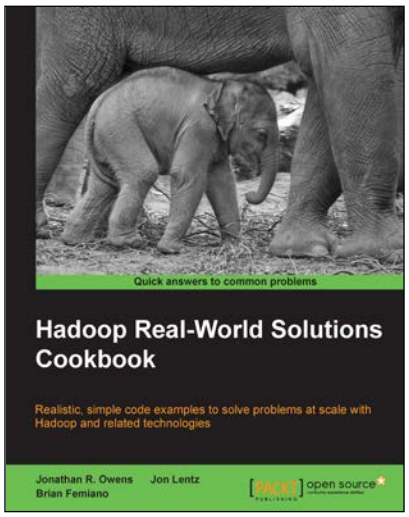
[PACKT] open source
community experience distilled
PUBLISHING

Hadoop Real-World Solutions Cookbook

Jonathan R. Owens

Jon Lentz

Brian Femiano



Chapter No. 6
"Big Data Analysis"

In this package, you will find:

A Biography of the authors of the book

A preview chapter from the book, Chapter NO.6 "Big Data Analysis"

A synopsis of the book’s content

Information on where to buy this book

About the Authors

Jonathan R. Owens has a background in Java and C++, and has worked in both private and public sectors as a software engineer. Most recently, he has been working with Hadoop and related distributed processing technologies.

Currently, he works for comScore, Inc., a widely regarded digital measurement and analytics company. At comScore, he is a member of the core processing team, which uses Hadoop and other custom distributed systems to aggregate, analyze, and manage over 40 billion transactions per day.

> I would like to thank my parents James and Patricia Owens, for their support and introducing me to technology at a young age.

Jon Lentz is a Software Engineer on the core processing team at comScore, Inc., an online audience measurement and analytics company. He prefers to do most of his coding in Pig. Before working at comScore, he developed software to optimize supply chains and allocate fixed-income securities.

> To my daughter, Emma, born during the writing of this book. Thanks for the company on late nights.

Brian Femiano has a B.S. in Computer Science and has been programming professionally for over 6 years, the last two of which have been spent building advanced analytics and Big Data capabilities using Apache Hadoop. He has worked for the commercial sector in the past, but the majority of his experience comes from the government contracting space. He currently works for Potomac Fusion in the DC/Virginia area, where they develop scalable algorithms to study and enhance some of the most advanced and complex datasets in the government space. Within Potomac Fusion, he has taught courses and conducted training sessions to help teach Apache Hadoop and related cloud-scale technologies.

I'd like to thank my co-authors for their patience and hard work building the code you see in this book. Also, my various colleagues at Potomac Fusion, whose talent and passion for building cutting-edge capability and promoting knowledge transfer have inspired me.

Hadoop Real-World Solutions Cookbook

Hadoop Real-World Solutions Cookbook helps developers become more comfortable with, and proficient at solving problems in, the Hadoop space. Readers will become more familiar with a wide variety of Hadoop-related tools and best practices for implementation.

This book will teach readers how to build solutions using tools such as Apache Hive, Pig, MapReduce, Mahout, Giraph, HDFS, Accumulo, Redis, and Ganglia.

This book provides in-depth explanations and code examples. Each chapter contains a set of recipes that pose, and then solve, technical challenges and that can be completed in any order. A recipe breaks a single problem down into discrete steps that are easy to follow. This book covers unloading/loading to and from HDFS, graph analytics with Giraph, batch data analysis using Hive, Pig, and MapReduce, machine-learning approaches with Mahout, debugging and troubleshooting MapReduce jobs, and columnar storage and retrieval of structured data using Apache Accumulo.

This book will give readers the examples they need to apply the Hadoop technology to their own problems.

What This Book Covers

Chapter 1, Hadoop Distributed File System – Importing and Exporting Data, shows several approaches for loading and unloading data from several popular databases that include MySQL, MongoDB, Greenplum, and MS SQL Server, among others, with the aid of tools such as Pig, Flume, and Sqoop.

Chapter 2, HDFS, includes recipes for reading and writing data to/from HDFS. It shows how to use different serialization libraries, including Avro, Thrift, and Protocol Buffers. Also covered is how to set the block size and replication, and enable LZO compression.

Chapter 3, Extracting and Transforming Data, includes recipes that show basic Hadoop ETL over several different types of data sources. Different tools, including Hive, Pig, and the Java MapReduce API, are used to batch-process data samples and produce one or more transformed outputs.

Chapter 4, Performing Common Tasks Using Hive, Pig, and MapReduce, focuses on how to leverage certain functionality in these tools to quickly tackle many different classes of problems. This includes string concatenation, external table mapping, simple table joins, custom functions, and dependency distribution across the cluster.

Chapter 5, Advanced Joins, contains recipes that demonstrate more complex and useful join techniques in MapReduce, Hive, and Pig. These recipes show merged, replicated, and skewed joins in Pig as well as Hive map-side and full outer joins. There is also a recipe that shows how to use Redis to join data from an external data store.

Chapter 6, Big Data Analysis, contains recipes designed to show how you can put Hadoop to use to answer different questions about your data. Several of the Hive examples will demonstrate how to properly implement and use a custom function (UDF) for reuse in different analytics. There are two Pig recipes that show different analytics with the Audioscrobbler dataset and one MapReduce Java API recipe that shows Combiners.

Chapter 7, Advanced Big Data Analysis, shows recipes in Apache Giraph and Mahout that tackle different types of graph analytics and machine-learning challenges.

Chapter 8, Debugging, includes recipes designed to aid in the troubleshooting and testing of MapReduce jobs. There are examples that use MRUnit and local mode for ease of testing. There are also recipes that emphasize the importance of using counters and updating task status to help monitor the MapReduce job.

Chapter 9, System Administration, focuses mainly on how to performance-tune and optimize the different settings available in Hadoop. Several different topics are covered, including basic setup, XML configuration tuning, troubleshooting bad data nodes, handling NameNode failure, and performance monitoring using Ganglia.

Chapter 10, Persistence Using Apache Accumulo, contains recipes that show off many of the unique features and capabilities that come with using the NoSQL datastore Apache Accumulo. The recipes leverage many of its unique features, including iterators, combiners, scan authorizations, and constraints. There are also examples for building an efficient geospatial row key and performing batch analysis using MapReduce.

6
Big Data Analysis

In this chapter, we will cover:

- Counting distinct IPs in weblog data using MapReduce and Combiners
- Using Hive date UDFs to transform and sort event dates from geographic event data
- Using Hive to build a per-month report of fatalities over geographic event data
- Implementing a custom UDF in Hive to help validate source reliability over geographic event data
- Marking the longest period of non-violence using Hive MAP/REDUCE operators and Python
- Calculating the cosine similarity of Artists in the Audioscrobbler dataset using Pig
- Trim outliers from the Audioscrobbler dataset using Pig and datafu

Introduction

Learning to apply Apache Hive, Pig, and MapReduce to solve the specific problems you are faced with can be difficult. The recipes in this chapter present a few big data problems and provide solutions that show how to tackle them. You will notice that the questions we ask of the data are not incredibly complicated, but you will require a different approach when dealing with a large volume of data. Even though the sample datasets in the recipes are small, you will find that the code is still very applicable to bigger problem spaces distributed over large Hadoop clusters.

The analytic questions in this chapter are designed to highlight many of the more powerful features of the various tools. You will find many of these features and operators useful as you begin solving your own problems.

Counting distinct IPs in weblog data using MapReduce and Combiners

This recipe will walk you through creating a MapReduce program to count distinct IPs in weblog data. We will demonstrate the application of a combiner to optimize data transfer overhead between the map and reduce stages. The code is implemented in a generic fashion and can be used to count distinct values in any tab-delimited dataset.

Getting ready

This recipe assumes that you have a basic familiarity with the Hadoop 0.20 MapReduce API. You will need access to the `weblog_entries` dataset supplied with this book and stored in an HDFS folder at the path `/input/weblog`.

You will need access to a pseudo-distributed or fully-distributed cluster capable of running MapReduce jobs using the newer MapReduce API introduced in Hadoop 0.20.

You will also need to package this code inside a JAR file to be executed by the Hadoop JAR launcher from the shell. Only the core Hadoop libraries are required to compile and run this example.

How to do it...

Perform the following steps to count distinct IPs using MapReduce:

1. Open a text editor/IDE of your choice, preferably one with Java syntax highlighting.
2. Create a class named `DistinctCounterJob.java` in your JAR file at whatever source package is appropriate.
3. The following code will serve as the Tool implementation for job submission:

```
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.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.TextOutputFormat;
import org.apache.hadoop.util.Tool;
import org.apache.hadoop.util.ToolRunner;
```

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


public class DistinctCounterJob implements Tool {

    private Configuration conf;
    public static final String NAME = "distinct_counter";
    public static final String COL_POS = "col_pos";


    public static void main(String[] args) throws Exception {
        ToolRunner.run(new Configuration(),    new
DistinctCounterJob(), args);
    }
```

4. The `run()` method is where we set the input/output formats, mapper class configuration, combiner class, and key/value class configuration:

```
    public int run(String[] args) throws Exception {
        if(args.length != 3) {
            System.err.println("Usage: distinct_counter <input>
<output> <element_position>");
            System.exit(1);
        }
        conf.setInt(COL_POS, Integer.parseInt(args[2]));

        Job job = new Job(conf, "Count distinct elements at
position");
        job.setInputFormatClass(TextInputFormat.class);
        job.setOutputFormatClass(TextOutputFormat.class);

        job.setMapperClass(DistinctMapper.class);
        job.setReducerClass(DistinctReducer.class);
        job.setCombinerClass(DistinctReducer.class);

        job.setMapOutputKeyClass(Text.class);
        job.setMapOutputValueClass(IntWritable.class);
        job.setJarByClass(DistinctCounterJob.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;
    }
}
```

5. The `map()` function is implemented in the following code by extending `mapreduce.Mapper`:

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

        private static int col_pos;
        private static final Pattern pattern = Pattern.
compile("\\t");
        private Text outKey = new Text();
        private static final IntWritable outValue = new
IntWritable(1);

        @Override
        protected void setup(Context context
        ) throws IOException, InterruptedException {
            col_pos = context.getConfiguration().
getInt(DistinctCounterJob.COL_POS, 0);
        }

        @Override
        protected void map(LongWritable key, Text value,
                           Context context) throws IOException,
InterruptedException {
            String field = pattern.split(value.toString())[col_
pos];
            outKey.set(field);
            context.write(outKey, outValue);
        }
    }
```

6. The `reduce()` function is implemented in the following code by extending `mapreduce.Reducer`:

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

        private IntWritable count = new IntWritable();

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

7. The following command shows the sample usage against weblog data with column position number `4`, which is the IP column:

```
hadoop jar myJobs.jar distinct_counter /input/weblog/ /output/
weblog_distinct_counter 4
```

How it works...

First we set up `DistinctCounterJob` to implement a Tool interface for remote submission. The static constant `NAME` is of potential use in the Hadoop `Driver` class, which supports the launching of different jobs from the same JAR file. The static constant `COL_POS` is initialized to the third required argument from the command line `<element_position>`. This value is set within the job configuration, and should match the position of the column you wish to count for each distinct entry. Supplying `4` will match the IP column for the weblog data.

Since we are reading and writing text, we can use the supplied `TextInputFormat` and `TextOutputFormat` classes. We will set the `Mapper` and `Reduce` classes to match our `DistinctMapper` and `DistinctReducer` implemented classes respectively. We also supply `DistinctReducer` as a combiner class. This decision is explained in more detail as follows:

It's also very important to call `setJarByClass()` so that the `TaskTrackers` can properly unpack and find the `Mapper` and `Reducer` classes. The job uses the static helper methods on `FileInputFormat` and `FileOutputFormat` to set the input and output directories respectively. Now we're set up and ready to submit the job.

The `Mapper` class sets up a few member variables as follows:

- `col_pos`: This is initialized to a value supplied in the configuration. It allows users to change which column to parse and apply the count distinct operation on.
- `pattern`: This defines the column's split point for each row based on tabs.
- `outKey`: This is a class member that holds output values. This avoids having to create a new instance for each output that is written.
- `outValue`: This is an integer representing one occurrence of the given key. It is similar to the WordCount example.

The `map()` function splits each incoming line's value and extracts the string located at `col_pos`. We reset the internal value for `outKey` to the string found on that line's position. For our example, this will be the IP value for the row. We emit the value of the newly reset `outKey` variable along with the value of `outValue` to mark one occurrence of that given IP address.

Without the assistance of the combiner, this would present the reducer with an iterable collection of 1s to be counted.

The following is an example of a reducer {key, value:[]} without a combiner:

{10.10.1.1, [1,1,1,1,1,1]} = six occurrences of the IP "10.10.1.1".

The implementation of the `reduce()` method will sum the integers and arrive at the correct total, but there's nothing that requires the integer values to be limited to the number 1. We can use a combiner to process the intermediate key-value pairs as they are output from each mapper and help improve the data throughput in the shuffle phase. Since the combiner is applied against the local map output, we may see a performance improvement as the amount of data we need to transfer for an intermediate key/value can be reduced considerably.

Instead of seeing {10.10.1.1, [1,1,1,1,1,1]}, the combiner can add the 1s and replace the value of the intermediate value for that key to {10.10.1.1, [6]}. The reducer can then sum the various combined values for the intermediate key and arrive at the same correct total. This is possible because addition is both a commutative and associative operation. In other words:

- **Commutative**: The order in which we process the addition operation against the values has no effect on the final result. For example, 1 + 2 + 3 = 3 + 1 + 2.
- **Associative**: The order in which we apply the addition operation has no effect on the final result. For example, (1 + 2) + 3 = 1 + (2 + 3).

For counting the occurrences of distinct IPs, we can use the same code in our reducer as a combiner for output in the map phase.

When applied to our problem, the normal output with no combiner from two separate independently running map tasks might look like the following where {key: value[]} is equal to the intermediate key-value collection:

- Map Task A = {10.10.1.1, [1,1,1]} = three occurrences
- Map Task B = {10.10.1.1, [1,1,1,1,1,1]} = six occurrences

Without the aid of a combiner, this will be merged in the shuffle phase and presented to a single reducer as the following key-value collection:

- {10.10.1.1, [1,1,1,1,1,1,1,1,1]} = nine total occurrences

Now let's revisit what would happen when using a Combiner against the exact same sample output:

Map Task A = {10.10.1.1, [1,1,1]} = three occurrences

- Combiner = {10.10,1,1, [3] = still three occurrences, but reduced for this mapper.

Map Task B = {10.10.1.1, [1,1,1,1,1,1] = six occurrences

- Combiner = {10.10.1.1, [6] = still six occurrences

Now the reducer will see the following for that key-value collection:

- {10.10.1.1, [3,6]} = nine total occurrences

We arrived at the same total count for that IP address, but we used a combiner to limit the amount of network I/O during the MapReduce shuffle phase by pre-reducing the intermediate key-value output from each mapper.

There's more...

The combiner can be confusing to newcomers. Here are some useful tips:

The Combiner does not always have to be the same class as your Reducer

The previous recipe and the default WordCount example show the `Combiner` class being initialized to the same implementation as the `Reducer` class. This is not enforced by the API, but ends up being common for many types of distributed aggregate operations such as `sum()`, `min()`, and `max()`. One basic example might be the `min()` operation of the `Reducer` class that specifically formats output in a certain way for readability. This will take a slightly different form from that of the `min()` operator of the `Combiner` class, which does not care about the specific output formatting.

Combiners are not guaranteed to run

Whether or not the framework invokes your combiner during execution depends on the intermediate spill file size from each map output, and is not guaranteed to run for every intermediate key. Your job should not depend on the combiner for correct results, it should be used only for optimization.

You can control the spill file threshold when MapReduce tries to combine intermediate values with the configuration property `min.num.spills.for.combine`.

Using Hive date UDFs to transform and sort event dates from geographic event data

This recipe will illustrate the efficient use of the Hive date UDFs to list the 20 most recent events and the number of days between the event date and the current system date.

Getting ready

Make sure you have access to a pseudo-distributed or fully-distributed Hadoop cluster with Apache Hive 0.7.1 installed on your client machine and on the environment path for the active user account.

This recipe depends on having the `Nigera_ACLED_cleaned.tsv` dataset loaded into a Hive table named `acled_nigeria_cleaned` with the fields mapped to the respective datatypes.

Issue the following command to the Hive client to see the mentioned fields:

```
describe acled_nigeria_cleaned
```

You should see the following response:

```
OK
Loc  string
event_date  string
event_type  string
actor  string
latitude  double
longitude  double
source  string
fatalities  int
```

How to do it...

Perform the following steps to utilize Hive UDFs for sorting and transformation:

1. Open a text editor of your choice, ideally one with SQL syntax highlighting.
2. Add the inline creation and transform syntax:

```
SELECT event_type,event_date,days_since FROM (
    SELECT event_type,event_date,
            datediff(to_date(from_unixtime(unix_timestamp())),
                     to_date(from_unixtime(
                             unix_timestamp(event_date,
                            'yyyy-MM-dd')))) AS days_since
     FROM acled_nigeria_cleaned) date_differences
     ORDER BY event_date DESC LIMIT 20;
```

3. Save the file as `top_20_recent_events.sql` in the active folder.
4. Run the script from the operating system shell by supplying the `-f` option to the Hive client. You should see the following five rows appear first in the output console:

```
OK
Battle-No change of territory  2011-12-31    190
Violence against civilians  2011-12-27    194
Violence against civilians  2011-12-25    196
Violence against civilians  2011-12-25    196
Violence against civilians  2011-12-25    196
```

How it works...

Let's start with the nested `SELECT` subqueries. We select three fields from our Hive table `acled_nigeria_cleaned`: `event_type`, `event_date`, and the result of calling the UDF `datediff()`, which takes as arguments an end date and a start date. Both are expected in the form yyyy-MM-dd. The first argument to `datediff()` is the end date, with which we want to represent the current system date. Calling `unix_timestamp()` with no arguments will return the current system time in milliseconds. We send that return value to `from_unixtimestamp()` to get a formatted timestamp representing the current system date in the default Java 1.6 format (yyyy-MM-dd HH:mm:ss). We only care about the date portion, so calling `to_date()` with the output of this function strips the HH:mm:ss. The result is the current date in the yyyy-MM-dd form.

The second argument to `datediff()` is the start date, which for our query is the `event_date`. The series of function calls operate in almost the exact same manner as our previous argument, except that when we call `unix_timestamp()`, we must tell the function that our argument is in the `SimpleDateFormat` format that is yyyy-MM-dd. Now we have both `start_date` and `end_date` arguments in the yyyy-MM-dd format and can perform the `datediff()` operation for the given row. We alias the output column of `datediff()` as `days_since` for each row.

The outer `SELECT` statement takes these three columns per row and sorts the entire output by `event_date` in descending order to get reverse chronological ordering. We arbitrarily limit the output to only the first 20.

The net result is the 20 most recent events with the number of days that have passed since that event occurred.

There's more...

The date UDFs can help tremendously in performing string date comparisons. Here are some additional pointers:

Date format strings follow Java SimpleDateFormat guidelines

Check out the Javadocs for `SimpleDateFormat` to learn how your custom date strings can be used with the date transform UDFs.

Default date and time formats

- Many of the UDFs operate under a default format assumption.
- For UDFs requiring only date, your column values must be in the form yyyy-MM-dd.
- For UDFs that require date and time, your column values must be in the form yyyy-MM-dd HH:mm:ss.

See also

- *Using Hive to build a per-month report of fatalities over geographic event data*

Using Hive to build a per-month report of fatalities over geographic event data

This recipe will show a very simple analytic that uses Hive to count fatalities for every month appearing in the dataset and print the results to the console.

Getting ready

Make sure you have access to a pseudo-distributed or fully-distributed Hadoop cluster with Apache Hive 0.7.1 installed on your client machine and on the environment path for the active user account.

This recipe depends on having the `Nigera_ACLED_cleaned.tsv` dataset loaded into a Hive table named `acled_nigeria_cleaned` with the following fields mapped to the respective datatypes.

Issue the following command to the Hive client:

```
describe acled_nigeria_cleaned
```

You should see the following response:

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

How to do it...

Follow the steps to use Hive for report generation:

1. Open a text editor of your choice, ideally one with SQL syntax highlighting.

2. Add the inline creation and transformation syntax:

```
SELECT from_unixtime(unix_timestamp(event_date,  'yyyy-MM-dd'),
'yyyy-MMM'),
       COALESCE(CAST(sum(fatalities) AS STRING), 'Unknown')
       FROM acled_nigeria_cleaned
       GROUP BY from_unixtime(unix_timestamp(event_date, 'yyyy-MM-
dd'),'yyyy-MMM');
```

3. Save the file as `monthly_violence_totals.sql` in the active folder.
4. Run the script from the operating system shell by supplying the `-f` option to the Hive client. You should see the following three rows appear first in the output console. Note that the output is sorted lexicographically, and not on the order of dates.

```
OK
1997-Apr  115
1997-Aug  4
1997-Dec  26
```

How it works...

The `SELECT` statement uses `unix_timestamp()` and `from_unixtime()` to reformat the `event_date` for each row as just a year-month concatenated field. This is also in the `GROUP BY` expression for totaling fatalities using `sum()`.

The `coalesce()` method returns the first non-null argument passed to it. We pass as the first argument, the value of fatalities summed for that given year-month, cast as a string. If that value is `NULL` for any reason, return the constant `Unknown`. Otherwise return the string representing the total fatalities counted for that year-month combination. Print everything to the console over `stdout`.

There's more...

The following are some additional helpful tips related to the code in this recipe:

The coalesce() method can take variable length arguments.

As mentioned in the Hive documentation, `coalesce()` supports one or more arguments. The first non-null argument will be returned. This can be useful for evaluating several different expressions for a given column before deciding the right one to choose.

The `coalesce()` will return `NULL` if no argument is non-null. It's not uncommon to provide a type literal to return if all other arguments are `NULL`.

Date reformatting code template

Having to reformat dates stored in your raw data is very common. Proper use of `from_unixtime()` and `unix_timestamp()` can make your life much easier.

Remember this general code template for concise date format transformation in Hive:

```
from_unixtime(unix_timestamp(<col>,<in-format>),<out-format>);
```

See also

- *Using Hive date UDFs to transform and sort event dates from geographic event data*

Implementing a custom UDF in Hive to help validate source reliability over geographic event data

There are many operations you will want to repeat across various data sources and tables in Hive. For this scenario, it makes sense to write your own user-defined function (UDF). You can write your own subroutine in Java for use on any Writable input fields and to invoke your function from Hive scripts whenever necessary. This recipe will walk you through the process of creating a very simple UDF that takes a source and returns `yes` or `no` for whether that source is reliable.

Getting ready

Make sure you have access to a pseudo-distributed or fully-distributed Hadoop cluster with Apache Hive 0.7.1 installed on your client machine and on the environment path for the active user account.

This recipe depends on having the `Nigera_ACLED_cleaned.tsv` dataset loaded into a Hive table with the name `acled_nigeria_cleaned` with the following fields mapped to the respective datatypes.

Issue the following command to the Hive client:

```
describe acled_nigeria_cleaned;
```

You should see the following response:

```
OK
loc  string
```

```
event_date  string
event_type  string
actor  string
latitude  double
longitude  double
source string
fatalities  int
```

Additionally, you will need to place the following recipe's code into a source package for bundling within a JAR file of your choice. This recipe will use `<myUDFs.jar>` as a reference point for your custom JAR file and `<fully_qualified_path_to_TrustSourceUDF>` as a reference point for the Java package your class exists within. An example of a fully qualified path for a pattern would be `java.util.regex.Pattern`.

In addition to the core Hadoop libraries, your project will need to have `hive-exec` and `hive-common` JAR dependencies on the classpath for this to compile.

How to do it...

Perform the following steps to implement a custom Hive UDF:

1. Open a text editor/IDE of your choice, preferably one with Java syntax highlighting.
2. Create `TrustSourceUDF.java` at the desired source package. Your class should exist at some package `<fully_qualified_path>.TrustSourceUDF.class`.
3. Enter the following source as the implementation for the `TrustSourceUDF` class:

```
import org.apache.hadoop.hive.ql.exec.UDF;
import org.apache.hadoop.io.Text;
import java.lang.String;import java.util.HashSet;
import java.util.Set;


public class TrustSourceUDF extends UDF {

    private static Set<String> untrustworthySources = new
HashSet<String>();
    private Text result = new Text();

    static {
     untrustworthySources.add("");
     untrustworthySources.add("\"\"\"
     http://www.afriquenligne.fr/3-soldiers\"");
     untrustworthySources.add("Africa News Service");
```

```
        untrustworthySources.add("Asharq Alawsat");
        untrustworthySources.add("News Agency of Nigeria (NAN)");
        untrustworthySources.add("This Day (Nigeria)");
    }

    @Override
    public Text evaluate(Text source) {

        if(untrustworthySources.contains(source.toString())) {
            result.set("no");
        } else {
            result.set("yes");
        }
        return result;
    }
}
```

4. Build the containing JAR `<myUDFs.jar>` and test your UDF through the Hive client. Open a Hive client session through the command shell. Hive should already be on the local user environment path. Invoke the Hive shell with the following command:

   ```
   hive
   ```

5. Add the JAR file to the Hive session's classpath:

   ```
   add jar /path/to/<myUDFs.jar>;
   ```

   You will know that the preceding operation succeeded if you see the following messages indicating that the JAR has been added to the classpath and the distributed cache:

   ```
   Added /path/to/<myUDFs.jar> to class path
   Added resource: /path/to/<myUDFs.jar>
   ```

6. Create the function definition `trust_source` as an alias to `TrustSourceUDF` at whatever source package you specified in your JAR:

   ```
   create temporary function trust_source as '<fully_qualified_path_
   to_TrustSourceUDF>';
   ```

   You should see the shell prompt you that the command executed successfully. If you see the following error, it usually indicates your class was not found on the classpath:

   ```
   FAILED: Execution Error, return code 1 from org.apache.hadoop.
   hive.ql.exec.FunctionTask
   ```

7. Test the function with the following query. You should see mostly **yes** printed on each line of the console, with a few no's here and there:

   ```
   select trust_source(source) from acled_nigeria_cleaned;
   ```

How it works...

The class `TrustSourceUDF` extends UDF. No methods are required for implementation; however, in order for the class to function at Hive runtime as a UDF, your subclass must override `evaluate()`. You can have one or more overloaded `evaluate()` methods with different arguments. Ours only needs to take in a `source` value to check.

During class initialization, we set up a static instance of the `java.util.Set` class named `untrustworthySources`. Within a static initialization block, we set up a few sources by their names to be flagged as unreliable.

The entries here are purely arbitrary and should not be considered reliable or unreliable outside of this recipe.

We flag an empty source as unreliable.

When the function is invoked, it expects a single `Text` instance to be checked against the sources we've flagged as unreliable. Return `yes` or `no` depending on whether the given source appears in the set of unreliable sources or not. We set up the private `Text` instance to be re-used every time the function is called.

Once the JAR file containing the class is added to the classpath, and we set up our temporary function definition, we can now use the UDF across many different queries.

There's more...

User-defined functions are a very powerful feature within Hive. The following sections list a bit more information regarding them:

Check out the existing UDFs

The Hive documentation has a great explanation of the built-in UDFs bundled with the language. A great write up is available at `https://cwiki.apache.org/confluence/display/Hive/LanguageManual+UDF#LanguageManualUDF-BuiltinAggregateFunctions%28UDAF%29`.

To see which functions are available in your specific version of Hive, issue the following command in the Hive shell.

```
show functions;
```

Once you pinpoint a function that looks interesting, learn more information about it from the Hive wiki or directly from the Hive shell by executing the following command:

```
describe function <func>;
```

User-defined table and aggregate functions

Hive UDFs do not need to have a one-to-one interaction for input and output. The API allows the generation of many outputs from one input (GenericUDTF) as well as custom aggregate functions that take a list of input rows and output a single value (UDAF).

Export HIVE_AUX_JARS_PATH in your environment

Adding JAR files dynamically to the classpath is useful for testing and debugging, but can be cumbersome if you have many libraries you repeatedly wish to use. The Hive command line interpreter will automatically look for the existence of `HIVE_AUX_JARS_PATH` in the executing user's environment. Use this environment variable to set additional JAR paths that will always get loaded in the classpath of new Hive sessions for that client machine.

See also

- *Using Hive date UDFs to transform and sort event dates from geographic event data*
- *Using Hive to build a per-month report of fatalities over geographic event data*

Marking the longest period of non-violence using Hive MAP/REDUCE operators and Python

The Hive query language provides facilities to control the MapReduce dataflow and inject your own custom map, and to reduce scripts at each stage. When used properly, this is a very powerful technique for writing concise MapReduce programs using minimal syntax.

This recipe will show a complete example of how to write custom MapReduce control flow using different operators in Hive. The analytic will specifically look for the longest gap in events for each location to get an idea of how frequently violence occurs in that location.

Getting ready

Make sure you have access to a pseudo-distributed or fully-distributed Hadoop cluster with Apache Hive 0.7.1 installed on your client machine and on the environment path for the active user account.

Your cluster will also need Python 2.7 or greater installed on each node and available on the environment path for the Hadoop user. The script shown in this recipe assumes an installation at `/usr/bin/env python`. If this does not match your installation, change the script accordingly.

This recipe depends on having the `Nigeria_ACLED_cleaned.tsv` dataset loaded into a Hive table named `acled_nigeria_cleaned` with the following fields mapped to the respective datatypes.

Issue the following command to the Hive client:

```
describe acled_nigeria_cleaned;
```

You should see the following response:

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

How to do it...

Perform the following steps to mark the longest period of non-violence using Hive:

1. Open a text editor of your choice, ideally one with SQL and Python syntax highlighting.
2. Add the following inline creation and transform syntax:

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

DROP TABLE IF EXISTS longest_event_delta_per_loc;
CREATE TABLE longest_event_delta_per_loc (
    loc STRING,
    start_date STRING,
    end_date STRING,
    days INT
);

ADD FILE calc_longest_nonviolent_period.py;
FROM (
        SELECT loc, event_date, event_type
        FROM acled_nigeria_cleaned
        DISTRIBUTE BY loc SORT BY loc, event_date
     ) mapout
INSERT OVERWRITE TABLE longest_event_delta_per_loc
REDUCE mapout.loc, mapout.event_date, mapout.event_type
USING 'python calc_longest_nonviolent_period.py'
AS loc, start_date, end_date, days;
```

3. Save the file in the local working folder as `longest_nonviolent_periods_per_location.sql`.
4. Create a new file in your text editor with the name `calc_longest_nonviolent_period.py` and save it in the same working folder as `longest_nonviolent_periods_per_location.sql`.
5. Add the Python syntax. Python is sensitive to indentation. Keep that in mind if you are cutting and pasting this code:

```
#!/usr/bin/python
import sys
from datetime import datetime, timedelta

current_loc = "START_OF_APP"
(prev_date, start_date, end_date, start_time_obj, end_time_obj,
current_diff)=('', '', '', None, None, timedelta.min)
for line in sys.stdin:
  (loc,event_date,event_type) = line.strip('\n').split('\t')
  if loc != current_loc and current_loc != "START_OF_APP":
    if end_date != '':
      print '\t'.join([current_loc,start_date,event_
date,str(current_diff.days)])
          (prev_date, start_date, end_date, start_time_obj, end_
time_obj,current_diff)=('', '', '', None, None, timedelta.min)
    end_time_obj = datetime.strptime(event_date,'%Y-%m-%d')
  current_loc = loc
  if start_time_obj is not None: # implies > 2 events
    diff = end_time_obj - start_time_obj
    if diff > current_diff:
      current_diff = diff # set the current max time delta
      start_date = prev_date
      end_date = event_date
  prev_date = event_date
  start_time_obj = end_time_obj
```

6. Run the script from the operating system's shell by supplying the –f option to the Hive client:

```
hive –f longest_nonviolent_periods_per_location.sql
```

7. Issue the following query directly to the Hive shell. You should see rows printed to the console in no particular order:

```
hive –e "select * from longest_event_delta_per_loc;"
```

How it works...

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

Then we create our table definition for the output, dropping any existing tables with a matching name `longest_event_delta_per_loc`. The table requires four fields per record: `loc` for the location, `start_date` to hold the value of the `event_date` field of the lower bound, `end_date` to hold the value of `event_date` field of the upper bound, and `days` to show the total number of days elapsed between the events.

We then add the file `calc_longest_nonviolent_period.py` to the distributed cache for use across the different reducer JVMs. This will be used as our reduce script, but first we must organize the map output. The inner `SELECT` statement grabs `loc`, `event_date`, and `event_type` from the `acled_nigeria_cleaned` table in Hive. The `DISTRIBUTE BY loc` statement tells Hive to guarantee that all rows with matching values for `loc` go to the same reducer. `SORT BY loc, event_date` tells Hive to sort the data as it arrives to each reducer by the combination of `loc` and `event_date`. Now the same reducer can process every row corresponding to each location locally, and in the sorted order of `event_date`.

We alias the output of this `SELECT` statement to `mapout` and use the shorthand `REDUCE` operator to process each row from `mapout`. The `USING` clause lets us supply a custom Python script to read each record as it comes over `stdin`. The `AS` operator lets us map the delimited fields that are output by the script over `stdout` to pipe into the fields of the receiving table.

The Python script `calc_longest_nonviolent_period.py` will be used by the reduce stage to compute the longest time gap between the events for each location. Since we have guaranteed that all records with a common `loc` value are at the same reducer and that those records are in the date-sorted order for each location, we are now in a position to understand how the Python script works.

In the Python script `calc_longest_nonviolent_period.py`, we start with `#!/usr/bin/python` as a hint to the shell on how to execute the script. We need to import `sys` to use the `stdin` and `stdout` operations. We also need the `datetime` and `timedelta` class definitions from the `datetime` package.

The script operates very procedurally and can be a bit difficult to follow. First we declare `current_loc` and initialize its value to `START_OF_APP` as a flag to the print out conditional. We then set up several different variables to hold different placeholder values to be used on a per-location basis by the `for` loop.

- `prev_date`: This holds the last observed `event_date` for the `loc` value. It is blank if it's the start of the app, or holds a new location value.
- `start_date`: This holds the lower bound for the longest currently observed time delta between events for that value of `loc`.

- `end_date`: This holds the upper bound for the longest currently observed time elapsed between events for the value of `current_loc`.
- `start_time_obj`: This holds the most recently iterated `datetime` object, or `None` if it's the start of the app, or holds a new location value.
- `end_time_obj`: This holds the current `event_date` `datetime` object, or `None` if it's the start of the app, or holds a new location value.
- `current_diff`: This holds the time delta for the current longest observed time elapsed between events for the `current_loc`, or the lowest possible time delta if it's the start of the app, or a new location value.

The `for` loop reads rows over `stdin` that have already been sorted by the combination of `loc` and `event_date`. We parse each row into variables representing the column values by first stripping any additional newlines and splitting the line on tab characters.

The first conditional is skipped as `current_loc` is equal to `START OF APP`. We have only begun processing the first row across all locations on that reducer, and have nothing to output yet. Should we have a value for `loc` that is different from the value of `current_loc`, and we are not at the start of the application, then that is a signal that we are done processing the rows for `current_loc`, and can safely output the longest time delta for events in that location. Should `end_date` still be set to an empty string, then that indicates we only saw one event for that location. In this scenario, we do not output anything for that location. Finally, we reset the six placeholder variables previously explained, so that we may accurately process the records for the next location.

Following the conditional, we immediately set the value of `current_loc` that we are processing equal to `loc`, to avoid unnecessary entry of the mentioned conditional on the next iteration when we have not yet transitioned locations. We set `end_time_obj` to the value of `event_date` for the current row. If `start_time_obj` is set to `None`, then that means we are on the first row for that location and cannot yet do a time delta comparison. Whether or not `start_time_obj` is set to `None`, at the end of the loop we set `prev_date` equal to `event_date` and `start_time_obj` equal to `end_time_obj` of the current iteration. By doing so, on the next iteration, `start_time_obj` will hold the `event_date` of the previous record, while `end_time_obj` will hold the `event_date` of the current record.

When `start_time_obj` is no longer set to `None` after the first iteration for a given location, we can begin doing `diff` comparisons on these two `datetime` objects. Subtracting `start_time_obj` from `end_time_obj` yields a time delta object, which if larger than the `current_diff` value, gets set as the value for `current_diff`. In doing so, we capture the longest elapsed time period for that location between events. We also set the values of `start_date` and `end_date` for easy output later, once we are done processing this location. As mentioned earlier, whether or not we reset `current_diff`, we then change `prev_date` to point to `event_date` and `start_time_obj` equal to the current `end_time_obj`.

The next time the loop encounters the condition where `loc` is not equal to `current_loc`, we output the currently held longest time difference between events, before we move onto the next event. Each print to `stdout` writes a row into the receiving Hive table that holds each location held by `current_loc`, the `lower_bound event_date` string held by `start_date`, the upper bound `event_date` string held by `end_date`, and the total number of days elapsed between those two dates held by `current_diff.days`.

There's more...

Here are a few additional notes on some of the operations touched upon in this recipe:.

SORT BY versus DISTRIBUTE BY versus CLUSTER BY versus ORDER BY

These four operator variants always cause confusion to Hive beginners. Here's a quick comparison so you'll know which one is appropriate for your use case:

- `DISTRIBUTE BY`: Rows with matching column values will partition to the same reducer. When used alone, it does not guarantee sorted input to the reducer.
- `SORT BY`: This dictates which columns to sort by when ordering reducer input records.
- `CLUSTER BY`: This is a shorthand operator to perform both `SORT BY` and `DISTRIBUTE BY` operations on a group of columns.
- `ORDER BY`: This is similar to the traditional SQL operator. Sorted order is maintained across all of the output from every reducer. Use this with caution as it can force all of the output records to a single reducer to perform the sorting. Usage with `LIMIT` is strongly recommended.

MAP and REDUCE keywords are shorthand for SELECT TRANSFORM

The Hive keywords `MAP` and `REDUCE` are shorthand notations for `SELECT TRANSFORM`, and do not force the query execution to jump around stages. You can use any one of the three and achieve the same functional results. They are simply for query readability purposes.

See also

- The *Using Hive and Python to clean and transform geographical event data* in recipe *Chapter 3*, *Extracting and Transforming Data*

Calculating the cosine similarity of artists in the Audioscrobbler dataset using Pig

Cosine similarity is used to measure the similarity of two vectors. In this recipe, it will be used to find the similarity of artists based on the number of times Audioscrobbler users have added each user to their playlist. The idea is to show how often users play both artist 1 and artist 2.

Getting ready

Download the `Audioscrobbler` dataset from `http://www.packtpub.com/support`.

How to do it...

Perform the following steps to calculate cosine similarity using Pig:

1. Copy the `artist_data.txt` and `user_artist_data.txt` files into HDFS:

```
hadoop fs –put artist_data.txt user_artist_data.txt /data/
audioscrobbler/
```

2. Load the data into Pig:

```
plays = load '/data/audioscrobbler/user_artist_data.txt'
        using PigStorage(' ') as (user_id:long, artist_id:long,
playcount:long);

artist = load '/data/audioscrobbler/artist_data.txt' as (artist_
id:long, artist_name:chararray);
```

3. Sample the `user_artist_data.txt` file:

```
plays = sample plays .01;
```

4. Normalize the play counts to `100`:

```
user_total_grp = group plays by user_id;

user_total = foreach user_total_grp generate group as user_id,
SUM(plays.playcount) as totalplays;

plays_user_total = join plays by user_id, user_total by user_id
using 'replicated';

norm_plays = foreach plays_user_total generate user_total::user_id
as user_id, artist_id, ((double)playcount/(double)totalplays) *
100.0 as norm_play_cnt;
```

5. Get artist pairs for each user:

```
norm_plays2 = foreach norm_plays generate *;

play_pairs = join norm_plays by user_id, norm_plays2 by user_id
using 'replicated';

play_pairs = filter play_pairs by norm_plays::plays::artist_id !=
norm_plays2::plays::artist_id;
```

6. Calculate cosine similarity:

```
cos_sim_step1 = foreach play_pairs generate ((double)norm_
plays::norm_play_cnt) * (double)norm_plays2::norm_play_cnt) as
dot_product_step1, ((double)norm_plays::norm_play_cnt *(double)
norm_plays::norm_play_cnt) as play1_sq;
((double)norm_plays2::norm_play_cnt *(double) norm_plays2::norm_
play_cnt) as play2_sq;

cos_sim_grp = group cos_sim_step1 by (norm_plays::plays::artist_
id, norm_plays2::plays::artist_id);

cos_sim_step2 = foreach cos_sim_grp generate flatten(group),
COUNT(cos_sim_step1.dot_prodct_step1) as cnt, SUM(cos_sim_step1.
dot_product_step1) as dot_product, SUM(cos_sim_step1.norm_
plays::norm_play_cnt) as tot_play_sq, SUM(cos_sim_step1.norm_
plays2::norm_play_cnt) as tot_play_sq2;

cos_sim = foreach cos_sim_step2 generate group::norm_
plays::plays::artist_id as artist_id1, group::norm_plays2::plays_
artist_id as artist_id2, dot_product / (tot_play_sq1 * tot_play_
sq2) as cosine_similarity;
```

7. Get the artist's name:

```
art1 = join cos_sim by artist_id1, artist by artist_id using
'replicated';
art2 = join art1 by artist_id2, artist by artist_id using
'replicated';
art3 = foreach art2 generate artist_id1, art1::artist::artist_name
as artist_name1, artist_id2, artist::artist_name as artist_name2,
cosin_similarity;
```

8. To output the top 25 records:

```
top = order art3 by cosine_similarity DESC;
top_25 = limit top 25;
dump top25;
```

The output would be:

```
(1000157,AC/DC,3418,Hole,0.9115799166673817)
(829,Nas,1002216,The Darkness,0.9110152004952198)
(1022845,Jessica Simpson,1002325,Mandy Moore,0.9097097460071537)
(53,Wu-Tang Clan,78,Sublime,0.9096468367168238)
(1001180,Godsmack,1234871,Devildriver,0.9093019011575069)
(1001594,Adema,1007903,Maroon 5,0.909297052154195)
(689,Bette Midler,1003904,Better Than Ezra,0.9089467492461345)
(949,Ben Folds Five,2745,Ladytron,0.908736095810886)
(1000388,Ben Folds,930,Eminem,0.9085664586931873)
(1013654,Who Da Funk,5672,Nancy Sinatra,0.9084521262343653)
(1005386,Stabbing Westward,30,Jane's Addiction,0.9075360259222892)
(1252,Travis,1275996,R.E.M.,0.9071980963712077)
(100,Phoenix,1278,Ryan Adams,0.9071754511713067)
(2247,Four Tet,1009898,A Silver Mt. Zion,0.9069623744896833)
(1037970,Kanye West,1000991,Alison Krauss,0.9058717234023009)
(352,Beck,5672,Nancy Sinatra,0.9056851798338253)
(831,Nine Inch Nails,1251,Morcheeba,0.9051453756031981)
(1007004,Journey,1005479,Mr. Mister,0.9041311825160151)
(1002470,Elton John,1000416,Ramones,0.9040551837635081)
(1200,Faith No More,1007903,Maroon 5,0.9038274644717641)
(1002850,Glassjaw,1016435,Senses Fail,0.9034604126636377)
(1004294,Thursday,2439,HiM,0.902728300518356)
(1003259,ABBA,1057704,Readymade,0.9026955950032872)
(1001590,Hybrid,791,Beenie Man,0.9020872203833108)
(1501,Wolfgang Amadeus Mozart,4569,Simon &
Garfunkel,0.9018860912385024)
```

How it works...

The `load` statements tell Pig about the format and datatypes of the data being loaded. Pig loads data lazily. This means that the `load` statements at the beginning of this script will not do any work until another statement is entered that asks for output.

The `user_artist_data.txt` file is sampled so that a replicated join can be used when it is joined with itself. This significantly reduces the processing time at the cost of accuracy. The sample value of `.01` is used, meaning that roughly one in hundred rows of data will be loaded.

A user selecting to play an artist is treated as a vote for that artist. The play counts are normalized to `100`. This ensures that each user is given the same number of votes.

A self join of the `user_artist_data.txt` file by `user_id` will generate all pairs of artists that users have added to their playlist. The filter removes duplicates caused by the self join.

The next few statements calculate the cosine similarity. For each pair of artists that users have added to their playlist, multiply the number of plays for artist 1 by the number of plays for artist 2. Then output the number of plays for artist 1 and the number of plays for artist 2. Group the previous result by each pair of artists. Sum the multiplication of the number of plays for artist 1 by the number of plays by artist 2 for each user generated previously as the dot product. Sum the number of plays for artist 1 by all users. Sum the number of plays for artist 2 by all users. The cosine similarly is the dot product over the total plays for artist 1 multiplied by the total plays for artist two. The idea is to show how often users play both artist 1 and artist 2.

Trim Outliers from the Audioscrobbler dataset using Pig and datafu

Datafu is a Pig UDF library open sourced by the SNA team at LinkedIn. It contains many useful functions. This recipe will use **play counts** from the `Audioscrobbler` dataset and the **Quantile** UDF from datafu to identify and remove outliers.

Getting ready

- Download Version 0.0.4 of datafu from `https://github.com/linkedin/datafu/downloads`.
- Uncompress and untar the files. Add the `datafu-0.0.4/dist/ datafu-0.0.4.jar` file to a location accessible by Pig.
- Download the `Audioscrobbler` dataset from `http://www.packtpub.com/support`.

How to do it...

1. Register the datafu JAR file and construct the `Quantile` UDF:

```
register /path/to/datafu-0.0.4.jar;
define Quantile datafu.pig.stats.Quantile('.90');
```

2. Load the `user_artist_data.txt` file:

```
plays = load '/data/audioscrobbler.txt'using PigStorage(' ') as
(user_id:long, artist_id:long, playcount:long);
```

3. Group all of the data:

```
plays_grp = group plays ALL;
```

4. Generate the ninetieth percentile value to be used as the outlier's max:

```
out_max = foreach plays_grp{
         ord = order plays by playcount;
          generate Quantile(ord.playcount) as ninetieth ;
          }
```

5. Trim outliers to the ninetieth percentile value:

```
trim_outliers = foreach plays generate user_id, artist_id,
(playcount>out_max.ninetieth ? out_max.ninetieth : playcount);
```

6. Store the `user_artist_data.txt` file with outliers trimmed:

```
store trim_outliers into '/data/audioscrobble/outliers_trimmed.
bcp';
```

How it works...

This recipe takes advantage of the datafu library open sourced by LinkedIn. Once a JAR file is registered, all of its UDFs are available to the Pig script. The `define` command calls the constructor of the `datafu.pig.stats.Quantile` UDF passing it a value of `.90`. The constructor of the `Quantile` UDF will then create an instance that will produce the ninetieth percentile of the input vector it is passed. The `define` also aliases `Quantile` as shorthand for referencing this UDF.

The user artist data is loaded into a relation named `plays`. This data is then grouped by `ALL`. The `ALL` group is a special kind of group that creates a single bag containing all of the input.

The `Quantile` UDF requires that the data it has passed be sorted first. The data is sorted by play count, and the sorted play count's vector is passed to the `Quantile` UDF. The sorted play count simplifies the job of the `Quantile` UDF. It now picks the value at the ninetieth percentile position and returns it.

This value is then compared against each of the play counts in the user artist file. If the play count is greater, it is trimmed down to the value returned by the `Quantile` UDF, otherwise the value remains as it is.

The updated user artist file with outliers trimmed is then stored back in HDFS to be used for further processing.

There's more...

The datafu library also includes a `StreamingQuantile` UDF. This UDF is similar to the `Quantile` UDF except that it does not require the data to be sorted before it is used. This will greatly increase the performance of this operation. However, it does come at a cost. The `StreamingQuantile` UDF only provides an estimation of the values.

```
define Quantile datafu.pig.stats.StreamingQuantile('.90');
```

Where to buy this book

You can buy Hadoop Real-World Solutions Cookbook from the Packt Publishing website: `http://www.packtpub.com/hadoop-real-world-solutions-cookbook/book`.

Free shipping to the US, UK, Europe and selected Asian countries. For more information, please read our shipping policy.

Alternatively, you can buy the book from Amazon, BN.com, Computer Manuals and most internet book retailers.

www.PacktPub.com