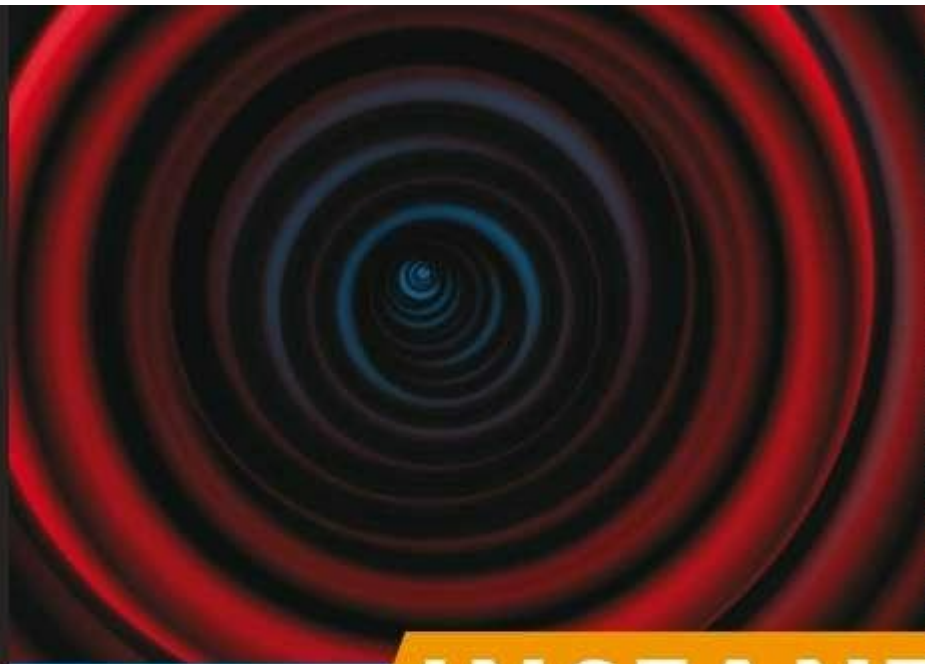




INSTANT

Short | Fast | Focused

Apache Hive Essentials How-to

Leverage your knowledge of SQL to easily write distributed data processing applications on Hadoop using Apache Hive

Darren Lee

[PACKT]
PUBLISHING

Table of Contents

[Instant Apache Hive Essentials How-to](#)

[Credits](#)

[About the Author](#)

[About the Reviewer](#)

[www.PacktPub.com](#)

[Support files, eBooks, discount offers and more](#)

[Why Subscribe?](#)

[Free Access for Packt account holders](#)

[Preface](#)

[What this book covers](#)

[What you need for this book](#)

[Who this book is for](#)

[Conventions](#)

[Reader feedback](#)

[Customer support](#)

[Downloading the example code](#)

[Errata](#)

[Piracy](#)

[Questions](#)

[1. Instant Apache Hive Essentials How-to](#)

[Tables and queries \(Simple\)](#)

[How to do it...](#)

[How it works...](#)

[There's more...](#)

[Understanding complex data types \(Simple\)](#)

[How to do it...](#)

[How it works...](#)

[There's more...](#)

[Using Hive non-interactively \(Simple\)](#)

[Getting ready](#)

[How to do it...](#)

[How it works...](#)

[Join optimizations \(Medium\)](#)

[How to do it...](#)

[How it works...](#)

[Map join](#)

[Bucketed tables](#)

[There's more...](#)

[Multiple joins](#)

[Skew joins](#)

[Multiple selects](#)

[List bucketing and skew joins](#)

[Setting the file format \(Simple\)](#)

[How to do it...](#)

- [How it works...](#)
- [Writing a custom SerDe \(Intermediate\)](#)
 - [How to do it...](#)
 - [How it works...](#)
 - [Object inspectors](#)
 - [Initialization](#)
 - [Serialization](#)
 - [Deserialization](#)
- [Using static partitions \(Intermediate\)](#)
 - [Getting ready](#)
 - [How to do it...](#)
 - [How it works...](#)
 - [There's more...](#)
 - [Loading data into partitioned internal tables](#)
 - [Writing data into specific partitions from queries](#)
- [Using dynamic partitions \(Intermediate\)](#)
 - [How to do it...](#)
 - [How it works...](#)
- [Using functions \(Simple\)](#)
 - [How to do it...](#)
 - [How it works...](#)
- [Adding custom logic with streaming \(Intermediate\)](#)
 - [Getting ready](#)
 - [How to do it...](#)
 - [How it works...](#)
- [Simple user-defined functions \(Intermediate\)](#)
 - [Getting ready](#)
 - [How to do it...](#)
 - [How it works...](#)
- [Advanced user-defined functions \(Advanced\)](#)
 - [How to do it...](#)
 - [How it works...](#)
 - [Initialization](#)
 - [Evaluation](#)
 - [The display string](#)
- [User-defined table-generating functions \(Advanced\)](#)
 - [How to do it...](#)
 - [How it works...](#)
 - [Initialization](#)
 - [Processing inputs](#)
 - [Final output](#)
- [User-defined aggregation functions \(Advanced\)](#)
 - [Getting ready](#)
 - [How to do it...](#)
 - [How it works...](#)
 - [The resolver](#)
 - [The modes of evaluation](#)
 - [Initializing the evaluator](#)

[Aggregation buffers](#)

[The logic of aggregation](#)

[There's more...](#)

Instant Apache Hive Essentials How-to

Instant Apache Hive Essentials

How-to

Copyright © 2013 Packt Publishing

All rights reserved. No part of this book may be reproduced, stored in a retrieval system, or transmitted in any form or by any means, without the prior written permission of the publisher, except in the case of brief quotations embedded in critical articles or reviews.

Every effort has been made in the preparation of this book to ensure the accuracy of the information presented. However, the information contained in this book is sold without warranty, either express or implied. Neither the author, nor Packt Publishing, and its dealers and distributors will be held liable for any damages caused or alleged to be caused directly or indirectly by this book.

Packt Publishing has endeavored to provide trademark information about all of the companies and products mentioned in this book by the appropriate use of capitals. However, Packt Publishing cannot guarantee the accuracy of this information.

First published: May 2013

Production Reference: 1170513

Published by Packt Publishing Ltd.

Livery Place

35 Livery Street

Birmingham B3 2PB, UK.

ISBN 978-1-78216-947-5

www.packtpub.com

Credits

Author

Darren Lee

Reviewer

Tanuj Khare

Acquisition Editor

Kartikey Pandey

Commissioning Editors

Sruthi Kutty

Meeta Rajani

Technical Editor

Sharvari Baet

Project Coordinator

Michelle Quadros

Proofreader

Clyde Jenkins

Production Coordinators

Melwyn D'sa

Nilesh R. Mohite

Cover Work

Nilesh R. Mohite

About the Author

Darren Lee studied mathematics at Stanford and New York University. He is currently the lead engineer on the data team at Bizo in San Francisco, CA. He has been using Hive since 2009 and occasionally blogs about it at <http://dev.bizo.com>.

I would like to thank Leah and Henry for their patience as I wrote this book and the Bizo engineering team for teaching me new things every day.

About the Reviewer

Tanuj Khare has worked as a professional in IT for more than eight years. He is involved in Process improvements using ITIL framework and techniques (LEAN and Six sigma). He is MCSA and ITIL v3/SO certified and has expertise in handling critical production server issues and has a track record of dealing with many complex problems. His quick resolution to issues faced in a production environment helped his team and clients a lot. Tanuj has SME (subject matter expertise) in Tomcat, WebLogic, and JBOSS Server Administration. His experience includes working with large enterprise web hosting environments for J2EE container and small teams, and his quick turnaround time resulted in on-time delivery. Apart from this, he is instrumental in setting-up and implementing one of the key business transitions to the organization. He has great skills on customer relationship management and is highly focused on "win-win" results for clients and mentoring a young dynamic team.

He has also worked on *Apache Tomcat 7 essentials*, Packt Publishing. He can be reached on <tanujkhare@gmail.com>.

I would like to thank my brother Tarun Khare and great people at Packt for letting me review this book. Special thanks to Mary Nader, Sruthi Kutty, and Michelle Quadros for having introduced me to the Packt family.

www.PacktPub.com

Support files, eBooks, discount offers and more

You might want to visit www.PacktPub.com for support files and downloads related to your book.

Did you know that Packt offers eBook versions of every book published, with PDF and ePub files available? You can upgrade to the eBook version at www.PacktPub.com and as a print book customer, you are entitled to a discount on the eBook copy. Get in touch with us at [<service@packtpub.com>](mailto:service@packtpub.com) for more details.

At www.PacktPub.com, you can also read a collection of free technical articles, sign up for a range of free newsletters and receive exclusive discounts and offers on Packt books and eBooks.



<http://PacktLib.PacktPub.com>

Do you need instant solutions to your IT questions? PacktLib is Packt's online digital book library. Here, you can access, read and search across Packt's entire library of books.

Why Subscribe?

- Fully searchable across every book published by Packt
- Copy and paste, print and bookmark content
- On demand and accessible via web browser

Free Access for Packt account holders

If you have an account with Packt at www.PacktPub.com, you can use this to access PacktLib today and view nine entirely free books. Simply use your login credentials for immediate access.

Preface

Hadoop provides a robust framework for building distributed applications, but working directly with Hadoop requires writing a lot of code. Adding structure to data and using a higher-level language such as SQL makes working with Hadoop both easier and faster. *Instant Apache Hive Essentials How-to* contains a series of practical tutorials that introduce the power and flexibility of Hive. Starting with your first query, we provide step-by-step instructions and behind-the-scenes explanations for how to write Map/Reduce jobs effectively with SQL.

What this book covers

Tables and queries introduces the Hive console by loading tab-delimited Olympic data into Hive and use a single SQL query to find which cities have most frequently bid on hosting the Olympics.

Understanding complex data types introduces one distinguishing feature of Hive that is its support for columns containing arrays, maps, structs, and unions. In this recipe, we load a file containing complex data types to find the athletes who have won most medals.

Using Hive non-interactively will demonstrate how to run the top athletes query non-interactively. The Hive console is great for exploring data interactively; however, we often need to run a Hive job automatically as part of a periodic job or other workflow.

JOIN optimizations explains how to use map join and bucketed table optimizations to combine the table of medals won by athletes with the table of athletes' country affiliations. Often, the most expensive part of a query is a JOIN. Hive supports several different optimizations that will speed up certain types of JOINS.

Setting the file format explains that the file format of input files is often prescribed by another system. We show how to use regular expressions to parse records and how to write records in Hive's three natively supported file formats: text files, sequence files, and row-columnar files.

Writing a custom SerDe will explain how to add support to Hive for reading delimited key-value pairs as columns in a table. When regular expressions are insufficient or inefficient to read and write data, we can write custom serialization and deserialization code to support the file format of our choice.

Using static partitions explains how we incrementally load our athlete data into different static partitions of an external table. This mirrors the common

use case of periodically adding an additional partition of data to a master table. Partitioning tables allows Hive to consider only a subset of the data when queries are restricted to particular partitions.

Using dynamic partitions explains how we can take our non-partitioned athlete data and partition it by country using a Hive query. If our input data is not already partitioned, we can ask Hive to partition the data for us.

Using functions demonstrates how to list the available functions, retrieve their documentation, and invoke them in queries. Hive contains a large number of built-in functions that perform tasks as simple as computing the length of a string or as complex as tokenizing a block of text into sentences and words.

Adding custom logic with streaming demonstrates how Hive uses Hadoop streaming to support inserting arbitrary code into a SQL query. In particular, we construct Ruby maps and reduce scripts to compute the average length of the unique Olympic athlete names.

Simple user-defined functions explains how Hive also supports adding our own user-defined functions, which run inside of Hive, and can be discovered and documented like a built-in function. We add support for computing MD5 hashes of strings to Hive.

Advanced user-defined functions explains how to rewrite our MD5 hash function as a generic user-defined function, allowing it to hook into the query parser, and accepting an arbitrary number of string or binary column inputs.

User-defined table-generating functions explains how to create a function that outputs all bigrams present in athlete names. Standard user-defined functions map columns from a single row to a single column of output. A user-defined table generating function can output an arbitrary number of rows for each input.

User-defined aggregation functions explains how to construct a user-defined aggregation function that computes a linear regression from two columns of numeric data. Aggregation functions allow us to create new reducer logic that collapses many rows of input into a single output.

What you need for this book

To run the examples in this book, you will need to have both Hadoop and Hive installed. The scripts in this book were written and tested with Hive 0.10.0 running on top of Hadoop 1.1.1 in pseudo-distributed mode; however, the majority of the scripts will also run on previous versions of Hadoop and Hive.

The recipes in this book are based off of the Freebase Olympics TSV data dump available at <http://download.freebase.com/datadumps/2012-11-04/>. Download the `olympics.tar.bz2` file and extract it into the current directory.

The headers of each file should be removed before use. On most Unix systems, one quick way to do so is the following two commands:

```
$ mkdir data/  
$ for f in `ls *.tsv` ; do tail -n +2 $f >  
data/$f ; done
```

We'll assume that these data files are available in the `data` subdirectory of the current working directory.

For the most up-to-date instructions for installing Hadoop and Hive, please refer to the online documentation at <http://hadoop.apache.org/> and <http://hive.apache.org/>.

Who this book is for

This book is written for data analysts and developers who want to use their current knowledge of SQL to be more productive with Hadoop. We assume that you are already familiar with SQL and have a basic understanding of the parts of a map-reduce algorithm.

Conventions

In this book, you will find a number of styles of text that distinguish between different kinds of information. Here are some examples of these styles, and an explanation of their meaning.

Code words in text are shown as follows: " We'll assume that these data files are available in the `data` subdirectory of the current working directory."

A block of code is set as follows:

```
hive> create table if not exists city_bid(  
    > name string,  
    > id string,  
    > bidding_city string,  
    > olympic_games string)  
    > row format delimited fields terminated by  
    '\t' ;
```

When we wish to draw your attention to a particular part of a code block, the relevant lines or items are set in bold:

```
hive> create table if not exists city_bid(  
    > name string,  
    > id string,  
    > bidding_city string,  
    > olympic_games string)  
    > row format delimited fields terminated by  
    '\t' ;
```

Any command-line input or output is written as follows:

```
$ mkdir data/  
$ for f in `ls *.tsv` ; do tail -n +2 $f >  
data/$f ; done
```

New terms and **important words** are shown in bold. Words that you see on the screen, in menus or dialog boxes for example, appear in the text like this: "clicking the **Next** button moves you to the next screen".

Reader feedback

Feedback from our readers is always welcome. Let us know what you think about this book—what you liked or may have disliked. Reader feedback is important for us to develop titles that you really get the most out of.

To send us general feedback, simply send an e-mail to [<feedback@packtpub.com>](mailto:feedback@packtpub.com), and mention the book title via the subject of your message.

If there is a topic that you have expertise in and you are interested in either writing or contributing to a book, see our author guide on www.packtpub.com/authors.

Customer support

Now that you are the proud owner of a Packt book, we have a number of things to help you to get the most from your purchase.

Downloading the example code

You can download the example code files for all Packt books you have purchased from your account at <http://www.packtpub.com>. If you purchased this book elsewhere, you can visit <http://www.packtpub.com/support> and register to have the files e-mailed directly to you.

Errata

Although we have taken every care to ensure the accuracy of our content, mistakes do happen. If you find a mistake in one of our books—maybe a mistake in the text or the code—we would be grateful if you would report this to us. By doing so, you can save other readers from frustration and help us improve subsequent versions of this book. If you find any errata, please report them by visiting <http://www.packtpub.com/submit-errata>, selecting your book, clicking on the **errata submission form** link, and entering the details of your errata. Once your errata are verified, your submission will be accepted and the errata will be uploaded on our website, or added to any list of existing errata, under the Errata section of that title. Any existing errata can be viewed by selecting your title from <http://www.packtpub.com/support>.

Piracy

Piracy of copyright material on the Internet is an ongoing problem across all media. At Packt, we take the protection of our copyright and licenses very seriously. If you come across any illegal copies of our works, in any form, on the Internet, please provide us with the location address or website name immediately so that we can pursue a remedy.

Please contact us at [<copyright@packtpub.com>](mailto:copyright@packtpub.com) with a link to the suspected pirated material.

We appreciate your help in protecting our authors, and our ability to bring you valuable content.

Questions

You can contact us at [<questions@packtpub.com>](mailto:questions@packtpub.com) if you are having a problem with any aspect of the book, and we will do our best to address it.

Chapter 1. Instant Apache Hive Essentials How-to

Welcome to *Instant Apache Hive Essentials How-to*. This book provides a series of tutorials about using Hive to read data in various formats, efficiently querying this data, and extending Hive with any custom functions that you may need to insert using your own logic into the data pipeline.

Tables and queries (Simple)

This section demonstrates loading data into a Hive table, manipulating it with HQL (the Hive variant of SQL), and outputting the results to the console. We will use Hive to count the number of times each city has bid on hosting the Olympics.

How to do it...

1. Start the Hive console by entering `hive` at the command prompt. We will enter the statements in the following steps into the Hive console. (The "OK" responses, timing, and Map/Reduce progress information have been omitted from the output for brevity).
2. First, we create and select a new database for us in which to work. While using databases is not strictly necessary to use Hive, they are useful for separating tables into logical groups. For this book, we recommend creating and using a separate database for each chapter. If at any point you want to start a chapter over, the command `drop database ch1 cascade` will delete a database and all tables that it contains.

```
hive> create database if not exists ch1;
hive> use ch1;
```

3. Hive has many configuration options; these can be set in the Hive configuration files, passed via the command line, or set within a script. Normally, the Hive query processor sends Map Reduce jobs to the external Hadoop process. The `hive.exec.mode.local.auto` setting instructs Hive to instead run Hadoop in-process when the amount of data to be processed is small, saving the overhead of communicating with an external process. This setting only needs to be specified once in each Hive session.

```
hive> set hive.exec.mode.local.auto =
true ;
```

4. We can now create our input table of the cities that bid on hosting each Olympics. Hive supports both primitive and complex column types. Primitive types include four sizes of integers (tinyint, smallint, int, and bigint), boolean, float, double, string, binary, or timestamp. Complex column types include arrays, maps, structs, and unions. The actual data in Hive tables is stored in files in HDFS. We can specify the format of these underlying data files as part of the table definition.

```
hive> create table if not exists city_bid(  
  > name string,  
  > id string,  
  > bidding_city string,  
  > olympic_games string)  
  > row format delimited fields  
terminated by '\t' ;
```

5. Next, we instruct Hive to copy data from a local file into our table. If we have multiple files to load, we can omit the `overwrite` keyword and Hive will append each file to the table data. These files must be in the format specified when the table was created.

```
hive> load data local inpath  
'data/olympic_city_bid.tsv'  
  > overwrite into table city_bid ;
```

6. The next table will contain our output, the number of bids made by each city. We don't particularly care about the file format, so we allow Hive to just use the defaults.

```
hive> create table bids_per_city (  
  > bidding_city string,  
  > num_bidsint) ;
```

7. We are now ready to aggregate the number of bids for each city and write the results into our output table. Hive will translate this query into a Map/Reduce job.

```
hive> insert overwrite table bids_per_city  
  > select bidding_city, count(1) as  
num_bids  
  > from city_bid  
  > group by bidding_city  
  > order by num_bids desc, bidding_city  
asc ;
```

8. Not all queries are converted into Map/Reduce jobs. Simple queries of the form `select * from table` with an optional `limit` clause just read data from HDFS and print it directly to the console.

```
hive> select * from bids_per_city limit 5 ;  
Istanbul 3
```

```
Amsterdam 2  
Vancouver 2  
Athens 1  
Beijing 1
```

Tip

Downloading the example code

You can download the example code files for all Packt books you have purchased from your account at <http://www.packtpub.com>. If you purchased this book elsewhere, you can visit <http://www.packtpub.com/support> and register to have the files e-mailed directly to you.

We've attempted to make the text in this book faithful to actual input and output in the Hive console; however, we've removed debug and status information that can differ from one run to the next in the interests of brevity and clarity. The following screenshot shows how the last two queries looked for the author with all output included:

```

hive> insert overwrite table bids_per_city
  > select bidding_city, count(1) as num_bids
  > from city_bid
  > group by bidding_city
  > order by num_bids desc, bidding_city asc ;
Automatically selecting local only mode for query
Total MapReduce jobs = 2
Launching Job 1 out of 2
Number of reduce tasks not specified. Estimated from input data size: 1
In order to change the average load for a reducer (in bytes):
  set hive.exec.reducers.bytes.per.reducer=<number>
In order to limit the maximum number of reducers:
  set hive.exec.reducers.max=<number>
In order to set a constant number of reducers:
  set mapred.reduce.tasks=<number>
13/04/27 20:23:51 WARN conf.HiveConf: hive-site.xml not found on CLASSPATH
WARNING: org.apache.hadoop.metrics.jvm.EventCounter is deprecated. Please use org.apache.h
adoop.log.metrics.EventCounter in all the log4j.properties files.
Execution log at: /tmp/darren/darren_20130427202323_0fdd04f5-4d9a-4f59-9692-711b7a6ba3c7.1
og
2013-04-27 20:23:51.234 java[60841:1c03] Unable to load realm info from SCDynamicStore
Job running in-process (local Hadoop)
Hadoop job information for null: number of mappers: 0; number of reducers: 0
2013-04-27 20:23:53,511 null map = 100%,   reduce = 100%
Ended Job = job_local_0001
Execution completed successfully
Mapred Local Task Succeeded . Convert the Join into MapJoin
Launching Job 2 out of 2
Number of reduce tasks determined at compile time: 1
In order to change the average load for a reducer (in bytes):
  set hive.exec.reducers.bytes.per.reducer=<number>
In order to limit the maximum number of reducers:
  set hive.exec.reducers.max=<number>
In order to set a constant number of reducers:
  set mapred.reduce.tasks=<number>
13/04/27 20:23:55 WARN conf.HiveConf: hive-site.xml not found on CLASSPATH
WARNING: org.apache.hadoop.metrics.jvm.EventCounter is deprecated. Please use org.apache.h
adoop.log.metrics.EventCounter in all the log4j.properties files.
Execution log at: /tmp/darren/darren_20130427202323_0fdd04f5-4d9a-4f59-9692-711b7a6ba3c7.1
og
2013-04-27 20:23:55.619 java[60868:1c03] Unable to load realm info from SCDynamicStore
Job running in-process (local Hadoop)
Hadoop job information for null: number of mappers: 0; number of reducers: 0
2013-04-27 20:23:58,274 null map = 100%,   reduce = 100%
Ended Job = job_local_0001
Execution completed successfully
Mapred Local Task Succeeded . Convert the Join into MapJoin
Loading data to table ch1.bids_per_city
Deleted hdfs://localhost:9000/user/hive/warehouse/ch1.db/bids_per_city
Table ch1.bids_per_city stats: [num_partitions: 0, num_files: 1, num_rows: 32, total_size:
376, raw_data_size: 344]
OK
Time taken: 9.519 seconds
hive> select * from bids_per_city limit 5 ;
OK
Istanbul          3
Amsterdam         2
Vancouver         2
Athens            1
Beijing           1
Time taken: 0.076 seconds
hive> █

```

How it works...

Hadoop has two main components: **Hadoop Distributed File System (HDFS)** and **Hadoop Map/Reduce**—the data processing framework. Hive adds three pieces to this: a metadata store for schema information, a serialization/deserialization framework for reading and writing data, and a

query processor that translates SQL statements into instructions for these other four systems.

If we were writing a standard Hadoop job to compute the number of bids per city, we would need to load our source data into HDFS, run the MapReduce job, and then read the output data from HDFS. The statements in our Hive script do exactly this, with some additional manipulation of metadata to let us view our data as tables instead of files.

For example, a `create table` statement adds a new table definition to the Hive metastore. These table definitions include the column types, the location of the underlying data files, and the information necessary to serialize and deserialize rows from these files.

When we issue a query on this table, the Hive query processor creates one or more Map/Reduce jobs that read from the table's data files, deserialize the rows according to the format specified in the table definition, perform the logic of the query, and serialize the results into the output table according to its file format.

If we want to see how the query processor translates SQL into Map/Reduce jobs, we can prefix our queries with the `explain` keyword:

```
hive> explain select bidding_city, count(1)
> from city_bid
> group by bidding_city;
```

Instead of running the query, Hive will output the parsed abstract syntax tree and the query plan for us to inspect. For more details, such as the location of temporary files in HDFS, we can use the `explain extended` prefix. If we want to see which tables (and later, which partitions) are used by a query, we can use the `explain dependency` prefix.

There's more...

By default, tables are **internal**, meaning that both the metadata and the actual data files in HDFS are owned by Hive. In particular, if a `drop table` command is issued, both the metadata and the data files will be deleted.

Tables can also be defined as **external** tables. In this case only the metadata is owned by Hive. For external tables, dropping the table will not cause the data files to be deleted. This is useful if other processes read or write data into HDFS that you want to access from Hive. When creating external tables, we tell Hive where to find the pre-existing data using the `location` keyword.

For example, if we already had data files stored in HDFS in the directory `/olympics/city_bid/`, we can create an external table pointing to that location, as follows:

```
hive> create external table city_bid(  
    >   name string,  
    >   id string,  
    >   bidding_city string,  
    >   olympic_games string)  
    > row format delimited fields terminated by  
    '\t'  
    > location '/olympics/city_bid/';
```

Hive will assume that all of the files in `/olympics/city_bid/` will be in the correct format.

Understanding complex data types (Simple)

In addition to the primitive data types, Hive also supports the complex data types of arrays, maps, structs, and uniontypes. (The keyword **union** is reserved for concatenating tables in queries, so the column type of "union" is called **uniontype**.) These types allow denormalized schemas that are better aligned with the Map/Reduce query engine.

How to do it...

We will calculate the top modal winners using an athlete table with array types.

1. First, we create our table and load data into it. This table has four array columns:

```
hive> create table if not exists athlete(  
  >   name string,  
  >   id string,  
  >   demonstration_events_competed_in  
array<string>,  
  >   demonstration_medals_won  
array<string>,  
  >   country array<string>,  
  >   medals_won array<string>)  
  > row format delimited  
  >   fields terminated by '\t'  
  >   collection items terminated by  
' ,' ;  
  
hive> load data  
  > local inpath  
'data/olympic_athlete.tsv'  
  > overwrite into table athlete ;
```

2. The next query verifies that we have correctly loaded data into this table:

```
hive> select * from athlete limit 1 ;  
Chris Hoy /m/06p707 [] []  
["/m/065t7_m", "/m/065t5z0", "/m/0kf7w30"]  
["/m/04gb_3f", "/m/07jn_ls", "/m/0kjf4vr", "/m/0kgbwtc", "  
/m/04hb914", "/m/04h64ff"]
```

3. Using complex types enables us to easily find the athletes who have won the most Olympic medals. In this example, we use the size function to compute the number of medals won by each athlete:

```
hive> select name, size(medals_won) as
num_medals
> from athlete
> where size(medals_won) >= 10
> order by num_medals desc, name asc ;
Michael Phelps 22
Larissa Latynina 14
Jenny Thompson 12
Nikolai Andrianov 12
Matt Biondi 11
Ole Einar Bjørndalen 11
Ryan Lochte 11
Boris Shakhlin 10
Takashi Ono 10
```

How it works...

The `olympic_athlete.tsv` file is in standard tab-separated format; however, several of the fields contain multiple identifiers separated by commas. By specifying the type of these columns as `array<string>` and the collection delimiter as a comma, Hive can understand this data as arrays instead of simply strings.

With flat files in HDFS and Map/Reduce as its execution engine, Hive often works more efficiently on denormalized data. In this case, both the calculation of the number of medals and the filter on that value operate on a single row at a time and can be performed entirely within the mapper. The only data sent to the reducers are the names and medal counts of athletes with at least 10 medals, which is a small amount of data compared to the entire athlete table.

By contrast, if we were using a typical normalized schema with athlete, medal, and an athlete-medal relationship table, we would need to send the complete athlete and athlete-medal tables to at least one reducer step to join them (assuming that both tables are large). Moving, sorting, and reducing this data will be a much more expensive operation than processing the denormalized data in the mappers.

There's more...

When reading data files, Hive expects structs and uniontypes to be formatted as fixed length arrays. For structs, the items in the array correspond to the fields of the struct. For uniontypes, the first item in the array should be the index of the type, and the second item should be the value.

So, suppose we have the following table definition:

```
hive> create table struct_and_union(  
  >   s struct<a: string, b: string, c: int>,  
  >   u uniontype<string, int, double>  
  > row format delimited  
  >   fields terminated by '\t'  
  >   collection items terminated by ',' ;
```

Then, the input file could look something as follows:

```
ant,ball,1  0,hello  
car,deal,2  2,5.2  
egg,flip,3  1,4
```

The struct column always has two strings and an integer, and the union column always has the index of the type followed by the value.

Using Hive non-interactively (Simple)

So far, we have only used Hive through its interactive console. In this recipe, we will learn how Hive supports uses cases, such as periodic ETL jobs, by rerunning the top athletes query in batch mode from the command line. This support allows us to use external tools to start Hive jobs and capture their output.

Getting ready

Here are the initial steps to be followed:

1. Create the `init.hql` file in the current directory. This file sets up the database, specifies our Hive settings, and creates our input table.

```
create database if not exists ch3 ;
use ch3 ;
set hive.exec.mode.local.auto=true ;

create table if not exists athlete(
    name string,
    id string,
    demonstration_events_competed_in
array<string>,
    demonstration_medals_won array<string>,
    country array<string>,
    medals_won array<string>)
row format delimited
    fields terminated by '\t'
    collection items terminated by ',' ;

load data
local inpath 'data/olympic_athlete.tsv'
overwrite into table athlete ;
```

2. Create the `script.hql` in the current directory. This file creates our output table and executes a query.

```
create table if not exists top_athletes(
    name string,
    num_medals int) ;

insert overwrite table top_athletes
select name, size(medals_won) as num_medals
from athlete
where size(medals_won) >= ${threshold}
```

```
order by num_medals desc, name asc ;
```

How to do it...

Follow these steps to complete the task:

1. We start by running our initialization and query scripts from the command line:

```
$ hive -v --hivevar threshold=10 -i  
init.hql -f script.hql
```

2. We write the header of our `top_athletes` table to a file using the standard cut and paste Unix tools:

```
$ hive -S -e "use ch4; describe  
top_athletes;" | cut -f 1 | paste -s - >  
output.tsv
```

3. We write the data from the `top_athletes` table to our output file by executing explicit SQL statements:

```
$ hive -S -e "use ch4; select * from  
top_athletes ;" >> output.tsv
```

4. We can verify the contents of our file by simply using the `cat` command:

```
$ cat output.tsv  
name num_medals  
Michael Phelps 22  
Larissa Latynina 14  
Jenny Thompson 12  
Nikolai Andrianov 12  
Matt Biondi 11  
Ole Einar Bjørndalen 11  
Ryan Lochte 11  
Boris Shakhlin 10  
Takashi Ono 10
```

How it works...

From the command line, Hive supports three mutually exclusive modes. In addition to the interactive mode that we have used in previous sections, we can pass commands to Hive using two different flags.

Flag	Argument	Effect
<code>-f</code>	filename	Execute the contents of the file
<code>-e</code>	SQL statements	Execute the argument as input
(none)	(none)	Run Hive interactively

For all three of these cases, Hive allows using the `-i` filename parameter for passing an initialization script. This script will be executed first, then the session will continue with the file contents, explicit statements, or interactive session.

Hive supports using the `-i` flag multiple times, so we can make our initialization scripts even more reusable by splitting them according to their functionality. For example, we could have one initialization script with common settings used by all jobs running on a particular cluster and a second initialization script for all jobs that use particular table definitions.

In this example, we first use our two files to create the input and output tables and run a query. By using interactive scripts, we can separate the context of our query (the database, data loading, and Hive configuration) from the logic and output. Different processes can share the same context without needing to duplicate the contents of the initialization file across all of their scripts.

We also make our `script.sql` file reusable with different thresholds through variable substitution. Hive will automatically replace any occurrences of `${variable}` with the values passed to the hive command via the `--hivevar` parameter. The `-d` and `--define` parameters are synonyms for `--hivevar`, and all of these parameters may be specified multiple times if necessary.

The `-v` flag simply puts Hive into verbose mode, so each statement is echoed to the console as it is executed. Combining the verbose flag, variable substitution, and our scripts gives us the first shell command we executed:

```
$ hive-v --hivevar threshold=10 -iinit.hql -f
script.hql
```

We then execute two explicit commands, first to describe the columns of the `top_athletes` table, and then to output its contents. These are redirected to our output file on the local filesystem.

The `-s` flag puts Hive into silent mode, so only the output of our queries will be written to the files. This helps us capture only the table contents to our output file.

```
$ hive -S -e "use ch3; describe top_athletes;" |
cut -f 1 | paste -s - > output.tsv

$ hive -S -e "use ch3; select * from
top_athletes ;" >> output.tsv
```

In Hive Versions 0.10.0 and later, we can use the `-database ch3` flag instead of specifying `use ch3;` as part of the query. Alternatively, we could refer to the table by its full name `ch3.top_athletes`.

Join optimizations (Medium)

Hive supports inner, outer, cross, and left semi joins on equality conditions. Generally speaking, joins are implemented using a sort-merge algorithm, also known in Hive as the **common join**. Mappers iterate over each table and output the columns involved in the join condition as keys. Rows with the same values of the join condition are sent to a single reducer, which buffers these rows and performs a cross join.

In many cases, the join can be sped up considerably by changing the algorithm. Additionally, in cases where a key has more rows than the reducer can fit into memory, the base sort-merge algorithm can fail completely. This recipe will explain how to configure some of the join optimizations that Hive supports and how to structure joins in ways that are most likely to be optimized.

How to do it...

We will use two different optimization techniques to join the athlete and athlete affiliation tables to generate a list of athletes with their medal count and country affiliation.

1. First, we specify the maximum percentage of mapper memory that Hive should allow a "map join" operation to use:

```
hive> set
hive.mapjoin.localtask.max.memory.usage =
0.99 ;
```

2. This next setting specifies that Hive should use a target table's bucketing specification when determining how to configure the number of mappers and reducers using in a query:

```
hive> set hive.enforce.bucketing = true ;
```

3. Next, we create the athlete table and load our source data into it as follows:

```
hive> create table if not exists athlete(
>   name string,
>   id string,
>   demonstration_events_competed_in
array<string>,
>   demonstration_medals_won
array<string>,
>   country array<string>,
>   medals_won array<string>)
> clustered by (name) into 2 buckets
> row format delimited
> fields terminated by '\t'
```

```

        > collection items terminated by
        ',' ;

hive> load data
    > local inpath
    'data/olympic_athlete.tsv'
    > overwrite into table athlete ;

```

4. When loading data into a table, Hive simply copies the local file into HDFS. The easiest way to ensure that data is correctly loaded into a bucketed table is to instruct Hive to copy our original table into a bucketed table:

```

hive> create table if not exists
athlete_bucketed(
    > name string,
    > id string,
    > demonstration_events_competed_in
array<string>,
    > demonstration_medals_won
array<string>,
    > country array<string>,
    > medals_won array<string>)
    > clustered by (name) into 2 buckets ;

hive> insert overwrite table
athlete_bucketed
    > select * from athlete ;

```

5. Similarly, we load the original athlete affiliation data and copy it into another bucketed table:

```

hive> create table if not exists
athlete_affiliation(
    > name string,
    > id string,
    > athlete string,
    > country string,
    > olympics string,
    > sport string)
    > row format delimited
    > fields terminated by '\t'
    > collection items terminated by
    ',' ;

hive> load data
    > local inpath
    'data/olympic_athlete_affiliation.tsv'
    > overwrite into table
athlete_affiliation ;

hive> create table if not exists
athlete_affiliation_bucketed(
    > name string,
    > id string,

```

```

> athlete string,
> country string,
> olympics string,
> sport string)
> clustered by (athlete) into 4
buckets ;

hive> insert overwrite table
athlete_affiliation_bucketed
> select * from athlete_affiliation ;

```

6. Because both the athlete and athlete_affiliation tables are bucketed, we can now perform a bucketed join by setting the following three properties.

```

hive> set hive.optimize.bucketmapjoin =
true ;

hive> set
hive.optimize.bucketmapjoin.sortedmerge =
true ;

hive> set hive.input.format =
>
org.apache.hadoop.hive.ql.io.BucketizedHiveInputFormat;

hive> select
> l.name,
> size(l.medals_won),
> r.country,
> collect_set(r.olympics)
> from athlete_bucketed l
> join athlete_affiliation_bucketed
r
> on l.name = r.athlete
> group by l.name, size(l.medals_won),
r.country
> order by name asc
> limit 5 ;
..... Daumain 1 France ["1900 Summer
Olympics"]
A Lamusi 0 China ["2012 Summer
Olympics"]
A.G. Kruger 0 United States of America
["2012 Summer Olympics"]
Aarik Wilson 0 United States of America
["2008 Summer Olympics"]
Aaron Brown 0 Canada ["2012 Summer
Olympics"]

```

How it works...

We will discuss each of the optimizations in this section separately.

Map join

By default, Hive will try to determine whether the common join can be converted to a **map join**. Map joins can be used in cases where all but one table is large. The small tables are hashed by the join key and placed into the distributed cache. The mappers then compute a hash join of the small tables against the large table, and there is no reduce stage. Basically, Hive is performing an in-memory sort-merge of the small tables against partitions of the large table.

In order to determine if a map join is possible, Hive needs to know which tables are large and small. By convention, the largest table should generally appear last in a join. (The exception is for right outer joins, where the large table should be on the left.) Hive will start a local task to create a hash table from the remaining tables, place this table into the distributed cache, and run the remaining join against the large table as a distributed Map/Reduce job.

If the hash table in the local task exceeds a certain percentage of the maximum mapper memory, Hive will abort the map join and run a common join instead. For this query, we can allocate almost all of a mapper's memory to the hash table.

Bucketed tables

The next optimization comes from bucketing our tables. Normally, a large table consists of possibly hundreds, if not thousands or tens of thousands, of files. The distribution of rows across these files is not specified. In a bucketed table, the table metadata specifies that the rows have been clustered into some number of buckets based on the values of one or more columns. The values of these columns are hashed, and the hash value determines which file contains each row.

When the tables in a join are each bucketed on the join key into a compatible numbers of buckets (meaning one is a multiple of the other), Hive can join each bucket from the first table to the corresponding buckets of the second and ignore bucket combinations that will have no matching join keys.

In this section, our `athlete_bucketed` table contains two buckets and our `athlete_affiliation_bucketed` table contains four buckets. When performing the join, Hive will compare each of the `athlete_bucketed` buckets to only two of the `athlete_affiliation_bucketed` buckets.

Therefore, the bucketed join will only process half as much data as a standard join.

As our test data is small, this particular query is unlikely to execute any faster with bucketed tables. Since we now have multiple files per table, Hive will probably execute this query as remote Map/Reduce jobs instead of local ones, which will overwhelm any speed advantages from join optimization. However, with much larger datasets, the speed increase can be considerable. We would use many more buckets, and the efficiency gains from using a bucketed join would be correspondingly larger.

There's more...

Join optimization is a very active area of development in Hive, especially in the case of skewed data. Here are a few more methods that we can use to try and improve query speed.

Multiple joins

Hive supports placing multiple joins in a single query. If there is one column that appears in all of the join conditions, Hive can execute all of the joins in a single Map/Reduce job. Otherwise, Hive will run a separate Map/Reduce job per join.

Joins are processed left to right, so assuming that there are no right outer joins, the largest table should be placed last.

Skew joins

One common difficulty with joins occurs when the data is skewed by key. If a few keys are much more frequent than the rest, the entire Map/Reduce job will often be waiting for a few reducers to finally complete.

Hive includes some support for automatically optimizing skew joins. In each reducer, if the number of rows with a particular key exceeds some threshold, the rows will not be joined and will instead be output directly to HDFS. Additional Map/Reduce jobs will then be run on each high-frequency key. Often, the number of rows corresponding to this key in the other tables will be small, so these secondary jobs can be run as fast map joins instead of common joins.

We can enable this optimization by setting two properties, where `SKEW_THRESHOLD` is the number of rows (defaulting to 100,000) that may correspond to a key before the rows are diverted to a secondary job.

```
hive> set hive.optimize.skewjoin = true ;
```

```
hive> set hive.skewjoin.key = SKEW_THRESHOLD ;
```

The main difficulty with this technique is that the reducers still need to process every row for the frequent keys, even if they are not actually performing a join.

Multiple selects

If we want to explicitly separate skewed data from non-skewed data, we can use multiple select statements in a query to write to multiple tables in a single query.

```
hive> from athlete_affiliation
> insert overwrite table us_athletes
> select name, olympics
> where country = 'United States of America'
> insert overwrite table ca_athletes
> select name, olympics
> where country = 'Canada'
> insert overwrite table other_athletes
> select name, olympics
> where country <> 'United States of America'
> and country <> 'Canada' ;
```

This pattern avoids scanning the source table more than once, and is useful whenever we want to read from one table and write to many others.

List bucketing and skew joins

As an alternative to the hash bucketing described in this section, recent versions of Hive have been incrementally including support for **list bucketing**. With hash bucketing, the location of a particular row is determined entirely by the value of the hashed columns. List bucketing addresses this problem by storing an explicit map of keys to buckets in the table metadata. In particular, Hive can now distribute keys across the buckets according to their frequency, so the buckets become much closer in size.

As list bucketing necessarily needs to know about the relative frequency of keys, this information can also be used during skew join optimization. Because, which keys are frequent was computed when the data was loaded into the table, rows for frequent keys can be sent directly to secondary Map/Reduce jobs without first being processed by the reducers of the common join task.

Setting the file format (Simple)

When the Hive query processor transforms a SQL query into a Hadoop job, it needs to instruct Hadoop how to read the input data, how to transform it, and how to write the output data. So far, we have focused on how the data is transformed. In this recipe, we will explore how Hive handles input and output.

How to do it...

The following script uses the regular expression row format to read the city bid data. It then copies this data into two other tables, one backed by sequence files and the other backed by row columnar files. This script demonstrates how we can use Hive's serialization and deserialization framework to convert files from one format to another.

1. We start by creating the `city_bid` table. Instead of using the delimited row format, we specify a custom `serde` to handle serialization and deserialization of the table data. This particular `serde` uses a regular expression to define the row format.

```
hive> create table if not exists city_bid(
>   name string,
>   id string,
>   bidding_city string,
>   olympic_games string)
> row format serde
>
'org.apache.hadoop.hive.serde2.RegexSerDe'
>   with serdeproperties(
>     'input.regex' =
>
'([^t]*)\t([^t]*)\t([^t]*)\t([^t]*)')
> stored as textfile ;

hive> load data
>   local inpath
>   'data/olympic_city_bid.tsv'
>   overwrite into table city_bid ;
```

2. Next, we define two output tables. The first stores data as a Hadoop sequence file. The second stores data as a row columnar file.

```
hive> create table if not exists
city_bid_seqfile(
>   name string,
>   id string,
>   bidding_city string,
>   olympic_games string)
```

```

> row format delimited
>   fields terminated by '\t'
>   collection items terminated by
'\002'
>   map keys terminated by '\003'
> stored as sequencefile ;

hive> create table if not exists
city_bid_rcfile(
>   name string,
>   id string,
>   bidding_city string,
>   olympic_games string)
> stored as rcfile ;

```

3. Finally, we can copy the data from our input table to our two output tables using a multiple insert statement. We can also verify that the data in the two output tables was created correctly.

```

hive> from city_bid
> insert overwrite table
city_bid_seqfile select *
> insert overwrite table
city_bid_rcfile select * ;

hive> select * from city_bid_seqfile limit
2 ;
/m/08lq3hx  Buenos Aires  2004 Summer
Olympics
/m/08lq3hp  Cape Town    2004 Summer
Olympics

hive> select * from city_bid_rcfile limit
2 ;
/m/08lq3hx  Buenos Aires  2004 Summer
Olympics
/m/08lq3hp  Cape Town    2004 Summer
Olympics

```

How it works...

To instruct Hadoop how to read or write data from a table, Hive needs two pieces of information. First, it needs to know how records are stored in the file. Second, it needs to know how to transform a record into the columns of the table. This information is specified when a table is created and it becomes part of the table's metadata.

We have been consistently defining our input tables as row format delimited text files, indicating that each row should be split by the field delimiter, then successively split according to collection and map key delimiters as necessary. By default, tables are row format delimited with the binary

delimiters `\001` (also known as `"^A"` or `Ctrl + A`) for fields, `'\002'` for collection items, and `\003` for key-value pairs in maps.

In this script, we specify a different row format with a **SerDe** (serialization and deserialization) clause in our create table statement. This particular SerDe uses a regular expression to parse each row into columns.

Hive currently has native support for three file formats. Text files and sequence files each contain one record per line, while RC files restructure data to store records in a columnar format.

Text files, the default, are the most basic and are useful for reading or writing data that will be touched by other systems.

The sequence file format is a standard Hadoop file format that stores data as key-value pairs. The main advantage of sequence files is that they can be easily partitioned into blocks for different mappers to read.

Finally, RC files first split the records into blocks, then each block is stored in columnar format. Because each row is completely contained in a single block, multiple mappers can process each file. Within each block, the columnar format can speed queries that do not need to read the entire row.

To specify a different file format, we use the `a stored as` phrase in the table definition.

Writing a custom SerDe (Intermediate)

In addition to built-in support for several file formats, Hive allows users to write their own custom serialization and deserialization code.

While exploring complex types, we briefly discussed how using maps over a key-value file format allows for a very flexible schema. One downside of that method is that the contents of each row are hidden as the keys of the map. In this recipe, we will write the serialization and deserialization code necessary to view these maps as normal columns of a table. This SerDe will maintain the flexibility of having a schema-less file format with the readability of a columnar table.

How to do it...

We first need to create an implementation of the `SerDe` class for our new file format.

1. We start by implementing the `SerDe` interface and setting up the internal state variables needed by other methods.

```
package com.leeriggins.hive;

// imports omitted...

public class ColumnarMapSerDe implements
    SerDe {

    private List<String> columnNames;
    private ObjectInspector objectInspector;
    private Map<String, String> rowMap;
    private List<String> rowFields;
    private long deserializedByteCount;
    private SerDeStats stats;
```

2. The `initialize` method is called when a table is created. It is responsible for verifying that the table definition is compatible with the underlying serialization and deserialization mechanism. In this case, we only support strings.

```
@Override
public void initialize(
    Configuration conf,
    Properties tableProperties)
    throws SerDeException {

    final List<TypeInfo> columnTypes =
```

```

TypeInfoUtils.getTypeInfosFromTypeString(
    tableProperties.getProperty(LIST_COLUMN_TYPES));

    // verify types
    for (TypeInfo type : columnTypes) {
        if
(!type.getCategory().equals(PRIMITIVE) ||
        !((PrimitiveTypeInfo) type)

        .getPrimitiveCategory().equals(String)) {
            throw new SerDeException(
                "This SerDe only supports
strings.");
        }
    }

    columnNames =
Arrays.asList(tableProperties

        .getProperty(LIST_COLUMNS).split(", "));

    List<ObjectInspector>
columnObjectInspectors =
        Collections.nCopies(
            columnNames.size(),
            (ObjectInspector)
javaStringObjectInspector);

    objectInspector =
ObjectInspectorFactory
        .getStandardStructObjectInspector(
            columnNames,
            columnObjectInspectors);

    rowMap =
        new HashMap<String,
String>(columnNames.size());
    rowFields =
        new
ArrayList<String>(columnNames.size());
    stats = new SerDeStats();
    deserializedByteCount = 0;
}

```

3. The `serialize` method is called whenever a row should be written to HDFS. It is responsible for using the object inspector to extract the columnar data from the row and converting that data to a Hadoop `Writable` object:

```

@Override
public Writable serialize(
    Object obj,
    ObjectInspector objectInspector)

```

```

        throws SerDeException {

            StringBuilder builder = new
StringBuilder();

            StructObjectInspector structOI =
                (StructObjectInspector)
objectInspector;

            List<? extends StructField>
structFields =
                structOI.getAllStructFieldRefs();

            for (int i = 0; i <
structFields.size(); i++) {
                StructField structField =
structFields.get(i);

                Object fieldData =
                    structOI.getStructFieldData(obj,
structField);

                StringObjectInspector fieldOI =
                    (StringObjectInspector)

structField.getFieldObjectInspector();

                String fieldContent =

fieldOI.getPrimitiveJavaObject(fieldData);

                if (fieldContent != null) {
                    String fieldName =
columnNames.get(i);

                    if (builder.length() > 0) {
                        builder.append("\001");
                    }
                    builder.append(fieldName);
                    builder.append("\002");
                    builder.append(fieldContent);
                }
            }
            return new Text(builder.toString());
        }
    }

```

4. Similar to the serialize method, the deserialize method is responsible for converting a Hadoop `Writable` object back into columnar form. In this case, the input is a `Text` object containing our serialized map, which we break into keys and values and return as a `List of Strings`.

```

@Override
public Object deserialize(Writable
writable)

```

```

throws SerDeException {

    // reset internal state
    rowMap.clear();
    rowFields.clear();

    Text text = (Text) writable;
    String content = text.toString();

    // update stats
    deserializedByteCount +=
text.getBytes().length;

    // parse map
    String[] pairs = content.split("\001");
    for (String pair : pairs) {
        int delimiterIndex =
pair.indexOf('\002');

        if (delimiterIndex >= 0) {
            String key = pair.substring(0,
delimiterIndex);
            String value =
                pair.substring(delimiterIndex +
1);
            rowMap.put(key, value);
        }
    }

    for (String columnName : columnNames) {

rowFields.add(rowMap.get(columnName));
    }

    return rowFields;
}

```

5. The next two methods are used by Hive to describe the types used by this [SerDe](#). Object inspectors describe how Hive should interpret the tabular data and the serialized class describes how Hadoop should serialize the data to HDFS.

```

@Override
public ObjectInspector
getObjectInspector()
throws SerDeException {
    return objectInspector;
}

@Override
public Class<? extends Writable>
getSerializedClass() {
    return Text.class;
}

```

6. Finally, implementations can optionally record and report statistics about the data they are serializing and deserializing:

```
@Override
public SerDeStats getSerDeStats() {

    stats.setRawDataSize(deserializedByteCount);
    return stats;
}
}
```

7. This class depends on the Hadoop and Hive libraries, which can be found in `$HADOOP_HOME/` and `$HIVE_HOME/lib/`. Compile this class and package it into a standard JAR file. You can use your preferred Java development environment or simply use the following commands:

```
$ javac \
    -cp
    ${HADOOP_HOME}/'':${HIVE_HOME}/lib/'': \
    *.java

$ mkdir -p com/leeriggins/hive/

$ mv *.class com/leeriggins/hive/

$ jar cf udf.jar com
```

8. We can now use the SerDe to define a new table. The first step is dynamically adding our new jar file to the Hive classpath:

```
hive> add jar udf.jar ;
```

9. Now, we can create our tables and copy the data from the input table to the output table:

```
hive> create table if not exists city_bid(
    >   name string,
    >   id string,
    >   bidding_city string,
    >   olympic_games string)
    >   row format delimited
    >   fields terminated by '\t' ;

hive> load data
    >   local inpath
    'data/olympic_city_bid.tsv'
    >   overwrite into table city_bid ;

hive> create table city_bid_map(
    >   name string,
    >   id string,
    >   bidding_city string,
    >   olympic_games string)
```

```

> row format serde
>
'com.leeriggins.hive.ColumnarMapSerDe'
> location '/data/map_serde/' ;

hive> insert overwrite table city_bid_map
> select * from city_bid ;

hive> select * from city_bid_map limit 5 ;
/m/08lq3hx  Buenos Aires  2004 Summer
Olympics
/m/08lq3hp  Cape Town    2004 Summer
Olympics
/m/0k8jmd7  Moscow      2012 Summer Olympics
/m/08lq3j3  Stockholm    2004 Summer
Olympics
/m/04frxgv  Åre,Östersund  1994 Winter
Olympics

```

10. We can also verify that the files in HDFS were written in the correct format, as follows

```

$ $HADOOP_HOME/bin/hadoop fs -cat /data/
map_serde/'*' |
head -n 2 | less

name^B^Aid^B/m/08lq3hx^Abidding_city^BBuenos
Aires^Aolympic_games^B2004 Summer Olympics

name^B^Aid^B/m/08lq3hp^Abidding_city^BCape
Town^Aolympic_games^B2004 Summer Olympics

```

How it works...

The central part of this example is our implementation of the `ColumnarMapSerDe` class, which implements the `SerDe` interface. Our class describes how to transform a Hadoop record into the columns of a Hive table.

Object inspectors

The two main types involved with serialization and deserialization are `Writable` and `ObjectInspector`. `Writable` is a Hadoop type used to represent keys and values in Map/Reduce jobs. An `ObjectInspector` is a Hive type containing the necessary logic for converting between the various Hive representations of data and the more standard Java and Hadoop types.

There are multiple `ObjectInspector` implementations for each of the primitive and complex Hive types. These can be lazy or not and backed by Hadoop `Writable` objects or standard Java classes. Generally, using the

lazy versions or the versions backed by `Writable` object can be more efficient; however, using these object inspectors efficiently is more complicated than using the standard Java object inspectors.

The complex Hive types all contain nested object inspectors describing the types of their contents. Primitive Hive types are all represented by subtypes of `PrimitiveObjectInspector`.

Object inspectors should never be created directly; instead, Hive provides the `ObjectInspectorFactory` and `PrimitiveObjectInspectorFactory` classes that may be used to create instances.

Initialization

First, we need to initialize our SerDe. Hive strongly encourages reusing objects to reduce the need for garbage collection. Therefore, we will precompute as much as possible on initialization and store this information in instance variables.

In the initialize method, Hive will pass us the Hadoop configuration and table properties so we can configure our internal state for the specific table. We can get the names and types of each of the columns from the table properties. Since we are modeling a map of strings to strings, we will throw an exception if any of the columns are not strings. Any other SerDe properties, such as the `input_regex` used by the `RegexSerDe`, are also available from this table properties object.

For this SerDe, we want to store the column names so we can later extract the appropriate values from each row. We also want to create the `ObjectInspector` that describes this table. We will return a single `StructObjectInspector` whose fields correspond to the columns of our table. Each field should be a string, so we will use `StringObjectInspectors`.

Finally, we will initialize the instance variables that we will use during serialization and deserialization. Hive asks for the `Writable` class and `ObjectInspector` used by our SerDe by calling `getSerializedClass` and `getObjectInspector`.

Serialization

When serializing a row, we are given an object containing that row's data plus the `ObjectInspector` necessary to read the data from the object. Since this object represents an entire row, the object inspector will be a `StructObjectInspector` whose fields correspond to the columns of the table. The `StructObjectInspector` also contains a nested

`ObjectInspector` for each field; in this case, each will be a `StringObjectInspector`.

So, our `serialize` method needs to use each of these nested object inspectors to read each field, combine the data with the name of the column that we read at initialization time, and build a map string. In this case, we have hard-coded the `\001` and `\002` delimiters; however, these could also have been determined at initialization time from table properties.

Deserialization

The `deserialize` method reverses the serialization process. In this case, we want to convert the `Writable` object passed to us into a row. Deserialization is somewhat simpler as we can just return a `List` of `Strings`. Hive will use the `ObjectInspector` we return from `getObjectInspector` to convert this value into whatever internal representation it may decide to use.

Because we returned the `Text` class from the `getSerializedClass` method, we know that we can convert it into text, split it by our delimiters, and extract the values corresponding to our column names.

The `deserialize` method has one additional side effect, which is incrementing the number of bytes that we read during deserialization. These will be encapsulated in a `SerDeStats` and returned to the framework in the `getSerDeStats` method. Tracking this information is optional; a `SerDe` may simply always return zero for the amount of deserialized data.

Using static partitions (Intermediate)

As we saw in the previous optimization sections, there are large performance gains when Hive can restrict the query to a subset of the table. Using partitions is one way to tell Hive how data in a table is organized into files.

One common pattern is for an external process to load data into HDFS periodically. We can define a single external table for this data, and add a new partition for each data drop.

Getting ready

To start, we will artificially partition our input data into three batches and copy them to HDFS.

1. Split the input files into multiple parts:

```
$ split -a 1 -l 25000
data/olympic_athlete.tsv \
    data/olympic_athlete_split_

$ split -a 1 -l 7000
data/olympic_athlete_affiliation.tsv \
    data/olympic_athlete_affiliation_split_
```

2. Insert each of these parts into HDFS:

```
$ $HADOOP_HOME/bin/hadoop fs
-mkdir /data/athlete/

$ $HADOOP_HOME/bin/hadoop fs -put \
    data/olympic_athlete_split_* /data/athlete/

$ $HADOOP_HOME/bin/hadoop fs
-mkdir /data/athlete_affiliation/

$ $HADOOP_HOME/bin/hadoop fs -put \
    data/olympic_athlete_affiliation_split_*
\
    /data/athlete_affiliation/
```

How to do it...

In this example, we will be reading our data from an external table set up with static partitions.

1. We begin with creating the partitioned input tables and loading data into them:

```
hive> create external table if not exists
athlete(
  >   name string,
  >   id string,
  >   demonstration_events_competed_in
array<string>,
  >   demonstration_medals_won
array<string>,
  >   country array<string>,
  >   medals_won array<string>)
> partitioned by (part_key string)
> row format delimited
>   fields terminated by '\t'
>   collection items terminated by
',' ;

hive> alter table athlete add if not exists
  >   partition(part_key = 'a')
  >   location
'/data/athlete/part_key=a/' ;

hive> alter table athlete add if not exists
  >   partition(part_key = 'b')
  >   location
'/data/athlete/part_key=b/' ;

hive> alter table athlete add if not exists
  >   partition(part_key = 'c')
  >   location
'/data/athlete/part_key=c/' ;

hive> create external table if not exists
  > athlete_affiliation(
  >   name string,
  >   id string,
  >   athlete string,
  >   country string,
  >   olympics string,
  >   sport string)
  > partitioned by (part_key string)
  > row format delimited fields
terminated by '\t' ;

hive> alter table athlete_affiliation add
if not exists
  >   partition(part_key = 'a')
  >   location
'/data/athlete_affiliation/part_key=a/' ;
```

```
hive> alter table athlete_affiliation add
if not exists
    > partition(part_key = 'b')
    > location
'/data/athlete_affiliation/part_key=b/'
> ;

hive> alter table athlete_affiliation add
if not exists
    > partition(part_key = 'c')
    > location
'/data/athlete_affiliation/part_key=c/'
> ;
```

2. We can use the `explain dependency` prefix to see how Hive will restrict the amount of data processed by a query based on where clauses and join conditions:

```
hive> explain dependency
    > select * from athlete where part_key
= 'a' ;
{"input_partitions":
["ch9@athlete@part_key=a"], "input_tables":
[{"tablename": "ch9@athlete", "tabletype": "EXTERNAL_TABLE"}]}
```

```
hive> explain dependency
    > select
    >     l.name,
    >     size(l.medals_won),
    >     r.country,
    >     collect_set(r.olympics)
    > from athlete l
    > join athlete_affiliation r
    > on l.name = r.athlete
    > and l.part_key = 'a'
    > and r.part_key = 'a'
    > group by l.name, size(l.medals_won),
r.country
    > order by name asc ;
{"input_partitions":
["ch9@athlete@part_key=a", "ch9@athlete_affiliation@part_key=a"], "
[{"tablename": "ch9@athlete_affiliation", "tabletype": "EXTERNAL_TAB
{"tablename": "ch9@athlete", "tabletype": "EXTERNAL_TABLE"}]}
```

How it works...

To create a partitioned table, we simply specify the partition columns as part of the table definition.

The partition columns are considered to be additional columns of the athlete table and can be used like any other column in queries. For example, the

partition columns will be included in the output of `select *` queries and appear as part of the table description.

Using partitions, we can incrementally and efficiently add data to a table. In this example, each new partition points to a different directory in HDFS. This same pattern could be used to add a new partition for each day of data, for example.

The partition information is stored as part of the table metadata. We can see which partitions exist by issuing a `show partitions` statement:

```
hive> show partitions athlete ;
part_key=a
part_key=b
part_key=c
```

In addition to simplifying incremental data loading, Hive will restrict queries to only the necessary partitions whenever possible. In the two select statements in step 2, Hive only considers a single partition of each input table. This optimization makes queries on parts of large partitioned tables efficient.

There's more...

In addition to pointing Hive at a pre-existing file in HDFS, we can load data into a partition using a `load data` statement or as the output of a query.

Loading data into partitioned internal tables

To have Hive copy local data into particular partitions, we can simply append the partition to the end of the `load data` statement as follows:

```
hive> load data
      > local inpath 'data/olympic_athlete_split_a'
      > overwrite into table athlete partition
(part_key='a') ;
```

The `overwrite` directive only applies to any previous data in the specified partition; data in other partitions will not be deleted.

Writing data into specific partitions from queries

To write data to a specific partition of a table, we simply specify the partition as part of the `insert` clause:

```
hive> insert overwrite table athlete_names
      > partition (part_key = 'a')
      > select name
```

```
> from athlete  
> where part_key = 'a' ;
```

Because partition columns are considered to be columns of tables, a `select *` statement would read an extra column of data from the athlete table. However, we are already specifying the value for the partition columns as part of the `insert` clause, so we only `select` the remaining columns in the rest of the query.

As in the `load data` statements, the `overwrite` directive in this query only applies to the partition that we specified. Other partitions will not be overwritten.

Using dynamic partitions (Intermediate)

In the previous recipe, we saw how data can be loaded into specific partitions from both files and from queries. In this recipe, we will create a table of athlete names partitioned by country based on the data in the `athlete_affiliation` table.

How to do it...

If we want to determine the target partition for each row of output, we need to use dynamic partitioning.

1. By default, Hive prevents queries from writing too many files to HDFS, which can negatively impact the performance of the HDFS namenode. In particular, dynamic partitions will write at least one file for each partition, so we need to configure Hive to allow the queries in this section:

```
hive> set hive.exec.dynamic.partition.mode
= nonstrict ;
hive> set
hive.exec.max.dynamic.partitions.pernode =
5000 ;
```

2. We now define our input and output tables. The output table is partitioned by country:

```
hive> create table if not exists
athlete_affiliation(
  > name string,
  > id string,
  > athlete string,
  > country string,
  > olympics string,
  > sport string)
> row format delimited
> fields terminated by '\t'
> collection items terminated by
',' ;

hive> load data
  > local inpath
'data/olympic_athlete_affiliation.tsv'
  > overwrite into table
athlete_affiliation ;

hive> create table if not exists
athlete_by_country(
```

```
> name string,  
> id string)  
> partitioned by (country string) ;
```

3. Finally, we can write data into our partitioned table and verify the partition values. The partition column is simply the last column in our `select` statement:

```
hive> insert overwrite table  
athlete_by_country  
> partition (country)  
> select name, id, country  
> from athlete_affiliation ;  
  
hive> show partitions athlete_by_country ;  
country=Afghanistan  
country=Albania  
country=Algeria  
country=American Samoa
```

How it works...

Using dynamic partitions is very similar to using static partitions. We simply do not specify a value for our partition in the insert clause and treat the dynamic partition as another column appended to our table definition.

If we have multiple partition columns, we can use a mix of static and dynamic partitions in our query. The static partitions must precede the dynamic ones in both the table definition and the query.

One performance consideration in using dynamic partitions is that HDFS is not designed to handle huge numbers of files. The namenode stores an object for each block and each file in memory, and the memory necessary to track billions of objects exceeds what is currently reasonably available. Additionally, the Hive metastore needs to track information for each partition. Depending on the metastore backend, this limit may be reached much more quickly.

Hive uses two settings to protect against accidentally running up against these limits. The first is `hive.exec.dynamic.partition.mode`, which defaults to strict mode. Strict mode disallows queries where all partitions are dynamic. We need to set the partition mode to nonstrict for this query.

The second setting is `hive.exec.max.dynamic.partitions.pernode`, an explicit limit on the number of dynamic partitions allowed per node. The default value of 100 is often too small to be useful, so we set it to 5000.

Using functions (Simple)

Hive includes a long list of built-in functions. All functions fall into one of three categories:

- A **standard function** takes one or more columns from a single row and returns a single value. This value may be a primitive or complex type. A typical example would be the length function, which computes the length of a string.
- A **table generating function** takes one or more columns from a single row and may return multiple values in multiple rows. A typical example would be the explode function, which converts an array of values into many rows, each containing a single value of the array.
- An **aggregation function** takes one or more columns from multiple rows and returns a single value. A typical example would be the sum function, which computes the sum of values in a column.

In this recipe, we will demonstrate how to use each of these types of functions in queries.

How to do it...

This script shows how to interactively retrieve the list of all available functions, retrieve documentation about a specific function, and use these functions in queries.

1. We can list all functions available and retrieve documentation about any single function:

```
hive> show functions ;
!
!=
%
&
...

hive> describe function regexp_extract ;
regexp_extract(str, regexp[, idx]) -
extracts a group that matches regexp

hive> describe function extended
regexp_extract ;
regexp_extract(str, regexp[, idx]) -
extracts a group that matches regexp
Example:
> SELECT regexp_extract('100-200',
'(\d+)-(\d+)', 1) FROM src LIMIT 1;
```

```
'100'
```

2. Next, we set up the the `athlete` and `athlete_affiliation` tables that we will use in this example:

```
hive> create table if not exists athlete(  
  > name string,  
  > id string,  
  > demonstration_events_competed_in  
array<string>,  
  > demonstration_medals_won  
array<string>,  
  > country array<string>,  
  > medals_won array<string>)  
> row format delimited  
> fields terminated by '\t'  
> collection items terminated by  
, ;
```

```
hive> load data  
  > local inpath  
'data/olympic_athlete.tsv'  
  > overwrite into table athlete ;
```

```
hive> create table if not exists  
athlete_affiliation(  
  > name string,  
  > id string,  
  > athlete string,  
  > country string,  
  > olympics string,  
  > sport string)  
> row format delimited  
> fields terminated by '\t' ;
```

```
hive> load data  
  > local inpath  
'data/olympic_athlete_affiliation.tsv'  
  > overwrite into table  
athlete_affiliation ;
```

3. We can now use the `regexp_extract` function to parse a string with a regular expression and return the value of a particular group:

```
hive> select  
  > regexp_extract(name, '(\S+).*  
(\S+)', 1),  
  > regexp_extract(name, '(\S+).*  
(\S+)', 2)  
> from athlete  
> limit 5 ;  
Chris Hoy  
Annemarie Moser-Pröll  
Madara Palameika
```

Chris Snelling
Håkan Mild

4. The `explode` table generating function converts an array into multiple rows, each containing one value of the array. We need to use a `lateral view` to join the output of this function with the data in the original table. A `lateral view` has a syntax similar to join; we specify the function as the second table, give it a table alias, and provide column names for its output. The `explode` function also works on maps. The function will produce two columns containing the keys and their corresponding values.

```
hive> select
>   name,
>   medal_id
> from athlete
>   lateral view
>     explode(medals_won)
combined_tables as medal_id
> limit 5 ;
Chris Hoy /m/04gb_3f
Chris Hoy /m/07jn_ls
Chris Hoy /m/0kjf4vr
Chris Hoy /m/0kgbwtc
Chris Hoy /m/04hb914
```

5. As an alternative way of using table generating functions, if we do not need any data from the original table, we can simply use table generating functions directly.

```
hive> select explode(medals_won) as
medal_id
> from athlete limit 5 ;
/m/04gb_3f
/m/07jn_ls
/m/0kjf4vr
/m/0kgbwtc
/m/04hb914
```

6. Finally, the `avg` aggregation function computes the average of a numeric value. We use it here in combination with the `length` function to compute the average athlete's name length by country:

```
hive> select
>   country,
>   avg(length(athlete)) as
avg_name_length
> from athlete_affiliation
> group by country order by
avg_name_length desc
> limit 5 ;
Yemen 22.0
Oman 21.5
Mongolia 19.846153846153847
```

```
Brunei 19.666666666666668  
Iceland 19.54054054054054
```

How it works...

In Hive, any logic that operates on the actual values of cells is implemented as a function. The three categories of functions cover the different modes of transforming data in SQL: converting each input into a single value, converting each input into multiple values, and aggregating multiple inputs into a single value.

Hive includes a large number of built-in functions. Some of these are expected; no SQL system would be complete without standard aggregations such as `count`, `sum`, and `avg`.

Other functions are more specialized, such as the `sentences` function, which converts a block of text into an array of arrays where each array contains the tokens for a single sentence.

All of these functions are implemented using the same framework that is available to users to extend Hive. We will see how to use this framework in subsequent recipes.

Adding custom logic with streaming (Intermediate)

Hadoop Streaming is a feature that allows users to invoke external programs as part of their Map/Reduce job. Hive supports this functionality through **map** and **reduce** clauses. In this recipe, we will demonstrate how to we can use map and reduce code written in Ruby to compute the average length of all unique names in the athlete table.

Getting ready

This recipe assumes that you have Ruby installed and available at `/usr/bin/ruby` on all nodes of your cluster.

How to do it...

1. Our map script will tokenize each field of input and output both the token and the length of that token. We will call this script `token_lengths.rb`.

```
#!/usr/bin/ruby

STDIN.read.split("\n").each do |line|
  line.split("\t").each do |field|
    field.split(" ").each do |word|
      puts "#{word}\t#{word.length}"
    end
  end
end
```

2. Our reduce script will average the length of the input tokens. However, it will ignore any repeated inputs. We will call this script `average_unique_lengths.rb`.

```
#!/usr/bin/ruby

last_key = nil
total = 0
count = 0

STDIN.read.split("\n").each do |line|
  fields = line.split("\t")
  key = fields[0]
  length = fields[1]
  if (key != last_key)
    last_key = key
    total += length.to_i
    count += 1
  end
end
```

```
end

puts total.to_f / count
```

3. We are now ready to invoke these scripts in a query. First, we create the database and the input table as follows:

```
hive> create database if not exists ch12 ;

hive> use ch12 ;

hive> create table if not exists athlete(
  >   name string,
  >   id string,
  >   demonstration_events_competed_in
array<string>,
  >   demonstration_medals_won
array<string>,
  >   country array<string>,
  >   medals_won array<string>)
  > row format delimited
  >   fields terminated by '\t'
  >   collection items terminated by
',' ;

hive> load data
  > local inpath
'data/olympic_athlete.tsv'
  > overwrite into table athlete ;
```

4. Next, we add our two script files to the Hadoop distributed cache. This is the easiest way to ensure that the files are available on all nodes of our cluster.

```
hive> add file token_lengths.rb ;
Added resource: token_lengths.rb

hive> add file average_unique_lengths.rb ;
Added resource: average_unique_lengths.rb
```

5. Finally, we can invoke our scripts from within a query by using `map...using` and `reduce...using` clauses:

```
hive> from (
  >   map name
  >     using 'token_lengths.rb' as
(word string, freq int)
  >   from athlete
  >   cluster by word) t
  > reduce word, freq
  >   using 'average_unique_lengths.rb'
as (avg) ;
7.16527792807416
```

How it works...

When Hive calls our scripts, it will pass us each row through standard input with the columns separated with tabs.

Similarly, if we specify aliases, Hive will interpret each line of output as a new row with the values separated with tabs. The outputs will be coerced into the number and types of columns we specify as aliases. If the number of aliases does not match the number of columns output by the script, Hive will ignore extra values and fill missing values with nulls.

If we do not specify aliases, Hive will assume that we are returning key value pairs and interpret the result as two string columns. In this case, the first field is interpreted as the key and the remainder of the line, including all tabs and subsequent fields, is interpreted as the value.

In this particular example, the map script treats each input field identically by tokenizing them and outputting both the token and its length. The `reduce` script adds these lengths to a running total provided that the current word is different from the previous one. It outputs the average length after all lines have been read.

We need all rows with the same key to be sent to the reducer in order. We accomplish this using the **cluster by** clause in our query.

Simple user-defined functions (Intermediate)

As an alternative to using Hadoop Streaming support, we can also add custom logic by implementing a user-defined function. In this recipe, we will implement a user-defined function that computes the MD5 hash of an input string.

Getting ready

This example uses the Apache Commons Codec library. Version 1.4 used by both Hadoop and Hive and should be available at `$HADOOP_HOME/lib` and `$HIVE_HOME/lib`. It is also available at <http://commons.apache.org/codec/>.

How to do it...

The following steps will take you through the process:

1. A simple user-defined function should extend the `UDF` class and implement one or more evaluate methods. We need to compile this class and package it in a JAR file.

```
package com.leeriggins.hive;

// imports omitted

@Description(
    name = "md5sum",
    value = "_FUNC_(string) - computes the MD5 sum of the input string. Returns the hash in hexadecimal format.",
    extended = "Example: _FUNC_('foobar') = 3858f62230ac3c915f300c664312c63f")
public class MD5SumUDF extends UDF {

    public String evaluate(String input) {
        return DigestUtils.md5Hex(input);
    }

}
```

2. Once we have our JAR file, we are ready to use our UDF in Hive. We start by creating the `athlete_affiliation` table:

```
hive> create table if not exists
athlete_affiliation(
    > name string,
```

```

> id string,
> athlete string,
> country string,
> olympics string,
> sport string)
> row format delimited fields
terminated by '\t' ;

hive> load data
> local inpath
'data/olympic_athlete_affiliation.tsv'
> overwrite into table
athlete_affiliation ;

```

3. We now add the JAR file containing our user-defined function to the Hive classpath. Once the JAR file has been added, we can create a temporary function and use it like any other function.

```

hive> add jar udf.jar ;

hive> create temporary function md5sum
> as
'com.leeriggins.hive.MD5SumUDF' ;

hive> describe function extended md5sum ;
md5sum(string) - computes the MD5 sum of
the input string. Returns the hash in
hexidecimal format.
Example: md5sum('foobar') =
3858f62230ac3c915f300c664312c63f

hive> select athlete, md5sum(athlete)
> from athlete_affiliation
> limit 5 ;
Robert David Webster
becc84ccd79ea95838164770c3a866c0
Richard Carroll Conner
1207d521993cf22d223db7672b02a006
Robert Haigh
0bc0eb198a5cdc6ea9d2fb47de68cba5
Moss Christie
f8570740013fa1f2d67de5204931fa7e
Danni Roche
b2531ee10509383318e885ed448d88a9

```

How it works...

The easiest way to implement a user-defined function is to extend the `UDF` class. We simply need to implement one or more `evaluate` methods.

When used in a query, Hive will use reflection to find a matching `evaluate` method. Hive will attempt to coerce values when necessary (for example,

between doubles and integers). Having multiple `evaluate` methods allows a single UDF to accept variable argument numbers and types.

To use our function in Hive, we package our compiled class into a JAR file, add it to the Hive classpath, and define it as a temporary function. All user-defined functions are temporary, meaning that they will not persist across Hive sessions. If we have common functions that we want to be always available, we can put the `add jar` and `create temporary function` statements into an initialization file.

To document our function, we can optionally annotate the source code with a `@Description`. These will be available as part of the function description inside of Hive.

Advanced user-defined functions (Advanced)

By extending `GenericUDF` instead of `UDF`, we can also add logic that will run when the query is parsed. This allows us to decide dynamically whether to accept the types of arguments we are passed.

In this recipe, we will rewrite our `md5sum` function to accept any number of either string or binary arguments.

How to do it...

A `GenericUDF` has considerably more flexibility than a `UDF`; however, the implementation is also more complex.

1. We start by extending `GenericUDF` and adding a `Description` to document our function:

```
package com.leeriggins.hive;

// imports omitted

@Description(
    name = "md5sum",
    value = "_FUNC_(string|byte, string|byte, ...) - computes the MD5 sum of the inputs. Returns the hash in hexadecimal format.",
    extended = "Example: _FUNC_('foo', 'bar') = 3858f62230ac3c915f300c664312c63f")
public class GenericMD5SumUDF extends GenericUDF {
```

2. The `initialize` method will be called when the query is parsed. It is responsible for validating the inputs to the function and describing the outputs. As with a custom `SerDe`, inputs and outputs are handled using object inspectors. We expect `String` and `Byte` object inspectors and return a `String` object inspector representing our output.

```
    private PrimitiveObjectInspector[] inputOIs;

    @Override
    public ObjectInspector initialize(
        ObjectInspector[] args)
        throws UDFArgumentException {
        if (args == null || args.length == 0) {
```

```

        throw new UDFArgumentLengthException(
            "Must supply at least one
argument.");
    }
    inputOIs = new
PrimitiveObjectInspector[args.length];

    for (int i = 0; i < args.length; i++) {
        ObjectInspector objectInspector =
args[i];
        if (objectInspector

.getCategory().equals(PRIMITIVE)) {
            PrimitiveObjectInspector oi =
            (PrimitiveObjectInspector)
objectInspector;
            PrimitiveCategory cat =
            oi.getPrimitiveCategory();
            if (cat.equals(STRING) ||
cat.equals(BINARY)) {
                inputOIs[i] = oi;
            } else {
                throw new
UDFArgumentTypeException(i,
                    String.format(
                        "Expected a string or byte
argument," +
                        "received a %s at index %d",
                        oi.getPrimitiveCategory(),
i));
            }
        } else {
            throw new
UDFArgumentTypeException(i,
                String.format(
                    "Expected a string or byte
argument," +
                    "received a %s at index %d",
                    objectInspector.getCategory(),
i));
        }
    }

    return javaStringObjectInspector;
}

```

3. The `evaluate` method is called once per row and is responsible for actually calculating the hash of the inputs. We use the object inspector that we received in the `initialize` call to convert the `DeferredObjects` into `Strings` or `Byte` arrays.

```

@Override
public Object evaluate(DeferredObject[]
args)
throws HiveException {

```

```

        try {
            MessageDigest digest =
                MessageDigest.getInstance("MD5");
            for (int i = 0; i < args.length;
i++) {
                DeferredObject arg = args[i];
                PrimitiveObjectInspector inspector
= inputOIs[i];
                if (inspector

                .getPrimitiveCategory().equals(String)) {
                    String content = (String)

inspector.getPrimitiveJavaObject(arg.get());

                digest.update(content.getBytes());
                } else {
                    ByteArrayRef ref = (ByteArrayRef)

inspector.getPrimitiveJavaObject(arg.get());
                    digest.update(ref.getData());
                }
            }
            return
Hex.encodeHexString(digest.digest());
        } catch (NoSuchAlgorithmException e) {
            throw new HiveException(e);
        }
    }
}

```

4. The `getDisplayString` method is called when a query is explained.

```

@Override
public String getDisplayString(String[]
args) {
    StringBuilder builder = new
StringBuilder();
    builder.append("md5sum(");
    for (String arg : args) {
        builder.append(arg).append(",");
    }
    builder.delete(builder.length() - 1,
builder.length());
    builder.append(")");
    return builder.toString();
}
}

```

5. Using this function is no different than using a standard UDF. As before, we start by compiling this class and packaging it into a JAR file.
6. Next, we add the JAR file to Hive, create the function, and use it in queries in the same way as with a simple user-defined function.

```

hive> create table if not exists
athlete_affiliation(
    > name string,
    > id string,
    > athlete string,
    > country string,
    > olympics string,
    > sport string)
    > row format delimited fields
terminated by '\t' ;

hive> load data
    > local inpath
'data/olympic_athlete_affiliation.tsv'
    > overwrite into table
athlete_affiliation ;

hive> add jar udf.jar ;

hive> create temporary function
generic_md5sum
    > as
'com.leeriggins.hive.GenericMD5SumUDF' ;

hive> select
    > athlete,
    > country,
    > generic_md5sum(athlete),
    > generic_md5sum(athlete, country)
    > from athlete_affiliation
    > limit 3 ;
Robert David Webster United States of
America becc84ccd79ea95838164770c3a866c0
e7cb6846df94caff0d92d21aa9fa915a
Richard Carroll Conner United States of
America 1207d521993cf22d223db7672b02a006
3e5f43086ddc41c8985fdd042cd4c227
Robert Haigh Australia
0bc0eb198a5cdc6ea9d2fb47de68cba5
d4fc00546b3c61b6413e3afd6a9062a9

```

How it works...

With an ordinary UDF, we only needed to implement an `evaluate` method. With a `GenericUDF`, we have several methods that we need to implement.

Initialization

Hive will pass the `initialize` method the object inspectors that describe the input types to the query. Our first task is to examine these object inspectors to see if the input types are compatible with our function. If not,

we should throw a `UDFArgumentLengthException` or `UDFArgumentTypeException`.

We also save these object inspectors in an instance variable for use during evaluation.

This method should return an `ObjectInspector` that describes the output type of our user-defined function. For our `md5sum` function, we will return a hex string, so we choose the `javaStringObjectInspector` available from the `PrimitiveObjectInspectorFactory`.

Evaluation

For each row, Hive will call `evaluate` with the values of our input columns. These inputs are wrapped in deferred objects, so we can lazily retrieve only those values needed for the current row. The actual values can be converted into strings and byte arrays using the object inspectors we received during initialization.

The display string

The final method, `getDisplayString`, is used by Hive to describe our function in `explain` statements. It will receive the display strings of its arguments and is responsible for outputting a display string representing the entire function invocation.

User-defined table-generating functions (Advanced)

A **user-defined table generating function (UDTF)** has the ability to output any number of fields and any number of rows for each row of input. Writing one is very similar to writing a generic user-defined function.

In this recipe, we will write a table generating function that outputs the character bigrams, or pairs of consecutive characters in athlete names. This could be the first step in building a "type-ahead" function for inputting athlete names or a predictive algorithm for classifying athletes by their country affiliation.

How to do it...

You need to follow these steps:

1. We start with `CharBigramsUDTF.java`, which contains the implementation of our UDTF. We extend `GenericUDTF` and add a `Description` for documentation.

```
package com.leeriggins.hive;

// imports omitted

@Description(
    name="char_bigram",
    value="_FUNC_(string) - outputs each
    pair of consecutive characters in each
    word of the input string")
public class CharBigramsUDTF extends
    GenericUDTF {

    private StringObjectInspector inputOI;
```

2. As with a `GenericUDF`, the `initialize` method is called when the query is first parsed. It is responsible for validating the inputs and specifying the outputs of our function.

```
@Override
public StructObjectInspector initialize(
    ObjectInspector[] args) throws
    UDFArgumentException {

    if (args.length != 1) {
        throw new UDFArgumentLengthException(
            "Expected 1 argument, received " +
            args.length);
    }
```

```

        ObjectInspector oi = args[0];
        if (oi.getCategory() != PRIMITIVE ||
            ((PrimitiveObjectInspector) oi)
                .getPrimitiveCategory() !=
STRING) {
            throw new
UDFArgumentTypeException(0,
                "Argument must be a string.");
        }

        inputOI = (StringObjectInspector) oi;

        List<String> fieldNames =
Arrays.asList("bigram");
        List<ObjectInspector> fieldOIs = Arrays
            .<ObjectInspector> asList(
                javaStringObjectInspector);
        return ObjectInspectorFactory
            .getStandardStructObjectInspector(
                fieldNames, fieldOIs);
    }

```

3. The `process` method is analogous to the `evaluate` method of a `GenericUDF`. Unlike `evaluate`, we may return any number of rows of output for each row of input, including zero. We do this by calling the `forward` method for each row of output.

```

@Override
public void process(Object[] args)
throws HiveException {
    String content =

        inputOI.getPrimitiveJavaObject(args[0]);
        String[] words = content.split("\\s+");
        for (String word : words) {
            if (word.length() > 1) {
                for (int i = 0; i < word.length()
- 2; i++) {
                    forward(new Object[] {
                        word.substring(i, i + 2)
                    });
                }
            }
        }
    }
}

```

4. The `close` method is called once all rows of input have been passed to the `process` method. For our function, there are no rows to `forward` or other cleanup to do here.

```

@Override
public void close() throws HiveException
{

```

```

        // do nothing
    }
}

```

5. Just as with a standard user-defined function, we must compile and package this class into a jar file.
6. Once we have a compiled JAR file, we can use our table generating function to compare common bigrams across countries.

```

hive> create table if not exists
athlete_affiliation(
    > name string,
    > id string,
    > athlete string,
    > country string,
    > olympics string,
    > sport string)
    > row format delimited fields
terminated by '\t' ;

hive> load data
    > local inpath
'data/olympic_athlete_affiliation.tsv'
    > overwrite into table
athlete_affiliation ;

hive> add jar udf.jar ;

hive> create temporary function
char_bigrams
    > as
'com.leeriggins.hive.CharBigramsUDTF' ;

hive> create table if not exists
bigram_freq_by_country(
    > country string,
    > bigram string,
    > freq int) ;

hive> insert overwrite table
bigram_freq_by_country
    > select country, bigram, count(1) as
freq
    > from athlete_affiliation
    > lateral view
    > char_bigrams(lower(athlete)) t as
bigram
    > group by country, bigram ;

hive> select bigram, freq
    > from bigram_freq_by_country
    > where country = "United States of
America"

```

```

        > order by freq desc limit 4 ;
ar  384
an  343
ri  289
er  272

hive> select bigram, freq
      > from bigram_freq_by_country
      > where country = "China"
      > order by freq desc limit 4 ;
an  280
in  134
ia  133
zh  123

hive> select bigram, freq
      > from bigram_freq_by_country
      > where country = "Russia"
      > order by freq desc limit 4 ;
an  254
ov  227
ko  199
al  199

```

From this analysis, we can see that athletes from the United States of America, China, and Russia all are likely to have the bigram `an` in their name. However, American athlete names are more likely to contain `ri` and `er`. The bigrams `ia` and `zh` are more common for Chinese athletes, and the bigrams `ov`, `ko`, and `al` are more common for Russian athletes.

How it works...

Writing a user-defined table generating function is very similar to writing a generic user-defined function. The three main phases of a UDTF are initialization, processing each row of the inputs, and optionally writing any final output.

Initialization

Like a generic UDF, the first step is writing an initialization method that validates the inputs and specifies the outputs. Since a UDTF may return multiple columns, the output is always a `StructObjectInspector` whose fields represent the types of each column. The names of each field are not used by Hive because the user will specify aliases for each one as part of the query.

In this example, we verify that we are given a single `StringObjectInspector`, store it for later use, and return a

`StructObjectInspector` containing a single string field that will contain the bigram.

Processing inputs

The logic of a UDTF lives in the `process` method. In this example, we use the input object inspector to read the content of the input, split it into words, and then iterate through the consecutive pairs of characters in each word. To output a row, we simply call the `forward` method and pass it an object array whose contents match the types we specified in our output object inspector.

Final output

The UDTF class also provides a `close` method that will be called after all rows have been processed by each mapper. We can use this method to forward any additional rows or to clean up any resources used by the UDTF. In this case, we have nothing extra to do, so we simply provide an empty implementation.

User-defined aggregation functions (Advanced)

In this recipe, we will implement a generic user-defined aggregation function that computes a simple linear regression. Standard functions and table generating functions both operate in the map phase. A reducer can run in both the map and the reduce phases; this adds some complexity to the implementation.

Getting ready

We will use a synthetic dataset as input. Create a file named `points.tsv` in the data subdirectory with the following contents:

```
0.91      16.19
10.86     246.40
21.00     475.88
30.68     705.55
41.30     936.11
50.65     1166.10
61.31     1396.09
70.56     1626.13
80.97     1856.33
90.56     2086.21
```

The two numbers in each line should be separated by a single tab character.

How to do it...

The following steps will give you a better idea about aggregation functions:

1. Unlike the other user-defined functions that we have implemented, aggregation functions are split into a resolver and one or more evaluators. For the resolver, we start by extending the `AbstractGenericUDAFResolver` class and documenting our function with a `Description`.

```
package com.leeriggins.hive;

// imports omitted

@Description(
    name="regression",
    value="_FUNC_(double x,double y) -
    computes the simple linear regression")
public class SimpleRegressionUDAF
```

```
extends AbstractGenericUDAFResolver {
```

2. The `getEvaluator` method takes the role of the `initialize` method in other user-defined functions. It is called when a query is parsed, and is responsible for validating the inputs and deciding which evaluation implementation should be used. In our case, we only have one possible evaluation implementation.

```
@Override
public GenericUDAFEvaluator getEvaluator(
    GenericUDAFParameterInfo info)
    throws SemanticException {

    ObjectInspector[] inputOIs =
        info.getParameterObjectInspectors();

    if (inputOIs.length != 2) {
        throw new UDFArgumentLengthException(
            "Expected 2 arguments, received " +
            inputOIs.length);
    }

    for (int i = 0; i < inputOIs.length;
i++) {
        ObjectInspector oi = inputOIs[i];
        if (oi.getCategory() != PRIMITIVE ||
            ((PrimitiveObjectInspector) oi)
                .getPrimitiveCategory() !=
DOUBLE) {
            throw new
UDFArgumentTypeException(i,
                "This function requires double
arguments.");
        }
    }

    return new
SimpleRegressionUDAFEvaluator();
}
```

3. When UDAFs have only a single evaluator, the evaluator and its support classes are implemented as static nested class of the resolver. This aggregation buffer will be used to hold the state of the aggregation as we process each row.

```
private static class
SimpleRegressionAggregationBuffer
    implements AggregationBuffer {
    double sumX = 0.0;
    double sumX2 = 0.0;
    double sumY = 0.0;
    double sumXY = 0.0;
    int n = 0;
}
```

4. The `SimpleRegressionUDAFEvaluator` class holds our evaluation implementation:

```
public static class
SimpleRegressionUDAFEvaluator
    extends GenericUDAFEvaluator {
```

5. Aggregation occurs in several distinct stages, each of which expects different types of inputs and outputs. When changing stages, Hive will call the evaluator's `init` method with the current `Mode` and input types. It expects the evaluator to specify the output types for that stage. We will explore these different stages in detail in the *How it works...* section.

```
    private DoubleObjectInspector[]
originalDataOIs
    = new DoubleObjectInspector[2];
    private ListObjectInspector
partialDataOI;
    private ObjectInspector doubleListOI
    = ObjectInspectorFactory
        .getStandardListObjectInspector(
            javaDoubleObjectInspector);

    @Override
    public ObjectInspector init(
        Mode m,
        ObjectInspector[] parameters)
        throws HiveException {

        super.init(m, parameters);
        if (m.equals(Mode.PARTIAL1)) {

            processOriginalDataObjectInspectors(parameters);
            return doubleListOI;
        } else if (m.equals(Mode.COMPLETE)) {

            processOriginalDataObjectInspectors(parameters);
            return createOutputOI();
        } else if (m.equals(Mode.PARTIAL2)) {

            processPartialDataObjectInspectors(parameters);
            return doubleListOI;
        } else if (m.equals(Mode.FINAL)) {

            processPartialDataObjectInspectors(parameters);
            return createOutputOI();
        }
        throw new
IllegalStateException("Unknown mode!");
    }

    private void
processOriginalDataObjectInspectors(
```

```

        ObjectInspector[] parameters)
        throws HiveException {
            originalDataOIs[0] =
                (DoubleObjectInspector)
parameters[0];
            originalDataOIs[1] =
                (DoubleObjectInspector)
parameters[1];
        }

        private void
processPartialDataObjectInspectors(
        ObjectInspector[] parameters)
        throws HiveException {
            partialDataOI =
                (ListObjectInspector)
parameters[0];
        }

        private StructObjectInspector
createOutputOI() {
            List<String> fieldNames
                = Arrays.asList("slope",
"intercept");
            List<ObjectInspector> fieldOIs
                = Collections.nCopies(2,
(ObjectInspector)
                javaDoubleObjectInspector);
            return ObjectInspectorFactory

                .getStandardStructObjectInspector(fieldNames,
                fieldOIs);
        }

```

6. Hive stores the state of the aggregation in a user-defined aggregation buffer. It uses this method whenever it needs to request a new one.

```

@Override
public AggregationBuffer
getNewAggregationBuffer()
    throws HiveException {
    return new
SimpleRegressionAggregationBuffer();
}

```

7. The `iterate` method is responsible for updating the state of the current aggregation buffer based on a row of inputs:

```

@Override
public void iterate(
    AggregationBuffer buffer,
    Object[] inputs)
    throws HiveException {

```

```

        SimpleRegressionAggregationBuffer
buff
    =
    (SimpleRegressionAggregationBuffer) buffer;
        double x = (Double)
originalDataOIs[0]
        .getPrimitiveJavaObject(inputs[0]);
        double y = (Double)
originalDataOIs[1]
        .getPrimitiveJavaObject(inputs[1]);

        buff.sumX += x;
        buff.sumY += y;
        buff.sumX2 += x * x;
        buff.sumXY += x * y;
        buff.n += 1;
    }

```

8. This method is used by Hive to serialize our user-defined aggregation buffer to transfer the state to a reducer, for example. For simplicity, we serialize our data as a simple array of doubles.

```

@Override
public Object terminatePartial(
    AggregationBuffer buffer)
    throws HiveException {
    SimpleRegressionAggregationBuffer
buff
    =
    (SimpleRegressionAggregationBuffer) buffer;
    return new Double[] {
        buff.sumX,
        buff.sumY,
        buff.sumX2,
        buff.sumXY,
        (double) buff.n };
    }

    private double getDouble(Object obj,
int index) {
        DoubleObjectInspector elemOI =
            (DoubleObjectInspector)

        partialDataOI.getListElementObjectInspector();
        Object field =
            partialDataOI.getListElement(obj,
index);
        return
            (Double)
            elemOI.getPrimitiveJavaObject(field);
    }

```

9. This method is used by Hive to deserialize our aggregation buffers and merge them into a single buffer. Reducers will commonly use this method to combine the partial results from each mapper.

```

        @Override
        public void merge(AggregationBuffer
buf, Object agg)
            throws HiveException {
            SimpleRegressionAggregationBuffer
buffer
            =
            (SimpleRegressionAggregationBuffer) buf;
            buffer.sumX += getDouble(agg, 0);
            buffer.sumY += getDouble(agg, 1);
            buffer.sumX2 += getDouble(agg, 2);
            buffer.sumXY += getDouble(agg, 3);
            buffer.n += getDouble(agg, 4);
        }

```

10. Hive prefers to reuse objects whenever possible. This method resets the state of our user-defined aggregation buffer.

```

        @Override
        public void reset(AggregationBuffer
buffer)
            throws HiveException {
            SimpleRegressionAggregationBuffer
aggBuffer
            =
            (SimpleRegressionAggregationBuffer) buffer;
            aggBuffer.sumX = 0.0;
            aggBuffer.sumX2 = 0.0;
            aggBuffer.sumY = 0.0;
            aggBuffer.sumXY = 0.0;
            aggBuffer.n = 0;
        }

```

11. Finally, the `terminate` method is called when all aggregation is complete and Hive is ready for the final output. In our case, we use the data stored in our aggregation buffer to compute the slope and intercept of the regression line.

```

        @Override
        public Object
terminate(AggregationBuffer buffer)
            throws HiveException {
            SimpleRegressionAggregationBuffer
aggBuffer
            =
            (SimpleRegressionAggregationBuffer) buffer;

            double numerator = aggBuffer.sumXY
- aggBuffer.sumX *
aggBuffer.sumY / aggBuffer.n;
            double denominator = aggBuffer.sumX2
- aggBuffer.sumX *
aggBuffer.sumX / aggBuffer.n;

```

```

        double slope = numerator /
denominator;
        double intercept = aggBuffer.sumY /
aggBuffer.n
        - slope * aggBuffer.sumX /
aggBuffer.n;

        return new Double[] { slope,
intercept };
    }
}
}

```

12. As with any other user-defined function, the next step is compiling and packaging our class into a JAR file.
13. We can then add this JAR file, define the function, and use it like any other aggregation function.

```

hive> create table if not exists points(
>   x double,
>   y double)
> row format delimited fields
terminated by '\t' ;

hive> load data
> local inpath 'data/points.tsv'
> overwrite into table points ;

hive> add jar udf.jar ;

hive> create temporary function regression
> as
'com.leeriggins.hive.SimpleRegressionUDAF' ;

hive> describe function regression ;
regression(double x,double y) - computes
the simple linear regression

hive> select regression(x,y) from points ;
{"slope":23.046390237415224,"intercept":-
6.269384092610835}

```

How it works...

A user-defined aggregation function actually consists of multiple classes: a resolver and one or more evaluators. The resolver is responsible for verifying the argument types and returning an appropriate evaluator. The evaluator contains the actual logic for aggregating the data.

The resolver

The resolver should extend `AbstractGenericUDAFResolver` and implement the `getResolver` method.

Typically, there will be a single evaluator implemented as a static nested class of the resolver; however, we can decide to use different implementations depending on the arguments. As an example, we could extend our simple linear regression UDAF to support multiple regression and return an evaluator that does vector-based aggregation whenever more than two arguments are present.

The modes of evaluation

The evaluator should extend `GenericUDAFEvaluator`. It will contain the actual logic of our aggregation.

Our evaluator needs to support four modes of operation. They are as follows:

- In **partial1** mode, the evaluator is passed the original input data and should output a partial aggregation result.
- In **partial2** mode, the evaluator merges multiple partial aggregation results together.
- In **final** mode, the evaluator merges the remaining partial aggregation results together and outputs the final result.
- In **complete** mode, the evaluator aggregates the original input data and outputs the final result.

For any query, either the first three modes will be used together to distribute the aggregation across multiple Hadoop tasks or the complete mode will be used when the aggregation takes place in a single Hadoop task.

In this example, we are computing a simple linear regression. We expect to operate on two columns of doubles representing the coordinates of our sample data, and we will output the slope and y-intercept of the best-fit line. To calculate these values, we need to compute the sums of each column, the sum of the product of the x and y coordinates, the sum of the square of the x coordinate, and the number of total points in the sample.

So, our input types will be two doubles. Our final output will be a struct containing a "slope" and an "intercept" field, both of which are also doubles. For the partial aggregation, we will simply use a list of doubles to hold the running sums and counts we need to compute the final answer.

Initializing the evaluator

Before an evaluator is used, Hive will call its `init` method with the current mode and the input object inspectors used for that mode. We are

responsible for returning an object inspector representing the result of that mode.

- In **partial1** and **complete** modes, we are consuming the original input data, so we should expect object inspectors representing our original inputs, a pair of doubles.
- In **partial2** and **final** modes, we are consuming our own partial aggregation results, so we should expect an object inspector representing the partial aggregation type that we defined, a list of doubles.
- In **partial1** and **partial2** modes, we are outputting partial aggregations, so we should return an object inspector representing a list of doubles.
- In **final** and **complete** modes, we are outputting the final result, so we should return an object inspector representing our struct.

We must also call `super.init` to set the mode correctly in the parent class; failing to do so can result in very confusing behavior as the wrong methods may be called in each phase.

Aggregation buffers

To promote reusing logic across different modes, Hive adds an `AggregationBuffer` interface, which is conceptually similar to a partial aggregation result except that it lives only within a single Hadoop task. In particular, it does not need to be serialized, and so does not need to be defined using object inspectors.

Our aggregation buffer could simply hold a list of doubles, which is what we use for our partial aggregation. However, for increased readability, we will instead use a struct-like class `SimpleRegressionAggregationBuffer`.

To create a new aggregation buffer, Hive will call our `getNewAggregationBuffer` method. Hive will also reuse these buffers whenever possible. Before being reused, Hive will pass the buffer to the `reset` method.

The logic of aggregation

The logic for computing our aggregation is spread across four methods.

- In the `iterate` method, we are given the current aggregation buffer and the fields for a single row. We must extract the data from these fields and update our buffer appropriately. This is the only method that directly handles the original row data, and it will be called in both `partial1` and `complete` modes.
- In `partial1` and `partial2` modes, the aggregation buffers must be converted into partial aggregations so they may be serialized and

potentially sent to other nodes. This is done in the `terminatePartial` method. In this example, we simply move the values in our aggregation buffer into an array of doubles.

- Nodes may receive a partial aggregation in either `partial2` or `final` modes. Our UDAF needs to merge these partial aggregations back into an aggregation buffer. We do this in the `merge` method.
- Finally, once all of the inputs have been condensed into a single aggregation buffer, the `terminate` method will use this information to generate the final output. This method will be called in the `complete` and `final` modes.

There's more...

Just as extending `UDF` instead of `GenericUDF` is a simpler and more restricted method of writing user-defined functions, we can extend `UDAF` instead of `GenericUDAF` for less complex aggregation functions. However, we pay a performance cost as Hive will use reflection and attempt to serialize our aggregation buffer automatically.

As a note of warning, the documentation for the `UDAF` class is currently out of date. The documentation indicates that the methods should be named `aggregate` and `evaluate` instead of `iterate` and `terminate`.