3
Extracting and Transforming Data

In this chapter, we will cover:

- Transforming Apache logs into TSV format using MapReduce
- Using Apache Pig to filter bot traffic from web server logs
- Using Apache Pig to sort web server log data by timestamp
- Using Apache Pig to sessionize web server log data
- Using Python to extend Apache Pig functionality
- Using MapReduce and secondary sort to calculate page views
- Using Hive and Python to clean and transform geographical event data
- Using Python and Hadoop Streaming to perform a time series analytic
- Using MultipleOutputs in MapReduce to name output files
- Creating custom Hadoop Writable and InputFormat to read geographical event data

Introduction

Parsing and formatting large amounts of data to meet business requirements is a challenging task. The software and the architecture must meet strict scalability, reliability, and run-time constraints. Hadoop is an ideal environment for extracting and transforming large-scale data. Hadoop provides a scalable, reliable, and distributed processing environment that is ideal for large-scale data processing. This chapter will demonstrate methods to extract and transform data using MapReduce, Apache Pig, Apache Hive, and Python.

Transforming Apache logs into TSV format using MapReduce

MapReduce is an excellent tool for transforming data into **tab-separated values** (**TSV**). Once the input data is loaded into HDFS, the entire Hadoop cluster can be utilized to transform large datasets in parallel. This recipe will demonstrate the method to extract records from Apache access logs and store those records as tab-separated values in HDFS.

Getting ready

You will need to download the `apache_clf.txt` dataset from the support page of the Packt website, `http://www.packtpub.com/support`, and place the file in HDFS.

How to do it...

Perform the following steps to transform Apache logs to TSV format using MapReduce:

1. Build a regular expression pattern to parse the Apache combined log format:

```
private Pattern p = Pattern.compile("^([\\d.]+) (\\S+) (\\S+) \\
[([\\w:/]+\\s[+\\-]\\d{4})\\] \"(\\w+) (.+?) (.+?)\" (\\d+) (\\d+)
\"([^\"]+|(.+?))\" \"([^\"]+|(.+?))\"", Pattern.DOTALL);
```

2. Create a mapper class to read the log files. The mapper should emit IP address as the key, and the following as values: timestamp, page, http status, bytes returned to the client, and the user agent of the client:

```
public class CLFMapper extends Mapper<Object, Text, Text, Text>{

    private SimpleDateFormat dateFormatter =
            new SimpleDateFormat("dd/MMM/yyyy:HH:mm:ss Z");
    private Pattern p =
            Pattern.compile("^([\\d.]+) (\\S+) (\\S+)"
            + " \\[([\\w:/]+\\s[+\\-]\\d{4})\\] \"(\\w+) (.+?)
(.+?)\" "
            + "(\\d+) (\\d+) \"([^\"]+|(.+?))\"
\"([^\"]+|(.+?))\"",
            Pattern.DOTALL);

    private Text outputKey = new Text();
    private Text outputValue = new Text();
    @Override
    protected void map(Object key, Text value, Context
      context) throws IOException, InterruptedException {
        String entry = value.toString();
```

```
            Matcher m = p.matcher(entry);
            if (!m.matches()) {
                return;
            }
            Date date = null;
            try {
                date = dateFormatter.parse(m.group(4));
            } catch (ParseException ex) {
                return;
            }
            outputKey.set(m.group(1)); //ip
            StringBuilder b = new StringBuilder();
            b.append(date.getTime()); //timestamp
            b.append('\t');
            b.append(m.group(6)); //page
            b.append('\t');
            b.append(m.group(8)); //http status
            b.append('\t');
            b.append(m.group(9)); //bytes
            b.append('\t');
            b.append(m.group(12)); //useragent
            outputValue.set(b.toString());
            context.write(outputKey, outputValue);
        }

    }
```

3. Now, create a map-only job to apply the transformation:

```
public class ParseWeblogs extends Configured implements Tool {

  public int run(String[] args) throws Exception {

    Path inputPath = new Path(args[0]);
    Path outputPath = new Path(args[1]);

    Configuration conf = getConf();
    Job weblogJob = new Job(conf);
    weblogJob.setJobName("Weblog Transformer");
    weblogJob.setJarByClass(getClass());
    weblogJob.setNumReduceTasks(0);
    weblogJob.setMapperClass(CLFMapper.class);
    weblogJob.setMapOutputKeyClass(Text.class);
    weblogJob.setMapOutputValueClass(Text.class);
    weblogJob.setOutputKeyClass(Text.class);
```

```
    weblogJob.setOutputValueClass(Text.class);
    weblogJob.setInputFormatClass(TextInputFormat.class);
    weblogJob.setOutputFormatClass(TextOutputFormat.class);

    FileInputFormat.setInputPaths(weblogJob, inputPath);
    FileOutputFormat.setOutputPath(weblogJob, outputPath);


    if(weblogJob.waitForCompletion(true)) {
      return 0;
    }
    return 1;
  }

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

}
```

4. Finally, launch the MapReduce job:

```
$ hadoop jar myjar.jar com.packt.ch3.etl.ParseWeblogs /user/
hadoop/apache_clf.txt /user/hadoop/apache_clf_tsv
```

How it works...

We first created a mapper that was responsible for the extraction of the desired information we from the Apache weblogs and for emitting the extracted fields in a tab-separated format.

Next, we created a map-only job to transform the web server log data into a tab-separated format. The key-value pairs emitted from the mapper were stored in a file in HDFS.

There's more...

By default, the `TextOutputFormat` class uses a tab to separate the key and value pairs. You can change the default separator by setting the `mapred.textoutputformat.separator` property. For example, to separate the IP and the timestamp by a ',', we could re-run the job using the following command:

```
$ hadoop jar myjar.jar com.packt.ch3.etl.ParseWeblogs -Dmapred.
textoutputformat.separator=',' /user/hadoop/apache_clf.txt /user/hadoop/
csv
```

See also

The tab-separated output from this recipe will be used in the following recipes:

- *Using Apache Pig to filter bot traffic from web server logs*
- *Using Apache Pig to sort web server log data by timestamp*
- *Using Apache Pig to sessionize web server log data*
- *Using Python to extend Apache Pig functionality*
- *Using MapReduce and secondary sort to calculate page views*

Using Apache Pig to filter bot traffic from web server logs

Apache Pig is a high-level language for creating MapReduce applications. This recipe will use Apache Pig and a Pig user-defined filter function (UDF) to remove all bot traffic from a sample web server log dataset. **Bot traffic** is the non-human traffic that visits a webpage, such as **spiders**.

Getting ready

You will need to download/compile/install the following:

- Version 0.8.1 or better of Apache Pig from `http://pig.apache.org/`
- Test data: `apache_tsv.txt` and `useragent_blacklist.txt` from the support page on the Packt website, `http://www.packtpub.com/support`
- Place `apache_tsv.txt` in HDFS and put `useragent_blacklist.txt` in your current working directory

How to do it...

Carry out the following steps to filter bot traffic using an Apache Pig UDF:

1. First, write a Pig UDF that extends the Pig `FilterFunc` abstract class. This class will be used to filter records in the weblogs dataset by using the user agent string.

```
public class IsUseragentBot extends FilterFunc {

    private Set<String> blacklist = null;

    private void loadBlacklist() throws IOException {
        blacklist = new HashSet<String>();
        BufferedReader in = new BufferedReader(new
```

```
            FileReader("blacklist"));
        String userAgent = null;
        while ((userAgent = in.readLine()) != null) {
            blacklist.add(userAgent);
        }
    }

    @Override
    public Boolean exec(Tuple tuple) throws IOException {
        if (blacklist == null) {
            loadBlacklist();
        }
        if (tuple == null || tuple.size() == 0) {
            return null;
        }

        String ua = (String) tuple.get(0);
        if (blacklist.contains(ua)) {
            return true;
        }
        return false;
    }

}
```

2. Next, create a Pig script in your current working directory. At the beginning of the Pig script, give the MapReduce framework the path to `useragent_blacklist.txt` in HDFS:

```
set mapred.cache.files '/user/hadoop/blacklist.txt#blacklist';
set mapred.create.symlink 'yes';
```

3. Register the JAR file containing the `IsUseragentBot` class with Pig, and write the Pig script to filter the weblogs by the user agent:

```
register myudfjar.jar;


all_weblogs = LOAD '/user/hadoop/apache_tsv.txt' AS (ip:
chararray, timestamp:long, page:chararray, http_status:int,
payload_size:int, useragent:chararray);
```

```
nobots_weblogs = FILTER all_weblogs BY NOT com.packt.ch3.etl.pig.
IsUseragentBot(useragent);


STORE nobots_weblogs INTO '/user/hadoop/nobots_weblogs';
```

To run the Pig job, put `myudfjar.jar` into the same folder as the Pig script and execute it.

```
$ ls
$ myudfjar.jar filter_bot_traffic.pig
$ pig –f filter_bot_traffic.pig
```

How it works...

Apache Pig is extendable through the use of user-defined functions (UDF). One way to create a UDF is through the use of the Java abstract classes and interfaces that come with the Apache Pig distribution. In this recipe, we wanted to remove all records that contain known bot user agent strings. One way to do this is to create our own Pig filter.

The `IsUseragentBot` class extends the abstract class `FilterFunc`, which allows us to override the `exec(Tuple t)` method. A Pig Tuple is an ordered list of fields that can be any Pig primitive, or null. At runtime, Pig will feed the `exec(Tuple t)` method of the `IsUseragentBot` class with the user agent strings from our dataset. The UDF will extract the user agent string by accessing the first field in the Tuple, and it will return `true` if we find the user agent string is a bot, otherwise the UDF returns `false`.

In addition, the `IsUseragentBot` UDF reads a file called `blacklist` and loads the contents into a `HashSet` instance. The file named `blacklist` is a symbolic link to `blacklist.txt`, which has been distributed to the nodes in the cluster using the **distributed cache** mechanism. To place a file into the distributed cache, and to create the symbolic link, set the following MapReduce properties:

```
set mapred.cache.files '/user/hadoop/blacklist.txt#blacklist';
set mapred.create.symlink 'yes';
```

It is important to note that these properties are not Pig properties. These properties are used by the MapReduce framework, so you can use these properties to load a file to the distributed cache for any MapReduce job.

Next, we told Pig where to find the JAR file containing the `IsUseragentBot` UDF:

```
register myudfjar.jar;
```

Finally, we call the UDF using the Java class name. When the job runs, Pig will instantiate an instance of the `IsUseragentBot` class and feed the `exec(Tuple t)` method with records from the `all_weblogs` relation.

There's more...

Starting in Pig Version 0.9, Pig UDFs can access the distributed cache without setting the `mapred.cache.files` and `mapred.create.symlink` properties. Most abstract Pig classes that used to create UDFs now have a method named `List<String> getCacheFiles()` that can be overridden to load files from HDFS into the distributed cache. For example, the `IsUseragentBot` class can be modified to load the `blacklist.txt` file to the distributed cache by adding the following method:

```
@Override
public List<String> getCacheFiles() {
      List<String> list = new ArrayList<String>();
      list.add("/user/hadoop/blacklist.txt#blacklist");
      return list;
}
```

See also

Apache Pig will be used with the following recipes in this chapter:

- *Using Apache Pig to sort web server log data by timestamp*
- *Using Apache Pig to sessionize web server log data*
- *Using Python to extend Apache Pig functionality*
- *Using MapReduce and secondary sort to calculate page views*

Using Apache Pig to sort web server log data by timestamp

Sorting data is a common data transformation technique. In this recipe, we will demonstrate the method of writing a simple Pig script to sort a dataset using the distributed processing power of the Hadoop cluster.

Getting ready

You will need to download/compile/install the following:

- Version 0.8.1 or better of Apache Pig from `http://pig.apache.org/`
- Test data: `apache_nobots_tsv.txt` from `http://www.packtpub.com/support`

How to do it...

Perform the following steps to sort data using Apache Pig:

1. First load the web server log data into a Pig relation:

```
nobots_weblogs = LOAD '/user/hadoop/apache_nobots_tsv.txt' AS
(ip: chararray, timestamp:long, page:chararray, http_status:int,
payload_size:int, useragent:chararray);
```

2. Next, order the web server log records by the `timestamp` field in the ascending order:

```
ordered_weblogs = ORDER nobots BY timestamp;
```

3. Finally, store the sorted results in HDFS:

```
STORE ordered_weblogs INTO '/user/hadoop/ordered_weblogs';\
```

4. Run the Pig job:

```
$ pig –f ordered_weblogs.pig
```

How it works...

Sorting data in a distributed, share-nothing environment is non-trivial. The Pig relational operator `ORDER BY` has the capability to provide total ordering of a dataset. This means any record that appears in the output file `part-00000`, will have a timestamp less than the timestamp in the output file `part-00001` (since our data was sorted by `timestamp`).

There's more...

The Pig `ORDER BY` relational operator sorts data by multiple fields, and also supports sorting data in the descending order. For example, to sort the `nobots` relationship by the `ip` and `timestamp` fields, we would use the following expression:

```
ordered_weblogs = ORDER nobots BY ip, timestamp;
```

To sort the `nobots` relationship by `timestamp` in the descending order, use the `desc` option:

```
ordered_weblogs = ORDER nobots timestamp desc;
```

See also

The following recipes will use Apache Pig:

- *Using Apache Pig to sessionize web server log data*
- *Using Python to extend Apache Pig functionality*
- *Using MapReduce and secondary sort to calculate page views*

Using Apache Pig to sessionize web server log data

A session represents a user's continuous interaction with a website, and the user session ends when an arbitrary activity timeout has occurred. A new session begins once the user returns to the website after a period of inactivity. This recipe will use Apache Pig and a Pig user-defined function (UDF) to generate a subset of records from `apache_nobots_tsv.txt` that marks the beginning of a session for a specific IP.

Getting ready

You will need to download/compile/install the following:

- Version 0.8.1 or better of Apache Pig from `http://pig.apache.org/`
- Test data: `apache_nobots_tsv.txt` from `http://www.packtpub.com/support`

How to do it...

The following are the steps to create an Apache Pig UDF to sessionize web server log data:

1. Start by creating a Pig UDF to emit only the first record of a session. The UDF extends the Pig abstract class `EvalFunc` and implements the Pig interface, `Accumulator`. This class is responsible for applying the session logic on the web server log dataset:

```
public class Sessionize extends EvalFunc<DataBag> implements
Accumulator<DataBag> {

    private long sessionLength = 0;
    private Long lastSession = null;
    private DataBag sessionBag = null;

    public Sessionize(String seconds) {
        sessionLength = Integer.parseInt(seconds) * 1000;
        sessionBag = BagFactory.getInstance().newDefaultBag();
    }

    @Override
    public DataBag exec(Tuple tuple) throws IOException {
        accumulate(tuple);
```

```
        DataBag bag = getValue();
        cleanup();
        return bag;
    }

    @Override
    public void accumulate(Tuple tuple) throws IOException {
        if (tuple == null || tuple.size() == 0) {
            return;
        }
        DataBag inputBag = (DataBag) tuple.get(0);
        for(Tuple t: inputBag) {
            Long timestamp = (Long)t.get(1);
            if (lastSession == null) {
                sessionBag.add(t);
            }
            else if ((timestamp - lastSession) >= sessionLength) {
                sessionBag.add(t);
            }
            lastSession = timestamp;
        }
    }

    @Override
    public DataBag getValue() {
        return sessionBag;
    }    @Override
    public void cleanup() {
        lastSession = null;
        sessionBag = BagFactory.getInstance().newDefaultBag();
    }
}
```

2. Next, create a Pig script to load and group the web server log records by IP address:

```
register myjar.jar;
define Sessionize com.packt.ch3.etl.pig.Sessionize('1800'); /* 30
minutes */

nobots_weblogs = LOAD '/user/hadoop/apache_nobots_tsv.txt' AS
(ip: chararray, timestamp:long, page:chararray, http_status:int,
payload_size:int, useragent:chararray);

ip_groups = GROUP nobots_weblogs BY ip;
```

3. Finally, write the Pig expression to order all of the records associated with a specific IP by timestamp. Then, send the ordered records to the `Sessionize` UDF:

```
sessions = FOREACH ip_groups {
                ordered_by_timestamp = ORDER nobots_weblogs BY
timestamp;
                GENERATE FLATTEN(Sessionize(ordered_by_
timestamp));
           }

STORE sessions INTO '/user/jowens/sessions';
```

4. Copy the JAR file containing the `Sessionize` class to the current working directory, and run the Pig script:

```
$ pig –f sessionize.pig
```

How it works...

We first created a UDF that extended the `EvalFunc` abstract class and implemented the `Accumulator` interface. The `EvalFunc` class is used to create our own function that can be used within a Pig script. Data will be passed to the UDF via the `exec(Tuple t)` method, where it is processed. The `Accumulator` interface is optional for custom `eval` functions, and allows Pig to optimize the data flow and memory utilization of the UDF. Instead of passing the whole dataset, similar to how the `EvalFunc` class works, the `Accumulator` interface allows for subsets of the data to be passed to the UDF.

Next, we wrote a Pig script to group all of the web server log records by IP, and sort the records by timestamp. We need the data sorted by timestamp because the `Sessionize` UDF uses the sorted order of the timestamps to determine the start of each session.

Then, we generated all of the sessions associated with a specific IP by calling the `Sessionize` alias.

Finally, we used the `FLATTEN` operator to unnest the Tuples in the DataBags emitted from the UDF.

See also

- *Using Python to extend Apache Pig functionality*

Using Python to extend Apache Pig functionality

In this recipe, we will use Python to create a simple Apache Pig user-defined function (UDF) to count the number of records in a Pig DataBag.

Getting ready

You will need to download/compile/install the following:

- Jython 2.5.2 from `http://www.jython.org/`
- Version 0.8.1 or better of Apache Pig from `http://pig.apache.org/`
- Test data: `apache_nobots_tsv.txt` from `http://www.packtpub.com/support`

This recipe requires the Jython standalone JAR file. To build the file, download the Jython java installer, run the installer, and select **Standalone** from the installation menu.

```
$ java –jar jython_installer-2.5.2.jar
```

Add the Jython standalone JAR file to Apache Pig's classpath:

```
$ export PIG_CLASSPATH=$PIG_CLASSPATH:/path/to/jython2.5.2/jython.jar
```

How to do it...

The following are the steps to create an Apache Pig UDF using Python:

1. Start by creating a simple Python function to count the number of records in a Pig DataBag:

```
#!/usr/bin/python

@outputSchema("hits:long")
def calculate(inputBag):
  hits = len(inputBag)
  return hits
```

2. Next, create a Pig script to group all of the web server log records by IP and page. Then send the grouped web server log records to the Python function:

```
register 'count.py' using jython as count;

nobots_weblogs = LOAD '/user/hadoop/apache_nobots_tsv.txt' AS
(ip: chararray, timestamp:long, page:chararray, http_status:int,
payload_size:int, useragent:chararray);

ip_page_groups = GROUP nobots_weblogs BY (ip, page);

ip_page_hits = FOREACH ip_page_groups GENERATE FLATTEN(group),
count.calculate(nobots_weblogs);

STORE ip_page_hits INTO '/user/hadoop/ip_page_hits';
```

How it works...

First, we created a simple Python function to calculate the length of a Pig DataBag. In addition, the Python script contained the Python decorator, `@outputSchema("hits:long")`, that instructs Pig on how to interpret the data returned by the Python function. In this case, we want Pig to store the data returned by this function as a Java Long in a field named `hits`.

Next, we wrote a Pig script that registers the Python UDF using the statement:

```
register 'count.py' using jython as count;
```

Finally, we called the `calculate()` function using the alias `count`, in the Pig DataBag:

```
count.calculate(nobots_weblogs);
```

Using MapReduce and secondary sort to calculate page views

In a typical MapReduce job, key-value pairs are emitted from the mappers, shuffled, and sorted, and then finally passed to the reducers. There is no attempt by the MapReduce framework to sort the values passed to the reducers for processing. However, there are cases when we need the values passed to the reducers to be sorted, such as in the case of counting page views.

To calculate page views, we need to calculate distinct IPs by page. One way to calculate this is to have the mappers emit the key-value pairs: page and IP. Then, in the reducer, we can store all of the IPs associated with a page in a set. However, this approach is not scalable. What happens if the weblogs contain a large number of distinct IPs visiting a single page? We might not be able to fit the entire set of distinct IPs in memory.

The MapReduce framework provides a way to work around this complication. In this recipe, we will write a MapReduce application that allows us to sort the values going to a reducer using an approach known as the **secondary sort**. Instead of holding all of the distinct IPs in memory, we can keep track of the last IP we saw while processing the values in the reducer, and we can maintain a counter to calculate distinct IPs.

Getting ready

You will need to download the `apache_nobots_tsv.txt` dataset from `http://www.packtpub.com/support` and place the file into HDFS.

How to do it...

The following steps show how to implement a secondary sort in MapReduce to calculate page views:

1. Create a class that implements the Hadoop `WritableComparable` interface. We will use this class to store the key and sort fields:

```
public class CompositeKey implements WritableComparable {

    private Text first = null;
    private Text second = null;

    public CompositeKey() {

    }

    public CompositeKey(Text first, Text second) {
        this.first = first;
        this.second = second;
    }

    //...getters and setters

    public void write(DataOutput d) throws IOException {
        first.write(d);
        second.write(d);
    }
```

```
    public void readFields(DataInput di) throws IOException {
        if (first == null) {
            first = new Text();
        }
        if (second == null) {
            second = new Text();
        }
        first.readFields(di);
        second.readFields(di);
    }

    public int compareTo(Object obj) {
        CompositeKey other = (CompositeKey) obj;
        int cmp = first.compareTo(other.getFirst());
        if (cmp != 0) {
            return cmp;
        }
        return second.compareTo(other.getSecond());
    }

    @Override
    public boolean equals(Object obj) {
        CompositeKey other = (CompositeKey)obj;
        return first.equals(other.getFirst());
    }

    @Override
    public int hashCode() {
        return first.hashCode();
    }
}
```

2. Next, write the `Mapper` and `Reducer` classes. The `Mapper` class will use the `CompositeKey` class to store two fields. The first will be the `page` field, which is used to group and partition the data leaving the mapper. The second is the `ip` field, which is used to sort the values passed to the reducer.

```
public class PageViewMapper extends Mapper<Object, Text,
CompositeKey, Text> {
    private CompositeKey compositeKey = new CompositeKey();
    private Text first = new Text();
    private Text second = new Text();
    private Text outputValue = new Text();
    @Override
```

```
    protected void map(Object key, Text value, Context
      context) throws IOException, InterruptedException {
        String[] tokens = value.toString().split("\t");
        if (tokens.length > 3) {
            String page = tokens[2];
            String ip = tokens[0];
            first.set(page);
            second.set(ip);
            compositeKey.setFirst(first);
            compositeKey.setSecond(second);
            outputValue.set(ip);
            context.write(compositeKey, outputValue);
        }
    }
}

public class PageViewReducer extends Reducer<CompositeKey, Text,
Text, LongWritable> {
    private LongWritable pageViews = new LongWritable();

    @Override
    protected void reduce(CompositeKey key, Iterable<Text>
      values, Context context)
      throws IOException, InterruptedException {
        String lastIp = null;
        long pages = 0;
        for(Text t : values) {
            String ip = t.toString();
            if (lastIp == null) {
                lastIp = ip;
                pages++;
            }
            else if (!lastIp.equals(ip)) {
                lastIp = ip;
                pages++;
            }
            else if (lastIp.compareTo(ip) > 0) {
                throw new IOException("secondary sort failed");
            }
        }
        pageViews.set(pages);
        context.write(key.getFirst(), pageViews);
    }
}
```

3. Create three classes to partition, group, and sort the data leaving the mapper. These classes are used by the MapReduce framework. First, write a class to partition the data emitted from the mapper by the `page` field:

```
static class CompositeKeyParitioner extends
Partitioner<CompositeKey, Writable> {

        @Override
        public int getPartition(CompositeKey key, Writable value,
int numParition) {
            return (key.getFirst().hashCode() &  0x7FFFFFFF) %
numParition;
        }
    }
```

4. Next, write a Comparator that will group all of the keys together:

```
static class GroupComparator extends WritableComparator {
        public GroupComparator() {
            super(CompositeKey.class, true);
        }

        @Override
        public int compare(WritableComparable a,
WritableComparable b) {
            CompositeKey lhs = (CompositeKey)a;
            CompositeKey rhs = (CompositeKey)b;
            return lhs.getFirst().compareTo(rhs.getFirst());
        }
    }
```

5. Write a second Comparator that will sort the values passed to the reducer:

```
static class SortComparator extends WritableComparator {
        public SortComparator() {
            super(CompositeKey.class, true);
        }

        @Override
        public int compare(WritableComparable a,
WritableComparable b) {
            CompositeKey lhs = (CompositeKey)a;
            CompositeKey rhs = (CompositeKey)b;
            int cmp = lhs.getFirst().compareTo(rhs.getFirst());
            if (cmp != 0) {
                return cmp;
            }
```

```
            return lhs.getSecond().compareTo(rhs.getSecond());
        }
    }
```

6. Finally, write the code to set up a normal MapReduce job, but tell the MapReduce framework to use our own partitioner and comparator classes:

```
public int run(String[] args) throws Exception {

        Path inputPath = new Path(args[0]);
        Path outputPath = new Path(args[1]);

        Configuration conf = getConf();
        Job weblogJob = new Job(conf);
        weblogJob.setJobName("PageViews");
        weblogJob.setJarByClass(getClass());
        weblogJob.setMapperClass(PageViewMapper.class);
        weblogJob.setMapOutputKeyClass(CompositeKey.class);
        weblogJob.setMapOutputValueClass(Text.class);

        weblogJob.setPartitionerClass(CompositeKeyParitioner.
class);
        weblogJob.setGroupingComparatorClass(GroupComparator.
class);
        weblogJob.setSortComparatorClass(SortComparator.class);

        weblogJob.setReducerClass(PageViewReducer.class);
        weblogJob.setOutputKeyClass(Text.class);
        weblogJob.setOutputValueClass(Text.class);
        weblogJob.setInputFormatClass(TextInputFormat.class);
        weblogJob.setOutputFormatClass(TextOutputFormat.class);

        FileInputFormat.setInputPaths(weblogJob, inputPath);
        FileOutputFormat.setOutputPath(weblogJob, outputPath);


        if(weblogJob.waitForCompletion(true)) {
            return 0;
        }
        return 1;
    }
```

How it works...

We first created a class named `CompositeKey`. This class extends the Hadoop `WritableComparable` interface so that we can use the `CompositeKey` class just like any normal Hadoop `WritableComparable` interface (for example, `Text` and `IntWritable`). The `CompositeKey` class holds two `Text` objects. The first `Text` object is used to partition and group the key-value pairs emitted from the mapper. The second `Text` object is used to perform the secondary sort.

Next, we wrote a mapper class to emit the key-value pair `CompositeKey` (which consists of page and IP) as the key, and IP as the value. In addition, we wrote a reducer class that receives a `CompositeKey` object and a sorted list of IPs. The distinct IP count is calculated by incrementing a counter whenever we see an IP that does not equal a previously seen IP.

After writing the mapper and reducer classes, we created three classes to partition, group, and sort the data. The `CompositeKeyPartitioner` class is responsible for partitioning the data emitted from the mapper. In this recipe, we want all of the same pages to go to the same partition. Therefore, we calculate the partition location based only on the first field of the `CompositeKey` class.

Next, we created a `GroupComparator` class that uses the same logic as `CompositeKeyPartitioner`. We want all of the same page keys grouped together for processing by a reducer. Therefore, the group comparator only inspects the first member of the `CompositeKey` class for comparison.

Finally, we created the `SortComparator` class. This class is responsible for sorting all of the values that are sent to the reducer. As you can see from the method signature, `compare(WritableComparable a, WritableComparable b)`, we only receive the keys that are sent to each reducer, which is why we needed to include the IP with each and every key the mapper emitted. The `SortComparator` class compares both the first and second members of the `CompositeKey` class to ensure that the values a reducer receives are sorted.

See also

- *Creating custom Hadoop Writable and InputFormat to read geographical event data*

Using Hive and Python to clean and transform geographical event data

This recipe uses certain operators in Hive to input/output data through a custom Python script. The script performs a few simple pruning operations over each row, and outputs a slightly modified version of the row into a Hive table.

Getting ready

You will need to download/compile/install the following:

- Version 0.7.1 of Apache Hive from `http://hive.apache.org/`
- Test data: `Nigeria_ACLED.csv`, `Nigeria_ACLED_cleaned.tsv` from `http://www.packtpub.com/support`
- Python 2.7 or greater

This recipe requires the `Nigera_ACLED.csv` file to be loaded into a Hive table named `acled_nigeria` with the following fields mapped to the respective data types.

Issue the following command to the Hive client:

```
describe acled_nigeria
```

You should see the following response:

```
OK
loc string
event_date  string
year  string
event_type   string
actor  string
latitude  double
longitude double
source string
fatalities   string
```

How to do it...

Follow these steps to use Python and Hive to transform data:

1. Create a file named `clean_and_transform_acled.hql` in your local working directory and add the inline creation and transformation syntax:

```
SET mapred.child.java.opts=-Xmx512M;

DROP TABLE IF EXISTS acled_nigeria_cleaned;
CREATE TABLE acled_nigeria_cleaned (
    loc STRING,
    event_date STRING,
    event_type STRING,
```

```
    actor STRING,
    latitude DOUBLE,
    longitude DOUBLE,
    source STRING,
    fatalities INT
) ROW FORMAT DELIMITED;

ADD FILE ./clean_acled_nigeria.py;
INSERT OVERWRITE TABLE acled_nigeria_cleaned
    SELECT TRANSFORM(
            if(loc != "", loc, 'Unknown'),
            event_date,
            year,
            event_type,
            actor,
            latitude,
            longitude,
            source,
            if(fatalities != "", fatalities, 'ZERO_FLAG'))
    USING 'python clean_acled_nigeria.py'
    AS (loc, event_date, event_type, actor, latitude, longitude,
source, fatalities)
    FROM acled_nigeria;
```

2. Next, create another file named `clean_acled_nigeria.py` in the same working directory as `clean_and_transform_acled.hql` and add the following Python code to read from stdin:

```
#!/usr/bin/env python
import sys

for line in sys.stdin:
   (loc, event_date, year, event_type, actor, lat, lon, src,
fatalities) = line.strip().split('\t')
  if loc != 'LOCATION': #remove header row
    if fatalities == 'ZERO_FLAG':
    fatalities = '0'
      print '\t'.join([loc, event_date, event_type, \ actor, lat,
lon, src, fatalities]) #strip out year
```

It is important to note that Python is sensitive to inconsistent indentation. Be careful if you are copying and pasting Python code.

3. Run the script from the operating system shell by supplying the `-f` option to the Hive client:

```
$ hive -f clean_and_transform_acled.hql
```

4. To verify that the script finished properly, run the following command using the `-e` option to the Hive client.

```
hive -e "select count(1) from acled_nigeria_cleaned"
```

Hive should count 2931 rows.

How it works...

Let's start with the Hive script that we created. The first line is simply to force a certain JVM heap size in our execution. You can set this to any size that may be appropriate for your cluster. For the ACLED Nigeria dataset, a 512 MB memory is more than enough.

Immediately following this, we drop any tables with the name `acled_nigeria_cleaned` and create a table by the same name. We can omit the fields delimited by `','` and rows delimited by `'\n'` since they are the default field and row delimiters assumed by `ROW FORMAT`, and the ACLED Nigeria data is in that format.

Once we have our receiving table defined, we need to define the `SELECT` statement that will transform and output the data. The common convention is to add scripts required by `SELECT` before the statement. The command `ADD FILE ./clean_acled_nigeria.py` tells Hive to load the script from the local filesystem into the distributed cache for use by the MapReduce tasks.

The `SELECT` statement uses the Hive `TRANSFORM` operator to separate each column by tabs and to cast all columns as `String` with nulls as `'\n'`. The columns `loc` and `fatalities` are conditionally checked for empty strings; and if found to be empty, are set to a default value.

We specify the `USING` operator to provide a custom script to work with the `TRANSFORM` operator. Hive requires that scripts that make a call to the `USING` operator for row transformation need to first invoke `TRANSFORM` with the appropriate columns. If the file has been placed on the distributed cache, and each node in the cluster has Python installed, the MapReduce JVM tasks will be able to execute the script and read the rows in parallel. The `AS` operator contains a list of named fields corresponding to the columns found in the receiving Hive table, `acled_nigeria_cleaned`.

The Python script is very straightforward. The `#!/usr/bin/env python` statement is a hint to tell the shell how to execute the script. Each row from the table is passed in as a line over standard input. The call to `strip()` method removes any leading/trailing whitespace, and then we tokenize it into an array of named variables. Each field from the row is put in a named variable. The raw ACLED Nigeria data was used to create the input Hive table, and contains a header row we wish to discard. The first condition will check for `'LOCATION'` as the value of `loc`, which indicates the header row we want to ignore.

If the row passes this check, we look for the presence of `'ZERO_FLAG'` as the value for `fatalities`, which we set in our Hive script. If the script detects this value for `fatalities`, we set the value of `fatalities` to the string `'0'`.

Finally, we output each field excluding `year` in the same order as it was input. Each row will be placed into the table `acled_nigeria_cleaned`.

There's more...

There is a lot going on in this recipe. The following are a few additional explanations that will help you with Hive `TRANSFORM/USING/AS` operations and ETL in general.

Making every column type String

This is a bit counterintuitive and certainly not found anywhere in the Hive documentation. If your initial Hive staging table for the incoming data maps each delimited field as a string, it will aid tremendously in data validation and debugging. You can use the Hive `STRING` type to successfully represent almost any input into a cleansing script or direct Hive `QL` statement. Trying to perfectly map datatypes over expected values is not flexible to an erroneous input. There may be malformed characters for fields where you expect numeric values, and other similar hang-ups that make it impossible to perform certain analytics. Using strings over the raw data fields will allow a custom script to inspect the invalid data and decide how to respond. Moreover, when dealing with CSV or tab-separated data, a slightly misaligned `INT` or `FLOAT` type mapping in your Hive table declaration, where the data has a `STRING`, could lead to `NULL` mappings per row. String mappings for every field in the raw table will show you column misalignment failures such as these, very quickly. This is just a matter of preference, and only applies to tables designed for holding the raw or dirty input for immediate validation and transformation into other Hive tables.

Type casing values using the AS keyword

This recipe only outputs strings from the Python script for use over standard output. Hive will attempt to cast them to the appropriate type in the receiving table. The advantage to this is the time and coding space saved by not having to explicitly cast every field with the `AS` operator. The disadvantage is that this will not fail should a value be cast to an incompatible type. For instance, outputting `HI THERE` to a numeric field will insert `NULL` for the field value for that row. This can lead to undesirable behavior for subsequent `SELECT` statements over the table.

Testing the script locally

This one is pretty self-explanatory. It is much easier to debug your script directly on the command line than it is across MapReduce task error logs. It likely will not prevent you from having to troubleshoot issues dealing with scale or data validity, but it will eliminate a large majority of the compile time and control flow issues.

Using Python and Hadoop Streaming to perform a time series analytic

This recipe shows how to use Hadoop Streaming with Python to perform a basic time series analysis over the cleansed ACLED Nigeria dataset. The program is designed to output a list of dates in sorted order for each location where the government in Nigeria regained territory.

For this recipe, we will use structured Nigerian conflict data provided by Armed Conflict Location and Event dataset collections team.

Getting ready

You will need to download/compile/install the following:

- Version 0.7.1 of Apache Pig from `http://hive.apache.org/`
- Test data: download `Nigeria_ACLED_cleaned.tsv` from `http://www.packtpub.com/support` and place the file into HDFS
- Python 2.6 or greater

How to do it...

The following are the steps to use Python with Hadoop Streaming:

1. Create a shell script named `run_location_regains.sh` that runs the Streaming job. It is important to change the streaming JAR path to match the absolute path of your `hadoop-streaming.jar` file. The path of the `hadoop-streaming.jar` file is different depending on the Hadoop distribution:

```
#!/bin/bash
$HADOOP_HOME/bin/hadoop jar $HADOOP_HOME/contrib/streaming/hadoop-streaming-0.20.2-cdh3u1.jar \
    -input /input/acled_cleaned/Nigeria_ACLED_cleaned.tsv \
    -output /output/acled_analytic_out \
    -mapper location_regains_mapper.py \
    -reducer location_regains_by_time.py \
    -file location_regains_by_time.py \
```

```
    -file location_regains_mapper.py \
    -jobconf stream.num.map.output.key.fields=2 \
    -jobconf map.output.key.field.separator=\t \
    -jobconf num.key.fields.for.partition=1 \
    -jobconf mapred.reduce.tasks=1
```

2. Create a Python file named `location_regains_mapper.py` and add the following:

```
#!/usr/bin/python
import sys

for line in sys.stdin:
    (loc, event_date, event_type, actor, lat, lon, src,
fatalities) = line.strip().split('\t');
    (day,month,year) = event_date.split('/')
    if len(day) == 1:
        day = '0' + day
    if len(month) == 1:
        month = '0' + month;
    if len(year) == 2:
        if int(year) > 30 and int(year) < 99:
            year = '19' + year
        else:
            year = '20' + year
    event_date = year + '-' + month + '-' + day
    print '\t'.join([loc, event_date, event_type]);
```

3. Create a Python file named `location_regains_by_time.py` and add the following:

```
#!/usr/bin/python
import sys

current_loc = "START_OF_APP"
govt_regains=[]
for line in sys.stdin:
  (loc,event_date,event_type) = line.strip('\n').split('\t')
  if loc != current_loc:
    if current_loc != "START_OF_APP":
      print current_loc + '\t' + '\t'.join(govt_regains)
    current_loc = loc
    govt_regains = []
  if event_type.find('regains') != -1:
      govt_regains.append(event_date)
```

4. Run the shell script from the local working directory, which should contain all of the Python scripts that we created previously:

```
./run_location_regains.sh
```

You should see the job start from the command line and finish successfully:

```
INFO streaming.StreamJob: Output: /output/acled_analytic_out
```

How it works...

The shell script sets up the Hadoop Streaming JAR path and passes the necessary arguments to the program. Each argument is explained in detail as follows:

Argument	Description
`-input /input/acled_cleaned/ Nigeria_ACLED_cleaned.tsv \`	The HDFS path to the input data for MapReduce.
`-output /output/acled_analytic_ out \`	The HDFS path for MapReduce to write the job output.
`-mapper location_regains_mapper. py \`	Script to be run as the map function; records passed via `STDIN/STDOUT`.
`-reducer location_regains_by_ time.py \`	Script to be run as the reduce function.
`-file location_regains_by_time. py \`	Add a file to the distributed cache. This is required for external scripts.
`-file location_regains_mapper. py \`	Add a file to the distributed cache.
`-jobconf stream.num.map.output. key.fields=2 \`	Tells the streaming tool which field/fields should be treated as the map output key. Our mapper outputs three fields per record. This parameter tells the program to treat the first two as the key. This will leverage the secondary sort feature in MapReduce to sort our rows based on the composite of these two fields.
`-jobconf map.output.key.field. separator=\t \`	Parameter for setting the delimiter token on the key.
`-jobconf num.key.fields.for. partition=1 \`	Guarantees that all of the map output records with the same value in the first field of the key are sent to the same reducer.
`-jobconf mapred.reduce.tasks=1`	Number of JVM tasks to reduce over the output keys.

The Python script used in the map phase gets a `line` corresponding to each record. We call `strip()` to remove any leading/trailing whitespace and then split the line on tabs. The result is an array of variables descriptively named to the row fields they hold.

The `event_date` field in the raw input requires some processing. In order for the framework to sort records in ascending order of dates, we want to take the current form, which is dd/mm/yy and convert it to yyyy-mm-dd. Since some of the events occurred before the year 2000, we need to expand the year variable out to four digits. Single-digit days and months are zero-padded, so that it sorts correctly.

This analytics only requires `location`, `event_date`, and `event_type` to be output to the reduce stage. In the shell script, we specified the first two fields as the output key. Specifying `location` as the first field groups all records with the same location on a common reducer. Specifying `event_date` as the second field allows the MapReduce framework to sort the records by the composite of `location` and `event_date`. The value in each key-value pair is simply of the `event_type` field.

Sample map output:

```
(cityA, 2010-08-09, explosion)
(cityB, 2008-10-10, fire)
(cityA, 2009-07-03, riots)
```

Order reducer shows the records that are sorted on the composite value of `location` and `event_date`

```
(cityA, 2009-07-03, riots)
(cityA, 2010-08-09,explosion)
(cityB, 2008-10-10,fire)
```

Our configuration specifies only one reducer, so in this recipe all of the rows will partition to the same reduce Java Virtual Machine (JVM). If multiple reduce tasks are specified, `cityA` and `cityB` could be processed independently on separate reduce JVMs.

Understanding how the MapReduce framework sorts and handles the output of the `location_regains_mapper.py` file is important to determine how the reduce script works.

We use `location_regains_by_time.py` to iterate over the sorted collection of events per location, and aggregate events that match a particular type.

As the records were partitioned by location, we can assume that each partition will go to its own mapper. Furthermore, because we specified `event_date` as an additional sort column, we can make the assumption that the events corresponding to a given location are sorted by date in the ascending order. Now we are in a position to understand how the script works.

The script must keep a track of when a `loc` input changes from the previous location. Such a change signifies that we are done processing the previous location, since they are all in sorted order. We initialize the `current_loc` flag to `START_OF_APP`. We also declare an empty array `govt_regains` to hold the dates of events we are interested in.

The program starts by processing each line into variables. If there is a change in `loc` and it is not the beginning of the application, we know to output the current `govt_regains` collection to standard out. The change means that we are done processing the previous location, and can safely write its collection of event dates out of the reducer.

If the incoming `loc` value is the same as `current_loc`, we know that the incoming event still corresponds to the location we are currently processing. We check to see if the event is of the type `regains` to show the government the regained territories in that region. If it matches that type, we add it to the current `govt_regains` collection. Since the incoming records are sorted by `event_date`, we are guaranteed that the records are inserted in `govt_regains` in the ascending order of dates.

The net result is a single part file that is output from the reducer with a list of locations in lexicographically sorted order. To the right-hand side of each location is a tab-separated sorted list of dates matching the occurrences of when the government regained territory in that location.

There's more...

Hadoop Streaming is a very popular component. The following are a few important additions to know:

Using Hadoop Streaming with any language that can read from stdin and write to stdout

You are not limited to just Python when working with Hadoop Streaming. Java classes, shell scripts, ruby scripts, and many other languages are frequently used to transition existing code and functionality into full-fledged MapReduce programs. Any language that can read stdin and write to stdout will work with Hadoop Streaming.

Using the –file parameter to pass additional required files for MapReduce jobs

Similar to normal MapReduce programs, you can pass additional dependencies over the distributed cache to be used in your applications. Simply add additional `–file` parameters. For example:

```
-file mapper.py \
-file wordlist.txt
```

Using MultipleOutputs in MapReduce to name output files

A common request among MapReduce users is to control output file names to something other than `part-*`. This recipe shows how you can use the `MultipleOutputs` class to emit different key-value pairs to the same named file that you chose.

Getting ready

You will need to download the `ip-to-country.txt` dataset from the Packt website, `http://www.packtpub.com/support`, and place the file in HDFS.

How to do it...

Follow these steps to use `MultipleOutputs`:

1. Create a class named `NamedCountryOutputJob` and configure the MapReduce job:

```
import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.fs.Path;
import org.apache.hadoop.io.IntWritable;
import org.apache.hadoop.io.LongWritable;
import org.apache.hadoop.io.NullWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Job;
import org.apache.hadoop.mapreduce.Mapper;
import org.apache.hadoop.mapreduce.Reducer;
import org.apache.hadoop.mapreduce.lib.input.FileInputFormat;
import org.apache.hadoop.mapreduce.lib.input.TextInputFormat;
import org.apache.hadoop.mapreduce.lib.output.FileOutputFormat;
import org.apache.hadoop.mapreduce.lib.output.MultipleOutputs;
import org.apache.hadoop.mapreduce.lib.output.TextOutputFormat;
import org.apache.hadoop.util.Tool;
import org.apache.hadoop.util.ToolRunner;

import java.io.IOException;
import java.util.regex.Pattern;


public class NamedCountryOutputJob implements Tool{
```

```
    private Configuration conf;
    public static final String NAME = "named_output";


    public static void main(String[] args) throws Exception {
        ToolRunner.run(new Configuration(), new
NamedCountryOutputJob(), args);
    }
    public int run(String[] args) throws Exception {
        if(args.length != 2) {
            System.err.println("Usage: named_output <input>
<output>");
            System.exit(1);
        }

        Job job = new Job(conf, "IP count by country to named
files");
        job.setInputFormatClass(TextInputFormat.class);

        job.setMapperClass(IPCountryMapper.class);
        job.setReducerClass(IPCountryReducer.class);

        job.setMapOutputKeyClass(Text.class);
        job.setMapOutputValueClass(IntWritable.class);
        job.setJarByClass(NamedCountryOutputJob.class);

        FileInputFormat.addInputPath(job, new Path(args[0]));
        FileOutputFormat.setOutputPath(job, new Path(args[1]));

        return job.waitForCompletion(true) ? 1 : 0;

    }

    public void setConf(Configuration conf) {
        this.conf = conf;
    }

    public Configuration getConf() {
        return conf;
    }
```

2. Create a mapper to emit the key-value pair `country`, and the number `1`:

```
public static class IPCountryMapper
            extends Mapper<LongWritable, Text, Text, IntWritable>
{

        private static final int country_pos = 1;
        private static final Pattern pattern = Pattern.
compile("\\t");

        @Override
        protected void map(LongWritable key, Text value,
                           Context context)    throws IOException,
InterruptedException {
            String country = pattern.split(value.toString())
[country_pos];
            context.write(new Text(country), new IntWritable(1));
        }
    }
```

3. Create a reducer that sums all of the `country` counts, and writes the output to separate files using `MultipleOutputs`:

```
public static class IPCountryReducer
            extends Reducer<Text, IntWritable, Text, IntWritable>
{

        private MultipleOutputs output;

        @Override
        protected void setup(Context context
        ) throws IOException, InterruptedException {
            output = new MultipleOutputs(context);
        }


        @Override
        protected void reduce(Text key, Iterable<IntWritable>
values, Context context) throws IOException, InterruptedException
{
            int total = 0;
            for(IntWritable value: values) {
                total += value.get();
            }
           output.write(new Text("Output by MultipleOutputs"),
```

```
                            NullWritable.get(), key.toString());
                output.write(key, new IntWritable(total), key.
toString());
            }

            @Override
            protected void cleanup(Context context
            ) throws IOException, InterruptedException {
                output.close();
            }
        }
```

Once the job completes successfully, you should see named output files under the provided output directory (for example, `Qatar-r-#####`, `Turkey-r-#####`).

How it works...

We first set up our job using the `Tool` interface provided by Hadoop. The `run()` method inside `NamedCountryOutputJob` checks that both input and output HDFS path directories are provided. In addition, both the mapper and reducer classes are set, and we configure the InputFormat to read lines of text.

The mapper class defines a statically initialized position to read the country from each line, as well as the regex pattern to split each line. The mapper will output the country as the key and `1` for every line it appears on.

At the reduce phase, each task JVM runs the `setup()` routine and initializes a `MultipleOutputs` instance named `output`.

Each call to `reduce()` presents a country and a tally of every occurrence of the country appearing in the dataset. We sum the tally into a final count. Before we emit the final count, we will use the output instance to write a header to the file. The key contains the text for the header `Output by MultipleOutputs`, and we null out the value since we don't need it. We specify `key.toString()` to write the header to a custom file named by the current country. On the next line we call `output.write()` again, except this time with the input key as the output key, the final count as the output value, and the `key.toString()` method to specify the same output file as the previous `output.write()` method.

The end result is a named country file containing both the header and the final tallied count for that country.

By using `MultipleOutputs`, we don't have to configure an `OutputFormat` class in our job setup routine. Also, we are not limited to just one concrete type for the reducer output key and value. We were able to output key-value pairs for both `Text/NullWritable` and `Text/IntWritable` to the exact same file.

Creating custom Hadoop Writable and InputFormat to read geographical event data

When reading input, or writing output from a MapReduce application, it is sometimes easier to work with data using an abstract class instead of the primitive Hadoop Writable classes (for example, `Text` and `IntWritable`). This recipe demonstrates how to create a custom Hadoop Writable and InputFormat that can be used by MapReduce applications.

Getting ready

You will need to download the `Nigeria_ACLED_cleaned.tsv` dataset from `http://www.packtpub.com/support` and place the file into HDFS.

How to do it...

Follow these steps to create custom InputFormat and Writable classes:

1. First we will define two custom `WritableComparable` classes. These classes represent the key-value pairs that are passed to the mapper, much as how `TextInputFormat` passes `LongWritable` and `Text` to the mapper.

   Write the key class:

```
public class GeoKey implements WritableComparable {
    private Text location;
    private FloatWritable latitude;
    private FloatWritable longitude;
    public GeoKey() {
        location = null;
        latitude = null;
        longitude = null;
    }

    public GeoKey(Text location, FloatWritable latitude,
      FloatWritable longitude) {
        this.location = location;
        this.latitude = latitude;
        this.longitude = longitude;
    }

    //...getters and setters
```

```
    public void readFields(DataInput di) throws IOException {
        if (location == null) {
            location = new Text();
        }
        if (latitude == null) {
            latitude = new FloatWritable();
        }
        if (longitude == null) {
            longitude = new FloatWritable();
        }
        location.readFields(di);
        latitude.readFields(di);
        longitude.readFields(di);
    }
    public int compareTo(Object o) {
        GeoKey other = (GeoKey)o;
        int cmp = location.compareTo(other.location);
        if (cmp != 0) {
            return cmp;
        }
        cmp = latitude.compareTo(other.latitude);
        if (cmp != 0) {
            return cmp;
        }
        return longitude.compareTo(other.longitude);
    }

}
```

2. Now, the value class:

```
public class GeoValue implements WritableComparable {
    private Text eventDate;
    private Text eventType;
    private Text actor;
    private Text source;
    private IntWritable fatalities;

    public GeoValue() {
        eventDate = null;
        eventType = null;
        actor = null;
        source = null;
        fatalities = null;
    }
```

```
    //...getters and setters

    public void write(DataOutput d) throws IOException {
        eventDate.write(d);
        eventType.write(d);
        actor.write(d);
        source.write(d);
        fatalities.write(d);
    }

    public void readFields(DataInput di) throws IOException {
        if (eventDate == null) {
            eventDate = new Text();
        }
        if (eventType == null) {
            eventType = new Text();
        }
        if (actor == null) {
            actor = new Text();
        }
        if (source == null) {
            source = new Text();
        }
        if (fatalities == null) {
            fatalities = new IntWritable();
        }
        eventDate.readFields(di);
        eventType.readFields(di);
        actor.readFields(di);
        source.readFields(di);
        fatalities.readFields(di);
    }

    public int compareTo(Object o) {
        GeoValue other = (GeoValue)o;
        int cmp = eventDate.compareTo(other.eventDate);
        if (cmp != 0) {
            return cmp;
        }
        cmp = eventType.compareTo(other.eventType);
        if (cmp != 0) {
            return cmp;
        }
        cmp = actor.compareTo(other.actor);
```

```
        if (cmp != 0) {
            return cmp;
        }
        cmp = source.compareTo(other.source);
        if (cmp != 0) {
            return cmp;
        }
        return fatalities.compareTo(other.fatalities);
    }

}
```

3. Next, we need to create an InputFormat to serialize the text from our input file and create the `GeoKey` and `GeoValue` instances. This input format extends the Hadoop `FileInputFormat` class and returns our own implementation of a RecordReader:

```
public class GeoInputFormat extends FileInputFormat<GeoKey,
GeoValue> {

    @Override
    public RecordReader<GeoKey, GeoValue>
createRecordReader(InputSplit split, TaskAttemptContext context) {
        return new GeoRecordReader();
    }

    @Override
    protected boolean isSplitable(JobContext context, Path file) {
        CompressionCodec codec =
                new CompressionCodecFactory(context.
getConfiguration()).getCodec(file);
        return codec == null;
    }
}
```

4. Now, create a RecordReader to read from the `Nigeria_ACLED_cleaned.tsv` dataset:

```
public class GeoRecordReader extends RecordReader<GeoKey,
GeoValue> {

    private GeoKey key;
    private GeoValue value;
    private LineRecordReader reader = new LineRecordReader();
    @Override
    public void initialize(InputSplit is, TaskAttemptContext tac)
throws IOException, InterruptedException {
        reader.initialize(is, tac);
```

```
    }

    @Override
    public boolean nextKeyValue() throws IOException,
InterruptedException {

        boolean gotNextKeyValue = reader.nextKeyValue();
        if(gotNextKeyValue) {
            if (key == null) {
                key = new GeoKey();
            }
            if (value == null) {
                value = new GeoValue();
            }
            Text line = reader.getCurrentValue();
            String[] tokens = line.toString().split("\t");
            key.setLocation(new Text(tokens[0]));
            key.setLatitude(new FloatWritable(Float.
parseFloat(tokens[4])));
            key.setLongitude(new FloatWritable(Float.
parseFloat(tokens[5])));

            value.setActor(new Text(tokens[3]));
            value.setEventDate(new Text(tokens[1]));
            value.setEventType(new Text(tokens[2]));
            try {
                value.setFatalities(new IntWritable(Integer.
parseInt(tokens[7])));
            } catch(NumberFormatException ex) {
                value.setFatalities(new IntWritable(0));
            }
            value.setSource(new Text(tokens[6]));
        }
        else {
            key = null;
            value = null;
        }
        return gotNextKeyValue;
    }

    @Override
    public GeoKey getCurrentKey() throws IOException,
InterruptedException {
        return key;
    }
```

```
    @Override
    public GeoValue getCurrentValue() throws IOException,
InterruptedException {
        return value;
    }
    @Override
    public float getProgress() throws IOException,
InterruptedException {
        return reader.getProgress();
    }
    @Override
    public void close() throws IOException {
        reader.close();
    }

}
```

5. Finally, create a simple map-only job to test the InputFormat:

```
public class GeoFilter extends Configured implements Tool {

    public static class GeoFilterMapper extends Mapper<GeoKey,
GeoValue, Text, IntWritable> {
        @Override
        protected void map(GeoKey key, GeoValue value, Context
context) throws IOException, InterruptedException {
            String location = key.getLocation().toString();
            if (location.toLowerCase().equals("aba")) {
                context.write(value.getActor(),
                  value.getFatalities());
            }
        }
    }

    public int run(String[] args) throws Exception {

        Path inputPath = new Path(args[0]);
        Path outputPath = new Path(args[1]);

        Configuration conf = getConf();
        Job geoJob = new Job(conf);
        geoJob.setNumReduceTasks(0);
        geoJob.setJobName("GeoFilter");
        geoJob.setJarByClass(getClass());
        geoJob.setMapperClass(GeoFilterMapper.class);
```

```
            geoJob.setMapOutputKeyClass(Text.class);
            geoJob.setMapOutputValueClass(IntWritable.class);
            geoJob.setInputFormatClass(GeoInputFormat.class);
            geoJob.setOutputFormatClass(TextOutputFormat.class);

            FileInputFormat.setInputPaths(geoJob, inputPath);
            FileOutputFormat.setOutputPath(geoJob, outputPath);

            if(geoJob.waitForCompletion(true)) {
                return 0;
            }
            return 1;
        }

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

How it works...

The first task was to define our own Hadoop key and value representations by implementing the `WritableComparable` interface. The `WritableComparable` interface allows us to create our own abstract types, which can be used as keys or values by the MapReduce framework.

Next, we created an InputFormat that inherits from the `FileInputFormat` class. The Hadoop `FileInputFormat` is the base class for all file-based InputFormats. The InputFormat takes care of managing the input files for a MapReduce job. Since we do not want to change the way in which our input files are split and distributed across the cluster, we only need to override two methods, `createRecordReader()` and `isSplitable()`.

The `isSplitable()` method is used to instruct the `FileInputFormat` class that it is acceptable to split up the input files if there is a codec available in the Hadoop environment to read and split the file. The `createRecordReader()` method is used to create a Hadoop RecordReader that processes individual file splits and generates a key-value pair for the mappers to process.

After the `GeoInputFormat` class was written, we wrote a RecordReader to process the individual input splits and create `GeoKey` and `GeoValue` for the mappers. The `GeoRecordReader` class reused the Hadoop `LineRecordReader` class to read from the input split. When the `LineRecordReader` class completed reading a record from the `Nigeria_ACLED_cleaned.tsv` dataset, we created two objects. These objects are `GeoKey` and `GeoValue`, which are sent to the mapper.