenizerMapper
       extends Mapper<Object, Text, Text, IntWritable>{

    private final static IntWritable one = new IntWritable(1);
    private Text word = new Text();

    public void map(Object key, Text value, Context context
                    ) throws IOException, InterruptedException {
      StringTokenizer itr = new StringTokenizer(value.toString());
      while (itr.hasMoreTokens()) {
        word.set(itr.nextToken());
        context.write(word, one);
      }
   }
}
```

WordCount Example with new Hadoop API (contd.)

```
/**
 * A reducer class that just emits the sum of the input values.
 */
 public static class IntSumReducer
       extends Reducer<Text,IntWritable,Text,IntWritable> {
    private IntWritable result = new IntWritable();

    public void reduce(Text key, Iterable<IntWritable> values,
                       Context context
                       ) throws IOException, InterruptedException {
      int sum = 0;
      for (IntWritable val : values) {
        sum += val.get();
      }
      result.set(sum);
      context.write(key, result);
    }
  }
```

WordCount Example with new Hadoop API (contd.)

```
public static void main(String[] args) throws Exception {
    Configuration conf = new Configuration();
    String[] otherArgs = new GenericOptionsParser(conf,
                             args).getRemainingArgs();
    if (otherArgs.length != 2) {
      System.err.println("Usage: wordcount <in> <out>");
      System.exit(2);
    }
    Job job = new Job(conf, "word count");
    job.setJarByClass(WordCount.class);
    job.setMapperClass(TokenizerMapper.class);
    job.setCombinerClass(IntSumReducer.class);
    job.setReducerClass(IntSumReducer.class);
    job.setOutputKeyClass(Text.class);
    job.setOutputValueClass(IntWritable.class);
    FileInputFormat.addInputPath(job, new Path(otherArgs[0]));
    FileOutputFormat.setOutputPath(job, new Path(otherArgs[1]));
    System.exit(job.waitForCompletion(true) ? 0 : 1);
  }
}
```