Apache Hive Essentials

Immerse yourself on a fantastic journey to discover the attributes of big data by using Hive

Dayong Du

Apache Hive Essentials

Table of Contents

Apache Hive Essentials

Apache Hive Essentials

Credits

Author

Dayong Du

Reviewers

Puneetha B M

Hamzeh Khazaei

Nitin Pradeep Kumar

Balaswamy Vaddeman

Commissioning Editor

Ashwin Nair

Acquisition Editor

Shaon Basu

Content Development Editor

Merwyn D'souza

Technical Editor

Taabish Khan

Copy Editors

Sameen Siddiqui

Laxmi Subramanian

Project Coordinator

Neha Bhatnagar

Proofreaders

Paul Hindle

Jonathan Todd

Indexer

Monica Ajmera Mehta

Production Coordinator

Aparna Bhagat

Cover Work

Aparna Bhagat

About the Author

Dayong Du is a big data practitioner, leader, and developer with expertise in technology consulting, designing, and implementing enterprise big data solutions. With more than 10 years of experience in enterprise data warehouse, business intelligence, and big data and analytics, he has provided his data intelligence expertise in various industries, such as media, travel, telecommunications, and so on. He is currently working with QuickPlay Media in Toronto, Canada, to build enterprise big data intelligence reporting for online media services and content providers. He has a master's degree in computer science from Dalhousie University, and he holds the Cloudera Certified Developer for Apache Hadoop certification.

About the Reviewers

Puneetha B M is a software engineer, data enthusiast, and technical blogger. Her research interests include big data, cloud computing, machine learning, and NoSQL databases. She is also a professional software engineer with more than 2 years of working experience. She holds a master's degree in computer applications from P.E.S. Institute of Technology. Other than programming, she enjoys painting and listening to music. You can learn more from her blog (http://blog.puneethabm.in/) and LinkedIn profile (https://www.linkedin.com/in/puneethabm).

I owe a great deal to Prof. Dr. Ram Rustagi for being a role model in my life and for his zealous inspiration. I would like to thank my brother, Nischith B.M., for supporting me in everything I do. I would also like to thank Packt Publishing and its staff for providing the opportunity to contribute to this book.

Hamzeh Khazaei is a postdoctoral research scientist at IBM Canada Research and Development Centre. He received his PhD degree in computer science from University of Manitoba, Winnipeg, Manitoba, Canada (2009–2012). Earlier, he received both his BSc and MSc degrees in computer science from Amirkabir University of Technology, Tehran, Iran (2000–2008). He is also a sessional instructor in the Computer Science department at Ryerson University (http://scs.ryerson.ca/~hkhazaei). He teaches software engineering to fourth year undergraduate students. His research area includes big data analytics, cloud computing infrastructure, analytics as a service, and modeling of computing systems.

I would like to thank my dear wife for her perpetual support in all my endeavors.

Nitin Pradeep Kumar is a passionate developer with extensive experience and oodles of interest in emerging technologies such as the cloud and mobile. He is currently a cloud quality engineer at Appcelerator, a leading Silicon Valley-based start-up that provides an MBaaS platform purpose-built for mobile and cloud development. Before this stint, he studied at the National University of Singapore toward a master's degree in knowledge engineering, which involves building intelligent systems using cutting-edge artificial intelligence and data-mining techniques. He enjoys the start-up environment and has worked with technologies such as Hadoop, Hive, and data warehousing. He lives in Singapore and spends his spare cycles playing retro PC games on his mobile and learning Muay Thai.

I would like to thank my family, friends, and my wonderful brother, Nivin, for supporting me in all my endeavors.

Balaswamy Vaddeman is a Hadoop hackathon winner for Hyderabad in 2013. He is one of the top contributors on the Hive tag at http://www.stackoverflow.com. He is a big data professional with 3 years of experience. He is well known for training people on big data/Hadoop. So far, he has delivered six big data projects. He is a Java/J2EE expert with 8 years of IT experience and 5 years of RDBMS experience. He is an automation expert on Unix-based systems using Shell scripting. He has experience in setting up teams and bringing them up to speed on big data projects. He is an active participant in Hadoop/big

data forums.

www.PacktPub.com

Support files, eBooks, discount offers, and more

For support files and downloads related to your book, please visit www.PacktPub.com.

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> 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.

https://www2.packtpub.com/books/subscription/packtlib

Do you need instant solutions to your IT questions? PacktLib is Packt's online digital book library. Here, you can search, access, and read 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 a 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 9 entirely free books. Simply use your login credentials for immediate access.

I dedicate this book to my daughter

Preface

With an increasing interest in big data analysis, Hive over Hadoop becomes a cutting-edge data solution for storing, computing, and analyzing big data. The SQL-like syntax makes Hive easier to learn and popularly accepted as a standard for interactive SQL queries over big data. The variety of features available within Hive provides us with the capability of doing complex big data analysis without advanced coding skills. The maturity of Hive lets it gradually merge and share its valuable architecture and functionalities across different computing frameworks beyond Hadoop.

Apache Hive Essentials prepares your journey to big data by covering the introduction of backgrounds and concepts in the big data domain along with the process of setting up and getting familiar with your Hive working environment in the first two chapters. In the next four chapters, the book guides you through discovering and transforming the value behind big data by examples and skills of Hive query languages. In the last four chapters, the book highlights well-selected and advanced topics, such as performance, security, and extensions as exciting adventures for this worthwhile big data journey.

What this book covers

Chapter 1, *Overview of Big Data and Hive,* introduces the evolution of big data, the Hadoop ecosystem, and Hive. You will also learn the Hive architecture and the advantages of using Hive in big data analysis.

Chapter 2, *Setting Up the Hive Environment,* describes the Hive environment setup and configuration. It also covers using Hive through the command line and development tools.

Chapter 3, *Data Definition and Description,* introduces the basic data types and data definition language for tables, partitions, buckets, and views in Hive.

Chapter 4, *Data Selection and Scope,* shows you ways to discover the data by querying, linking, and scoping the data in Hive.

Chapter 5, *Data Manipulation,* describes the process of exchanging, moving, sorting, and transforming the data in Hive.

Chapter 6, *Data Aggregation and Sampling,* explains how to do aggregation and sample using aggregation functions, analytic functions, windowing, and sample clauses.

Chapter 7, *Performance Considerations,* introduces the best practices of performance considerations in the aspects of design, file format, compression, storage, query, and job.

Chapter 8, *Extensibility Considerations,* describes how to extend Hive by creating user-defined functions, streaming, serializers, and deserializers.

Chapter 9, *Security Considerations,* introduces the area of Hive security in terms of authentication, authorization, and encryption.

Chapter 10, *Working with Other Tools,* discusses how Hive works with other big data tools. It also reviews the key milestones of Hive releases.

What you need for this book

You will need to install both Hadoop and Hive to run the examples in this book. The scripts in this book were written and tested with Cloudera Distributed Hadoop (CDH) v5.3 (contains Hive v0.13.x and Hadoop v2.5.0), Hortonworks Data Platform (HDP) v2.2 (contains Hive v0.14.0 and Hadoop v2.6.0), and Apache Hive 1.0.0 (with Hadoop 1.2.1) in pseudo-distributed mode. However, the majority of the scripts will also run on the previous versions of Hadoop and Hive. The following are the other software applications you may need for a better understanding of the Hive-related tools mentioned in the book. These tools are also available in the CDH or HDP packages.

- Hue 2.2.0 and above
- HBase 0.98.4
- Oozie 4.0.0 and above
- Zookeeper 3.4.5
- Tez 0.6.0

Who this book is for

If you are a data analyst, developer, and user who wants to use Hive to explore and analyze data in Hadoop, this is the book for you. Whether you are new to big data or an expert, you will be able to master both the basic and the advanced features of Hive. Since Hive is an SQL-like language, some previous experience with the SQL language and database is useful to have a better understanding of this book.

Conventions

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

Code words in text, database table names, folder names, filenames, file extensions, pathnames, dummy URLs, user input, and Twitter handles are shown as follows: "Aggregate function can be used with other aggregate functions in the same `select` statement."

A block of code is set as follows:

```
<property>
  <name>javax.jdo.option.ConnectionURL</name>
  <value>jdbc:mysql://myhost:3306/hive?createDatabase
IfNotExist=true</value>
  <description>JDBC connect string for a JDBC metastore</description>
</property>
```

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

```
customAuthenticator.java
package com.packtpub.hive.essentials.hiveudf;

import java.util.Hashtable;
import javax.security.sasl.AuthenticationException;
import org.apache.hive.service.auth.PasswdAuthenticationProvider;
```

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

```
bash-4.1$ hdfs dfs –mkdir /tmp
```

New terms and **important words** are shown in bold. Words that you see on the screen, for example, in menus or dialog boxes, appear in the text like this: "Click on the **OK** button and restart Oracle SQL Developer."

Note

Warnings or important notes appear in a box like this.

Tip

Tips and tricks appear like this.

Reader feedback

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

To send us general feedback, simply e-mail <feedback@packtpub.com>, and mention the book's title in 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 at 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 from your account at http://www.packtpub.com for all the Packt Publishing books you have purchased. 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 could 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 to our website or added to any list of existing errata under the Errata section of that title.

To view the previously submitted errata, go to https://www.packtpub.com/books/content/support and enter the name of the book in the search field. The required information will appear under the **Errata** section.

Piracy

Piracy of copyrighted 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> 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

If you have a problem with any aspect of this book, you can contact us at <questions@packtpub.com>, and we will do our best to address the problem.

Chapter 1. Overview of Big Data and Hive

This chapter is an overview of big data and Hive, especially in the Hadoop ecosystem. It briefly introduces the evolution of big data so that readers know where they are in the journey of big data and find their preferred areas in future learning. This chapter also covers how Hive has become one of the leading tools in big data warehousing and why Hive is still competitive.

In this chapter, we will cover the following topics:

- A short history from database and data warehouse to big data
- Introducing big data
- Relational and NoSQL databases versus Hadoop
- Batch, real-time, and stream processing
- Hadoop ecosystem overview
- Hive overview

A short history

In the 1960s, when computers became a more cost-effective option for businesses, people started to use databases to manage data. Later on, in the 1970s, relational databases became more popular to business needs since they connected physical data to the logical business easily and closely. In the next decade, around the 1980s, **Structured Query Language** (**SQL**) became the standard query language for databases. The effectiveness and simplicity of SQL motivated lots of people to use databases and brought databases closer to a wide range of users and developers. Soon, it was observed that people used databases for data application and management and this continued for a long period of time.

Once plenty of data was collected, people started to think about how to deal with the old data. Then, the term data warehousing came up in the 1990s. From that time onwards, people started to discuss how to evaluate the current performance by reviewing the historical data. Various data models and tools were created at that time for helping enterprises to effectively manage, transform, and analyze the historical data. Traditional relational databases also evolved to provide more advanced aggregation and analyzed functions as well as optimizations for data warehousing. The leading query language was still SQL, but it was more intuitive and powerful as compared to the previous versions. The data was still well structured and the model was normalized. As we entered the 2000s, the Internet gradually became the topmost industry for the creation of the majority of data in terms of variety and volume. Newer technologies, such as social media analytics, web mining, and data visualizations, helped lots of businesses and companies deal with massive amounts of data for a better understanding of their customers, products, competition, as well as markets. The data volume grew and the data format changed faster than ever before, which forced people to search for new solutions, especially from the academic and open source areas. As a result, big data became a hot topic and a challenging field for many researchers and companies.

However, in every challenge there lies great opportunity. Hadoop was one of the open source projects earning wide attention due to its open source license and active communities. This was one of the few times that an open source project led to the changes in technology trends before any commercial software products. Soon after, the NoSQL database and real-time and stream computing, as followers, quickly became important components for big data ecosystems. Armed with these big data technologies, companies were able to review the past, evaluate the current, and also predict the future. Around the 2010s, time to market became the key factor for making business competitive and successful. When it comes to big data analysis, people could not wait to see the reports or results. A short delay could make a great difference when making important business decisions. Decision makers wanted to see the reports or results immediately within a few hours, minutes, or even possibly seconds in a few cases. Real-time analytical tools, such as Impala (http://www.cloudera.com/content/cloudera/en/products-and-services/cdh/impala.html), Presto (http://prestodb.io/), Storm (https://storm.apache.org/), and so on, make this possible in different ways.

Introducing big data

Big data is not simply a big volume of data. Here, the word "Big" refers to the big scope of data. A well-known saying in this domain is to describe big data with the help of three words starting with the letter V. They are volume, velocity, and variety. But the analytical and data science world has seen data varying in other dimensions in addition to the fundament 3 Vs of big data such as veracity, variability, volatility, visualization, and value. The different Vs mentioned so far are explained as follows:

- **Volume**: This refers to the amount of data generated in seconds. 90 percent of the world's data today has been created in the last two years. Since that time, the data in the world doubles every two years. Such big volumes of data is mainly generated by machines, networks, social media, and sensors, including structured, semi-structured, and unstructured data.
- **Velocity**: This refers to the speed in which the data is generated, stored, analyzed, and moved around. With the availability of Internet-connected devices, wireless or wired, machines and sensors can pass on their data immediately as soon as it is created. This leads to real-time streaming and helps businesses to make valuable and fast decisions.
- **Variety**: This refers to the different data formats. Data used to be stored as text, dat, and csv from sources such as filesystems, spreadsheets, and databases. This type of data that resides in a fixed field within a record or file is called structured data. Nowadays, data is not always in the traditional format. The newer semi-structured or unstructured forms of data can be generated using various methods such as e-mails, photos, audio, video, PDFs, SMSes, or even something we have no idea about. These varieties of data formats create problems for storing and analyzing data. This is one of the major challenges we need to overcome in the big data domain.
- **Veracity**: This refers to the quality of data, such as trustworthiness, biases, noise, and abnormality in data. Corrupt data is quite normal. It could originate due to a number of reasons, such as typos, missing or uncommon abbreviation, data reprocessing, system failures, and so on. However, ignoring this malicious data could lead to inaccurate data analysis and eventually a wrong decision. Therefore, making sure the data is correct in terms of data audition and correction is very important for big data analysis.
- **Variability**: This refers to the changing of data. It means that the same data could have different meanings in different contexts. This is particularly important when carrying out sentiment analysis. The analysis algorithms are able to understand the context and discover the exact meaning and values of data in that context.
- **Volatility**: This refers to how long the data is valid and stored. This is particularly important for real-time analysis. It requires a target scope of data to be determined so that analysts can focus on particular questions and gain good performance out of the analysis.
- **Visualization**: This refers to the way of making data well understood. Visualization does not mean ordinary graphs or pie charts. It makes vast amounts of data

comprehensible in a multidimensional view that is easy to understand. Visualization is an innovative way to show changes in data. It requires lots of interaction, conversations, and joint efforts between big data analysts and business domain experts to make the visualization meaningful.

- **Value**: This refers to the knowledge gained from data analysis on big data. The value of big data is how organizations turn themselves into big data-driven companies and use the insight from big data analysis for their decision making.

In summary, big data is not just about lots of data, it is a practice to discover new insight from existing data and guide the analysis for future data. A big-data-driven business will be more agile and competitive to overcome challenges and win competitions.

Relational and NoSQL database versus Hadoop

Let's compare different data solutions with the ways of traveling. You will be surprised to find that they have many similarities. When people travel, they either take cars or airplanes depending on the travel distance and cost. For example, when you travel to Vancouver from Toronto, an airplane is always the first choice in terms of the travel time versus cost. When you travel to Niagara Falls from Toronto, a car is always a good choice. When you travel to Montreal from Toronto, some people may prefer taking a car to an airplane. The distance and cost here is like the big data volume and investment. The traditional relational database is like the car in this example. The Hadoop big data tool is like the airplane in this example. When you deal with a small amount of data (short distance), a relational database (like the car) is always the best choice since it is more fast and agile to deal with a small or moderate size of data. When you deal with a big amount of data (long distance), Hadoop (like the airplane) is the best choice since it is more linear, fast, and stable to deal with the big size of data. On the contrary, you can drive from Toronto to Vancouver, but it takes too much time. You can also take an airplane from Toronto to Niagara, but it could take more time and cost way more than if you travel by a car. In addition, you may have a choice to either take a ship or a train. This is like a NoSQL database, which offers characters from both a relational database and Hadoop in terms of good performance and various data format support for big data.

Batch, real-time, and stream processing

Batch processing is used to process data in batches and it reads data input, processes it, and writes it to the output. Apache Hadoop is the most well-known and popular open source implementation of batch processing and a distributed system using the MapReduce paradigm. The data is stored in a shared and distributed filesystem called **Hadoop Distributed File System** (**HDFS**), divided into splits, which are the logical data divisions for MapReduce processing. To process these splits using the MapReduce paradigm, the map task reads the splits and passes all of its key/value pairs to a map function and writes the results to intermediate files. After the map phase is completed, the reducer reads intermediate files and passes it to the reduce function. Finally, the reduce task writes results to the final output files. The advantages of the MapReduce model include making distributed programming easier, near-linear speed up, good scalability, as well as fault tolerance. The disadvantage of this batch processing model is being unable to execute recursive or iterative jobs. In addition, the obvious batch behavior is that all inputs must be ready by map before the reduce job starts, which makes MapReduce unsuitable for online and stream processing use cases.

Real-time processing is to process data and get the result almost immediately. This concept in the area of real-time ad hoc queries over big data was first implemented in Dremel by Google. It uses a novel columnar storage format for nested structures with fast index and scalable aggregation algorithms for computing query results in parallel instead of batch sequences. These two techniques are the major characters for real-time processing and are used by similar implementations, such as Cloudera Impala, Facebook Presto, Apache Drill, and Hive on Tez powered by Stinger whose effort is to make a 100x performance improvement over Apache Hive. On the other hand, in-memory computing no doubt offers other solutions for real-time processing. In-memory computing offers very high bandwidth, which is more than 10 gigabytes/second, compared to hard disks' 200 megabytes/second. Also, the latency is comparatively lower, nanoseconds versus milliseconds, compared to hard disks. With the price of RAM going lower and lower each day, in-memory computing is more affordable as real-time solutions, such as Apache Spark, which is a popular open source implementation of in-memory computing. Spark can be easily integrated with Hadoop and the resilient distributed dataset can be generated from data sources such as HDFS and HBase for efficient caching.

Stream processing is to continuously process and act on the live stream data to get a result. In stream processing, there are two popular frameworks: Storm (https://storm.apache.org/) from Twitter and S4 (http://incubator.apache.org/s4/) from Yahoo!. Both the frameworks run on the **Java Virtual Machine** (**JVM**) and both process keyed streams. In terms of the programming model, S4 is a program defined as a graph of **Processing Elements** (**PE**), small subprograms, and S4 instantiates a PE per key. In short, Storm gives you the basic tools to build a framework, while S4 gives you a well-defined framework.

Overview of the Hadoop ecosystem

Hadoop was first released by Apache in 2011 as version 1.0.0. It only contained HDFS and MapReduce. Hadoop was designed as both a computing (MapReduce) and storage (HDFS) platform from the very beginning. With the increasing need for big data analysis, Hadoop attracts lots of other software to resolve big data questions together and merges to a Hadoop-centric big data ecosystem. The following diagram gives a brief introduction to the Hadoop ecosystem and the core software or components in the ecosystems:
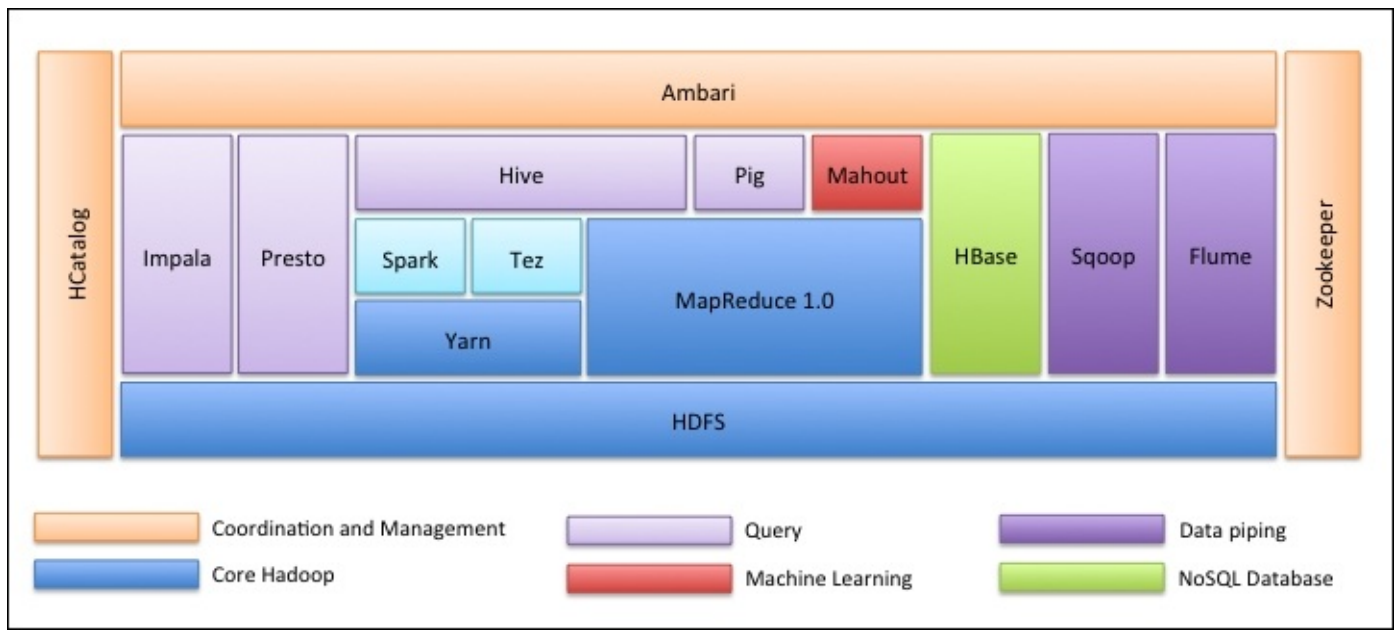
The Hadoop ecosystem

In the current Hadoop ecosystem, **HDFS** is still the major storage option. On top of it, snappy, RCFile, Parquet, and ORCFile could be used for storage optimization. Core Hadoop MapReduce released a version 2.0 called **Yarn** for better performance and scalability. **Spark** and **Tez** as solutions for real-time processing are able to run on the Yarn to work with Hadoop closely. **HBase** is a leading NoSQL database, especially when there is a NoSQL database request on the deployed Hadoop clusters. **Sqoop** is still one of the leading and matured tools for exchanging data between Hadoop and relational databases. **Flume** is a matured distributed and reliable log-collecting tool to move or collect data to HDFS. **Impala** and **Presto** query directly against the data on HDFS for better performance. However, Hortonworks focuses on Stringer initiatives to make Hive 100 times faster. In addition, Hive over Spark and Hive over Tez offer a choice for users to run Hive on other computing frameworks rather than MapReduce. As a result, Hive is playing more important roles in the ecosystem than ever.

Hive overview

Hive is a standard for SQL queries over petabytes of data in Hadoop. It provides SQL-like access for data in HDFS making Hadoop to be used like a warehouse structure. The **Hive Query Language** (**HQL**) has similar semantics and functions as standard SQL in the relational database so that experienced database analysts can easily get their hands on it. Hive's query language can run on different computing frameworks, such as MapReduce, Tez, and Spark for better performance.

Hive's data model provides a high-level, table-like structure on top of HDFS. It supports three data structures: tables, partitions, and buckets, where tables correspond to HDFS directories and can be divided into partitions, which in turn can be divided into buckets. Hive supports a majority of primitive data formats such as `TIMESTAMP`, `STRING`, `FLOAT`, `BOOLEAN`, `DECIMAL`, `DOUBLE`, `INT`, `SMALLINT`, `BIGINT`, and complex data types, such as `UNION`, `STRUCT`, `MAP`, and `ARRAY`.

The following diagram is the architecture seen inside the view of Hive in the Hadoop ecosystem. The Hive metadata store (or called metastore) can use either embedded, local, or remote databases. Hive servers are built on Apache Thrift Server technology. Since Hive has released 0.11, Hive Server 2 is available to handle multiple concurrent clients, which support Kerberos, LDAP, and custom pluggable authentication, providing better options for JDBC and ODBC clients, especially for metadata access.
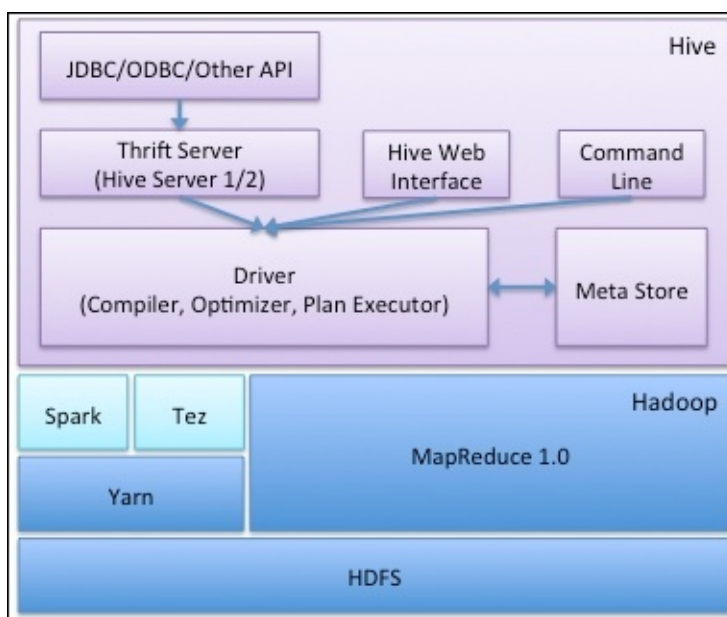
Hive architecture

Here are some highlights of Hive that we can keep in mind moving forward:

- Hive provides a simpler query model with less coding than MapReduce
- HQL and SQL have similar syntax
- Hive provides lots of functions that lead to easier analytics usage
- The response time is typically much faster than other types of queries on the same

type of huge datasets

- Hive supports running on different computing frameworks
- Hive supports ad hoc querying data on HDFS
- Hive supports user-defined functions, scripts, and a customized I/O format to extend its functionality
- Hive is scalable and extensible to various types of data and bigger datasets
- Matured JDBC and ODBC drivers allow many applications to pull Hive data for seamless reporting
- Hive allows users to read data in arbitrary formats, using SerDes and Input/Output formats
- Hive has a well-defined architecture for metadata management, authentication, and query optimizations
- There is a big community of practitioners and developers working on and using Hive

Summary

After going through this chapter, we are now able to understand why and when to use big data instead of a traditional relational database. We also understand the difference between batch processing, real-time processing, and stream processing. We got familiar with the Hadoop ecosystem, especially Hive. We have also gone back in time and brushed through the history of database and warehouse to big data along with some big data terms, the Hadoop ecosystem, Hive architecture, and the advantage of using Hive. In the next chapter, we will practice setting up Hive and all the tools needed to get started using Hive in the command line.

Chapter 2. Setting Up the Hive Environment

This chapter will introduce how to install and set up the Hive environment in the cluster and cloud. It also covers the usage of basic Hive commands and the Hive-integrated development environment.

In this chapter, we will cover the following topics:

- Installing Hive from Apache
- Installing Hive from vendor packages
- Starting Hive in the cloud
- Using the Hive command line and Beeline
- The Hive-integrated development environment

Installing Hive from Apache

To introduce the Hive installation, we use Hive version 1.0.0 as an example. The pre-installation requirements for this installation are as follows:

- JDK 1.7.0_51
- Hadoop 0.20.x, 0.23.x.y, 1.x.y, or 2.x.y
- Ubuntu 14.04/CentOS 6.2

  ### Note

  Since we focus on Hive in this book, the installation steps for Java and Hadoop are not provided here. For steps on installing them, please refer to https://www.java.com/en/download/help/download_options.xml and http://hadoop.apache.org/docs/current/hadoop-project-dist/hadoop-common/ClusterSetup.html.

The following steps describe how to install Hive from Apache through the Linux command line:

1. Download Hive from Apache Hive and unpack it:

   ```
   bash-4.1$ wget http://apache.mirror.rafal.ca/hive/hive-1.0.0/apache-hive-1.0.0-bin.tar.gz
   bash-4.1$ tar -zxvf apache-hive-1.0.0-bin.tar.gz
   ```

2. Add Hive to the system path by opening `/etc/profile` or `~/.bashrc` and add the following two rows:

   ```
   export HIVE_HOME=/home/hivebooks/apache-hive-1.0.0-bin
   export PATH=$PATH:$HIVE_HOME/bin:$HIVE_HOME/conf
   ```

3. Enable the settings immediately:

   ```
   bash-4.1$ source /etc/profile
   ```

4. Create the configuration files:

   ```
   bash-4.1$ cd apache-hive-1.0.0-bin/conf
   bash-4.1$ cp hive-default.xml.template hive-site.xml
   bash-4.1$ cp hive-env.sh.template hive-env.sh
   bash-4.1$ cp hive-exec-log4j.properties.template hive-exec-log4j.properties
   bash-4.1$ cp hive-log4j.properties.template hive-log4j.properties
   ```

5. Modify the configuration file at `$HIVE_HOME/conf/hive-env.sh`:

   ```
   #Set HADOOP_HOME to point to a specific Hadoop install directory
   export HADOOP_HOME=/home/hivebooks/hadoop-2.2.0
   #Hive Configuration Directory can be accessed at:
   export HIVE_CONF_DIR=/home/hivebooks/apache-hive-1.0.0-bin/conf
   ```

6. Modify the configuration file at `$HIVE_HOME/conf/hive-site.xml`. There are some important parameters that need special attention:

- hive.metastore.warehourse.dir: This is the path for Hive warehouse storage. By default it is /user/hive/warehouse.
- hive.exec.scratchdir: This is the temporary data file path. By default it is /tmp/hive-${user.name}.

By default, Hive uses the Derby (http://db.apache.org/derby/) database as the metadata store. Hive can also use other databases, such as PostgreSQL (http://www.postgresql.org/) or MySQL (http://www.mysql.com/) as the metadata store. To configure Hive to use other databases, the following parameters should be configured:

```
javax.jdo.option.ConnectionURL          // the database URL
javax.jdo.option.ConnectionDriverName   // the JDBC driver name
javax.jdo.option.ConnectionUserName     // database username
javax.jdo.option.ConnectionPassword     // database password
```

The following is an example setting using MySQL as the metastore database:

```
<property>
  <name>javax.jdo.option.ConnectionURL</name>
  <value>jdbc:mysql://myhost:3306/hive?createDatabase
IfNotExist=true</value>
  <description>JDBC connect string for a JDBC metastore</description>
</property>
<property>
  <name>javax.jdo.option.ConnectionDriverName</name>
  <value>com.mysql.jdbc.Driver</value>
  <description>Driver class name for a JDBC metastore</description>
</property>
<property>
  <name>javax.jdo.option.ConnectionUserName</name>
  <value>hive</value>
  <description>username to use against metastore database</description>
</property>
<property>
  <name>javax.jdo.option.ConnectionPassword</name>
  <value>hive</value>
  <description>password to use against metastore database</description>
</property>
```

Make sure the MySQL JDBC driver is available at $HIVE_HOME/lib.

Note

The differences between an embed Derby database and an external database is that an external database offers a shared service so that users can share the metadata of Hive. However, an embed database is only visible to the local users.

Create folders and grant proper write permissions to the user group in the HDFS folder:

```
bash-4.1$ hdfs dfs –mkdir /tmp
bash-4.1$ hdfs dfs –mkdir /user/hive/warehouse
bash-4.1$ hdfs dfs -chmod g+w /tmp
bash-4.1$ hdfs dfs -chmod g+w /user/hive/warehouse
```

That's all about Apache Hive installation. In one of the Hive nodes installed, type `hive` to enter the Hive command-line environment (`hive>`), which verifies Hive is successfully installed.

Installing Hive from vendor packages

Right now, many companies, such as Cloudera, MapR, IBM, and Hortonworks, have packaged Hadoop into more easily manageable distributions. Each company takes a slightly different strategy, but the consensus for all of these packages is to make Hadoop easier to use for enterprise. For example, we can easily install Hive from **Cloudera Distributed Hadoop** (**CDH**), which can be downloaded from http://www.cloudera.com/content/cloudera/en/downloads/cdh.html.

Once CDH is installed to have the Hadoop environment ready, we can add Hive to the Hadoop cluster by following a few steps:

1. Log in to the Cloudera manager and click on the dropdown button after the cluster name to choose **Add a Service**.
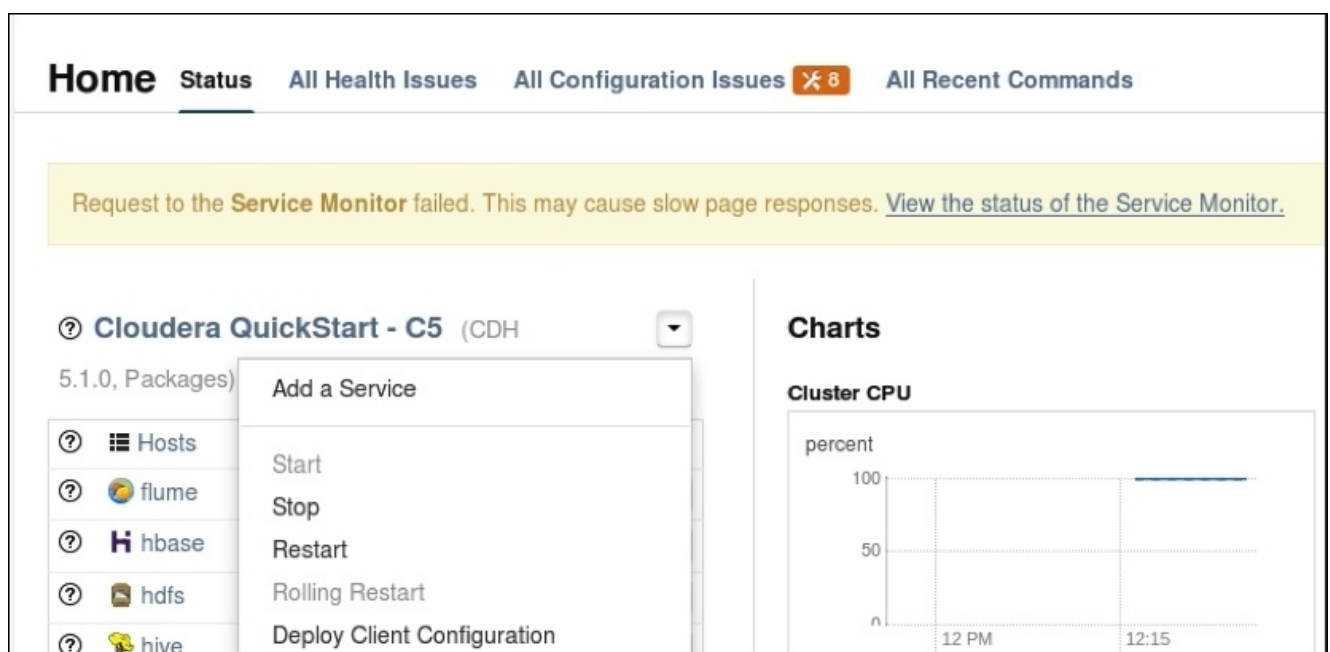
Cloudera manager main page

2. In the first **Add Service Wizard** page, choose **Hive** to install.

3. In the second **Add Service Wizard** page, set the dependencies for the service. **Sentry** is the authorization policy service for Hive.

4. In the third **Add Service Wizard** page, choose the proper hosts for HiveServer2, Hive Metastore Server, WebHCat Server, and Gateway.

5. In the fourth **Add Service Wizard** page, configure Hive Metastore Server database connections.

Add Service Wizard

Database Setup

Configure and test database connections. Create the databases first according to the **Installing and Configuring an External Database** section of the Installation Guide ⊘ .

Hive ✓ **Successful**

Database Host Name: *	**Database Type:**	**Database Name :** *	**Username:** *	**Password:**
quickstart.cloudera	MySQL	metastore	root	cloudera

☑ **Show Password**

Test Connection

Notes:

- The value in the **Database Host Name** field must match the value you used for the host name when creating the database. Learn more ⊘
- If the database is not running on its default port, specify the port number using **host:port** in the **Database Host Name** field.
- It is highly recommended that each database is on the same host as the corresponding role instance.

6. In the last page of **Add Service Wizard**, review the changes on the Hive warehouse directory and metastore server port number. Keep the default values and click on the **Continue** button to start installing the Hive service. Once it is complete, close the wizard to finish the Hive installation.

Note

Hive can also be installed along with other services when we first install CDH in the cluster or we can directly import the vendors' quick-start Hadoop virtual machine image.

Starting Hive in the cloud

Right now, Amazon EMR, Cloudera Director, and Microsoft Azure HDInsight Service are some of the major vendors offering matured Hadoop and Hive services in the cloud. Using the cloud version of Hive is very convenient. It almost requests no installation and setup.

Amazon EMR (http://aws.amazon.com/elasticmapreduce/) is the earliest Hadoop service in the cloud. However, it is not a pure open sourced version of Hadoop, but is customized to run only on AWS cloud. Cloudera is one of the first few players that offered open source Hadoop solutions to the enterprise. Since the middle of October 2014, Cloudera has delivered Cloudera Director (http://www.cloudera.com/content/cloudera/en/products-and-services/director.html), which opens up Hadoop deployments in the cloud through a simple, self-service interface, and is fully supported on Amazon Web Services. Windows Azure HDInsight Service (http://azure.microsoft.com/en-us/documentation/services/hdinsight/) is a service that deploys and provisions Apache Hadoop clusters in the Azure cloud. Although Hadoop was first built on Linux, Hortonworks and Microsoft have partnered to bring the benefits of Apache Hadoop to the Windows Azure cloud.

The consensus among all the vendors here is to allow the enterprise to provision highly available Hadoop clusters powered with flexibility, security, management, and governance functionalities with a very simple user interface.

Using the Hive command line and Beeline

Hive first started with HiveServer1. However, this version of the Hive server was not very stable. It sometimes suspended or blocked clients' connection quietly. Since version 11, Hive includes a new Hive server called HiveSever2 as an addition to HiveServer1. HiveServer2 is an enhanced Hive server designed for multiclient concurrency and improved authentication. HiveServer2 also supports Beeline as the alternative command-line interface. HiveServer1 is deprecated and removed from Hive since version 1.0.0.

The primary difference between the two Hive servers is how the clients connect to Hive. Hive CLI is an Apache Thrift-based client, and Beeline is a JDBC client based on SQLLine (http://sqlline.sourceforge.net/) CLI. The Hive CLI directly connects to the Hive drivers and requires installing Hive on the same machine as the client. However, Beeline connects to HiveServer2 through JDBC connections and does not require the installation of Hive libraries on the same machine as the client. That means we can run Beeline remotely from outside of the Hadoop cluster.

The following table is the commonly used commands for both Beeline and Hive CLI. For more usage of HiveServer2 and Beeline, refer to https://cwiki.apache.org/confluence/display/Hive/HiveServer2+Clients.

Purpose	HiveServer2 Beeline	HiveServer1 CLI
Server connection	`beeline –u  <jdbcurl> -n <username> -p <password>`	`hive -h <hostname> -p <port>`
Help	`beeline -h or beeline --help`	`hive -H`
Run query	`beeline -e <query in quotes>` `beeline -f <query file name>`	`hive -e <query in quotes>` `hive -f <query file name>`
Define variable	`beeline --hivevar key=value.` This is available after Hive 0.13.0.	`hive --hivevar key=value`

The following is the command-line syntax in Beeline or Hive CLI:

Purpose	HiveServer2 Beeline	HiveServer1 CLI
Enter mode	`beeline`	`hive`
Connect	`!connect <jdbcurl>`	`n/a`
List tables	`!table`	`show tables;`
List columns	`!column <table_name>`	`desc <table_name>;`
Run query	`<HQL query>;`	`<HQL query>;`
Save result set	`!record <file_name>` `!record`	`N/A`

Run shell CMD	`!sh ls` This is available since Hive 0.14.0.	`!ls;`
Run dfs CMD	`dfs -ls`	`dfs -ls;`
Run file of SQL	`!run <file_name>`	`source <file_name>;`
Check Hive version	`!dbinfo`	`!hive --version;`
Quit mode	`!quit`	`quit;`

Note

For Beeline, `;` is not needed after the command that starts with `!`.

When running a query in Hive CLI, the MapReduce statistics information is shown in the console screen while processing, whereas Beeline does not.

Both Beeline and Hive CLI do not support running a pasted query with <tab> inside, because <tab> is used for autocomplete by default in the environment. Alternatively, running the query from files has no such issues.

Hive CLI shows the exact line and position of the Hive query or syntax errors when the query has multiple lines. However, Beeline processes the multiple-line query as a single line, so only the position is shown for query or syntax errors with the line number as 1 for all instances. For this aspect, Hive CLI is more convenient than Beeline for debugging the Hive query.

In both Hive CLI and Beeline, using the up and down arrow keys can retrieve up to 10,000 previous commands. The `!history` command can be used in Beeline to show all history.

Both Hive CLI and Beeline supports variable substitution; refer to https://cwiki.apache.org/confluence/display/Hive/LanguageManual+VariableSubstitution.

A list of Hive configuration settings and properties can be accessed and overwritten by the `set` keyword from the command-line environment. For more details, refer to the Apache Hive wiki at https://cwiki.apache.org/confluence/display/Hive/Configuration+Properties.

The Hive-integrated development environment

Besides the command-line interface, there are a few **integrated development environment** (**IDE**) tools available for Hive development. One of the best is Oracle SQL Developer, which leverages the powerful functionalities of Oracle IDE and is totally free to use. If we have to use Oracle along with Hive in a project, it is quite convenient to switch between them only from the same IDE.

Oracle SQL developer has supported Hive since version 4.0.3. Configuring it to work with Hive is quite straightforward. The following are a few steps to configure the IDE to connect to Hive:
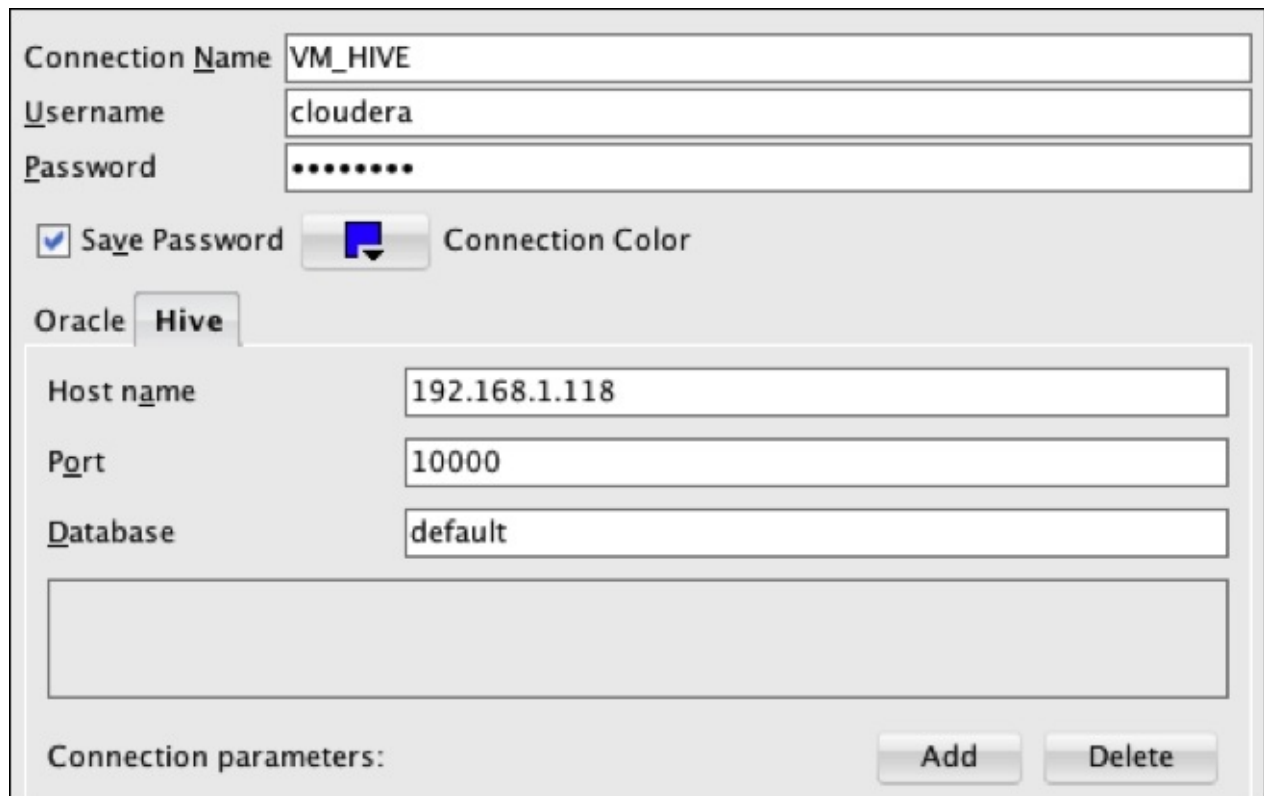
1. Download Hive JDBC drivers from the vendor website, such as Cloudera.
2. Unzip the JDBC version 4 driver to a local directory.
3. Start Oracle SQL Developer and navigate to **Preferences** | **Database** | **Third Party JDBC Drivers**.
4. Add all of the JAR files contained in the unzipped directory to the **Third-party JDBC Driver Path** setting as follows:
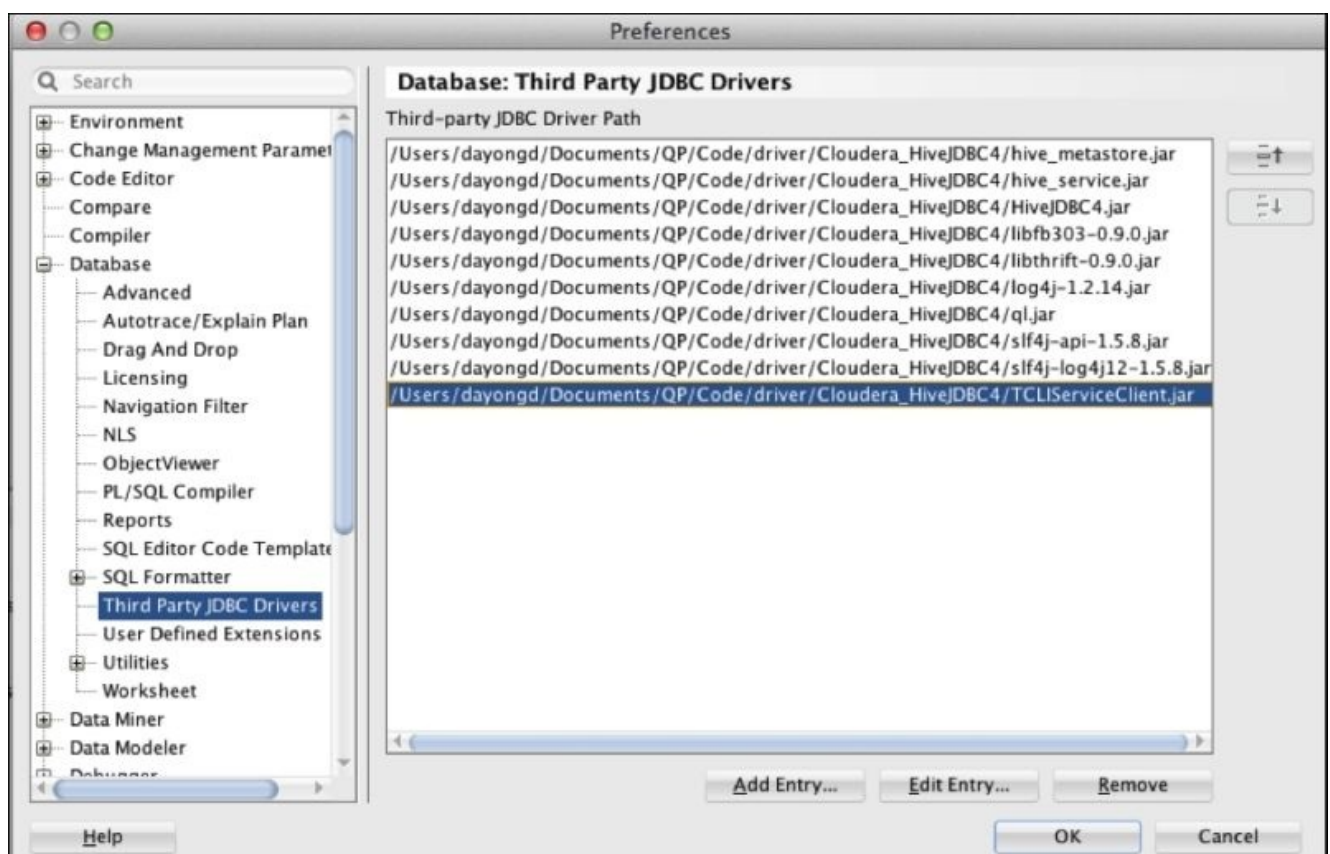
SQL developer configuration

5. Click on the **OK** button and restart Oracle SQL Developer.
6. Create new connections in the **Hive** tab giving a proper **Connection Name**,

Username, **Password**, **Host name** (Hive server hostname), **Port**, and **Database**. Then, click on the **Add** and **Connect** buttons to connect to Hive.

SQL developer connections

In Oracle SQL Developer, we can run all Hive interactive commands as well as Hive queries. We can also leverage the power of Oracle SQL Developer to browse and export data into a Hive table from the graphic user interface and wizard.

Besides Hive IDE, Hive also has its own built-in web interface, HiveWebInterface. However, it is not powerful and is not being used very often. Hue (http://gethue.com/) is another web interface for the Hadoop ecosystem, including Hive. It is a very powerful and user-friendly web user interface. More details about using Hue with Hive are introduced in Chapter 10, *Working with Other Tools*.

Summary

In this chapter, we introduced the setup of Hive in different environments with proper settings. We also looked into a few of the Hive interactive commands and queries in Hive CLI, Beeline, and IDEs. After going through this chapter, we should be able to set up our own Hive environment locally and use Hive from CLI or IDE tools.

In the next chapter, we will dive into the details of Hive data definition languages.

Chapter 3. Data Definition and Description

This chapter introduces the basic data types, data definition language, and schema in Hive to describe data. It also covers the best practices to describe data correctly and effectively by using internal or external tables, partitions, buckets, and views.

In this chapter, we will cover the following topics:

- Hive primitive and complex data types
- Data type conversions
- Hive tables
- Hive partitions
- Hive buckets
- Hive views

Understanding Hive data types

Hive data types are categorized into two types: primitive and complex data types. String and integer are the most useful primitive types, which are supported by most Hive functions.

Tip

Downloading the example code

You can download the example code files from your account at http://www.packtpub.com for all the Packt Publishing books you have purchased. 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

The details of primitive types are as follows:

Primitive data type	Description	Example
TINYINT	It has 1 byte from -128 to 127. The postfix is Y. It is used as a small range of numbers.	10Y
SMALLINT	It has 2 bytes from -32,768 to 32,767. The postfix is S. It is used as a regular descriptive number.	10S
INT	It has 4 bytes from -2,147,483,648 to 2,147,483,647.	10
BIGINT	It has 8 bytes from -9,223,372,036,854,775,808 to 9,223,372,036,854,775,807. The postfix is L.	100L
FLOAT	This is a 4-byte single precision floating point number from $1.40129846432481707e^{-45}$ to $3.40282346638528860e^{+38}$ (positive or negative). Scientific notation is not yet supported. It stores very close approximations of numeric values.	1.2345679
DOUBLE	This is an 8-byte double precision floating point number from $4.94065645841246544e^{-324d}$ to $1.79769313486231570e^{+308d}$ (positive or negative). Scientific notation is not yet supported. It stores very close approximations of numeric values.	1.2345678901234567
DECIMAL	This was introduced in Hive 0.11.0 with a hardcode precision of 38 digits. Hive 0.13.0 introduced user definable precision and scale. It is around 10^{39} - 1 to 1 - 10^{38}. Decimal data types store exact representations of numeric values. The default definition of this type is decimal(10,0).	DECIMAL (3,2) for 3.14
BINARY	This was introduced in Hive 0.8.0 and only supports CAST to STRING and vice versa.	1011
BOOLEAN	This is a TRUE or FALSE value.	TRUE
STRING	This includes characters expressed with either single quotes (') or double quotes ("). Hive uses C-style escaping within the strings. The max size is around 2G.	'Books' or "Books"

CHAR	This is available starting with Hive 0.13.0. Most UDF will work for this type after Hive 0.14.0. The maximum length is fixed at 255.	'US' or "US"
VARCHAR	This is available starting with Hive 0.12.0. Most UDF will work for this type after Hive 0.14.0. The maximum length is fixed at 65355. If a string value being converted/assigned to a varchar value exceeds the length specified, the string is silently truncated.	'Books' or "Books"
DATE	This describes a specific year, month, and day in the format of YYYY-MM-DD. It is available since Hive 0.12.0. The range of date is from 0000-01-01 to 9999-12-31.	'2013-01-01'
TIMESTAMP	This describes a specific year, month, day, hours, minutes, seconds, and milliseconds in the format of YYYY-MM-DD HH:MM:SS[.fff…]. It is available since Hive 0.8.0.	'2013-01-01 12:00:01.345'

Hive has three main complex types: `ARRAY`, `MAP`, and `STRUCT`. These data types are built on top of the primitive data types. `ARRAY` and `MAP` are similar to that in Java. `STRUCT` is a record type, which may contain a set of any type of fields. Complex types allow the nesting of types. The details of complex types are as follows:

Complex data type	Description	Example
ARRAY	This is a list of items of the same type, such as (val1, val2, and so on). You can access the value using `array_name[index]`, for example, `fruit[0]='apple'`.	['apple','orange','mango']
MAP	This is a set of key-value pairs, such as (key1, val1, key2, val2, and so on). You can access the value using `map_name[key]`, for example, `fruit[1]="apple"`.	{1: "apple",2: "orange"}
STRUCT	This is a user-defined structure of any type of fields, such as {val1, val2, val3, and so on}. By default, `STRUCT` field names will be col1, col2, and so on. You can access the value using `structs_name.column_name`, for example, `fruit.col1=1`.	{1, "apple"}
NAMED STRUCT	This is a user-defined structure of any number of typed fields, such as (name1, val1, name2, val2, and so on). You can access the value using `structs_name.column_name`, for example, `fruit.apple="gala"`.	{"apple":"gala","weight kg":1}
UNION	This is a structure that has exactly any one of the specified data types. It is available since Hive 0.7.0. It is not commonly used.	{2:["apple","orange"]}

Note

For `MAP`, the type of keys and values are unified. However, `STRUCT` is more flexible. `STRUCT` is more like a table whereas `MAP` is more like an `ARRAY` with a customized index.

The following is a short practice for all the commonly used Hive types. The details of the `CREATE`, `LOAD`, and `SELECT` statements will be described later. Let's take a look at the process:

1. Prepare the data as follows:

    ```
    -bash-4.1$ vi employee.txt
    ```

```
Michael|Montreal,Toronto|Male,30|DB:80|Product:Developer^DLead
Will|Montreal|Male,35|Perl:85|Product:Lead,Test:Lead
Shelley|New York|Female,27|Python:80|Test:Lead,COE:Architect
Lucy|Vancouver|Female,57|Sales:89,HR:94|Sales:Lead
```

2. Log in to Beeline with the proper HiveServer2 hostname, port number, database name, username, and password:

```
-bash-4.1$ beeline
beeline> !connect jdbc:hive2://localhost:10000/default
scan complete in 20ms Connecting to
jdbc:hive2://localhost:10000/default
Enter username for jdbc:hive2://localhost:10000/default:dayongd Enter
password for jdbc:hive2://localhost:10000/default:
```

3. Create a table using ARRAY, MAP, and STRUCT composite data types:

```
jdbc:hive2://> CREATE TABLE employee
. . . . . . .> (
. . . . . . .>   name string,
. . . . . . .>   work_place ARRAY<string>,
. . . . . . .>   sex_age STRUCT<sex:string,age:int>,
. . . . . . .>   skills_score MAP<string,int>,
. . . . . . .>   depart_title MAP<string,ARRAY<string>>
. . . . . . .> )
. . . . . . .> ROW FORMAT DELIMITED
. . . . . . .> FIELDS TERMINATED BY '|'
. . . . . . .> COLLECTION ITEMS TERMINATED BY ','
. . . . . . .> MAP KEYS TERMINATED BY ':';
No rows affected (0.149 seconds)
```

4. Verify the table's creation:

```
jdbc:hive2://>!table employee
+---------+------------+------------+--------------+---------+
|TABLE_CAT|TABLE_SCHEMA| TABLE_NAME |  TABLE_TYPE  | REMARKS |
+---------+------------+------------+--------------+---------+
|         |default     | employee   | MANAGED_TABLE|         |
+---------+------------+------------+--------------+---------+
```

```
jdbc:hive2://>!column employee
+--------------+------------+--------------+--------------+
| TABLE_SCHEM | TABLE_NAME  | COLUMN_NAME  |  TYPE_NAME    |
+--------------+------------+--------------+--------------+
| default      | employee   | name         | STRING        |
| default      | employee   | work_place   | array<string> |
| default      | employee   | sex_age      |
struct<sex:string,age:int>|
| default      | employee   | skills_score | map<string,int>|
| default      | employee   | depart_title | map<string,array<string>>
|
+--------------+------------+--------------+--------------+
```

5. Load data into the table:

```
jdbc:hive2://>LOAD DATA LOCAL INPATH '/home/hadoop/employee.txt'
. . . . . . .>OVERWRITE INTO TABLE employee;
```

```
No rows affected (1.023 seconds)
```

6. Query all the rows in the table:

```
jdbc:hive2://> SELECT * FROM employee;
+-------+------------------+-----------+----------------+----------
------------------+
| name  |    work_place    |  sex_age  |  skills_score  |
depart_title              |
+-------+------------------+-----------+----------------+----------
------------------+
|Michael|[Montreal, Toronto]|[Male, 30] |{DB=80}        |{Product=
[Developer, Lead]}    |
|Will   |[Montreal]        |[Male, 35] |{Perl=85}      |{Test=
[Lead], Product=[Lead]} |
|Shelley|[New York]        |[Female, 27]|{Python=80}   |{Test=
[Lead], COE=[Architect]}|
|Lucy   |[Vancouver]       |[Female, 57]|{Sales=89, HR=94}|{Sales=
[Lead]}                  |
+-------+------------------+-----------+----------------+----------
------------------+
4 rows selected (0.677 seconds)
```

7. Query the whole array and each array column in the table:

```
jdbc:hive2://> SELECT work_place FROM employee;
+----------------------+
|      work_place      |
+----------------------+
| [Montreal, Toronto]  |
| [Montreal]           |
| [New York]           |
| [Vancouver]          |
+----------------------+
4 rows selected (27.231 seconds)

jdbc:hive2://> SELECT work_place[0] AS col_1,
. . . . . . .> work_place[1] AS col_2, work_place[2] AS col_3
. . . . . . .> FROM employee;
+------------+----------+--------+
|   col_1    |  col_2   | col_3  |
+------------+----------+--------+
| Montreal   | Toronto  |        |
| Montreal   |          |        |
| New York   |          |        |
| Vancouver  |          |        |
+------------+----------+--------+
4 rows selected (24.689 seconds)
```

8. Query the whole struct and each struct column in the table:

```
jdbc:hive2://> SELECT sex_age FROM employee;
+---------------+
|    sex_age    |
+---------------+
| [Male, 30]    |
| [Male, 35]    |
```

```
| [Female, 27]  |
| [Female, 57]  |
+---------------+
4 rows selected (28.91 seconds)


jdbc:hive2://> SELECT sex_age.sex, sex_age.age FROM employee;
+---------+------+
|   sex   | age  |
+---------+------+
| Male    | 30   |
| Male    | 35   |
| Female  | 27   |
| Female  | 57   |
+---------+------+
4 rows selected (26.663 seconds)
```

9. Query the whole map and each map column in the table:

```
jdbc:hive2://> SELECT skills_score FROM employee;
+---------------------+
|     skills_score    |
+---------------------+
| {DB=80}             |
| {Perl=85}           |
| {Python=80}         |
| {Sales=89, HR=94}   |
+---------------------+
4 rows selected (32.659 seconds)


jdbc:hive2://> SELECT name, skills_score['DB'] AS DB,
. . . . . . .> skills_score['Perl'] AS Perl,
. . . . . . .> skills_score['Python'] AS Python,
. . . . . . .> skills_score['Sales'] as Sales,
. . . . . . .> skills_score['HR'] as HR
. . . . . . .> FROM employee;
+----------+-----+-------+---------+--------+-----+
|   name   | db  | perl  | python  | sales  | hr  |
+----------+-----+-------+---------+--------+-----+
| Michael  | 80  |       |         |        |     |
| Will     |     | 85    |         |        |     |
| Shelley  |     |       | 80      |        |     |
| Lucy     |     |       |         | 89     | 94  |
+----------+-----+-------+---------+--------+-----+
4 rows selected (24.669 seconds)
```

Note

Note that the column name shown in the result set for Hive is always in lowercase letters.

10. Query the composite type in the table:

```
jdbc:hive2://> SELECT depart_title FROM employee;
+---------------------------------+
|          depart_title           |
+---------------------------------+
| {Product=[Developer, Lead]}     |
```

```
| {Test=[Lead], Product=[Lead]}   |
| {Test=[Lead], COE=[Architect]}  |
| {Sales=[Lead]}                  |
+--------------------------------+
4 rows selected (30.583 seconds)

jdbc:hive2://> SELECT name,
. . . . . . .> depart_title['Product'] AS Product,
. . . . . . .> depart_title['Test'] AS Test,
. . . . . . .> depart_title['COE'] AS COE,
. . . . . . .> depart_title['Sales'] AS Sales
. . . . . . .> FROM employee;
+--------+-------------------+--------+-------------+------+
|  name  |      product      |  test  |     coe     |sales |
+--------+-------------------+--------+-------------+------+
| Michael| [Developer, Lead] |        |             |      |
| Will   | [Lead]            | [Lead] |             |      |
| Shelley|                   | [Lead] | [Architect] |      |
| Lucy   |                   |        |             |[Lead]|
+--------+-------------------+--------+-------------+------+
4 rows selected (26.641 seconds)

jdbc:hive2://> SELECT name,
. . . . . . .> depart_title['Product'][0] AS product_col0,
. . . . . . .> depart_title['Test'][0] AS test_col0
. . . . . . .> FROM employee;
+---------+--------------+-----------+
|  name   | product_col0 | test_col0 |
+---------+--------------+-----------+
| Michael | Developer    |           |
| Will    | Lead         | Lead      |
| Shelley |              | Lead      |
| Lucy    |              |           |
+---------+--------------+-----------+
4 rows selected (26.659 seconds)
```

Note

The default delimiters in Hive are as follows:

- **Row delimiter**: This can be used with *Ctrl + A* or ^A (Use \001 when creating the table)
- **Collection item delimiter**: This can be used with *Ctrl + B* or ^B (\002)
- **Map key delimiter**: This can be used with *Ctrl + C* or ^C (\003)

If the delimiter is overidden during the table creation, it only works when used in the flat structure. This is still a limitation in Hive described in Apache Jira Hive-365 (https://issues.apache.org/jira/browse/HIVE-365).

For nested types, for example, the depart_title column in the preceding tables, the level of nesting determines the delimiter. Using ARRAY of ARRAY as an example, the delimiters for the outer ARRAY are *Ctrl + B* (\002) characters, as expected, but for the inner ARRAY they are *Ctrl + C* (\003) characters, the next delimiter in the list. For our example of using MAP

of ARRAY, the MAP key delimiter is \003, and the ARRAY delimiter is *Ctrl + D* or ^D (\004).

Data type conversions

Similar to Java, Hive supports both implicit type conversion and explicit type conversion.

Primitive type conversion from a narrow to a wider type is known as implicit conversion. However, the reverse conversion is not allowed. All the integral numeric types, `FLOAT`, and `STRING` can be implicitly converted to `DOUBLE`, and `TINYINT`, `SMALLINT`, and `INT` can all be converted to `FLOAT`. `BOOLEAN` types cannot be converted to any other type. In the Apache Hive wiki, there is a data type cross table describing the allowed implicit conversion between every two types in Hive and this can be found at https://cwiki.apache.org/confluence/display/Hive/LanguageManual+Types.

Explicit type conversion is using the `CAST` function with the `CAST(value AS TYPE)` syntax. For example, `CAST('100' AS INT)` will convert the string `100` to the integer value 100. If the cast fails, such as `CAST('INT' AS INT)`, the function returns `NULL`. In addition, the `BINARY` type can only cast to `STRING`, then cast from `STRING` to other types, if needed.

Hive Data Definition Language

Hive Data Definition Language (**DDL**) is a subset of Hive SQL statements that describe the data structure in Hive by creating, deleting, or altering schema objects such as databases, tables, views, partitions, and buckets. Most Hive DDL statements start with the keywords `CREATE`, `DROP`, or `ALTER`. The syntax of Hive DDL is very similar to the DDL in SQL. The comments in Hive start from `--`.

Hive database

The database in Hive describes a collection of tables that are used for a similar purpose or belong to the same groups. If the database is not specified, the default database is used. Whenever a new database is created, Hive creates a directory for each database at `/user/hive/warehouse`, defined in `hive.metastore.warehouse.dir`. For example, the `myhivebook` database is located at `/user/hive/datawarehouse/myhivebook.db`. However, the default database doesn't have its own directory. The following is the core DDL for Hive databases:

- Create the database without checking whether the database already exists:

  `jdbc:hive2://> CREATE DATABASE myhivebook;`

- Create the database and check whether the database already exists:

  `jdbc:hive2://> CREATE DATABASE IF NOT EXISTS myhivebook;`

- Create the database with location, comments, and metadata information:

  ```
  jdbc:hive2://> CREATE DATABASE IF NOT EXISTS myhivebook
  . . . . . . .> COMMENT 'hive database demo'
  . . . . . . .> LOCATION '/hdfs/directory'
  . . . . . . .> WITH DBPROPERTIES ('creator'='dayongd','date'='2015-01-
  01');
  ```

- Show and describe the database with wildcards:

  ```
  jdbc:hive2://> SHOW DATABASES;
  +----------------+
  | database_name  |
  +----------------+
  | default        |
  +----------------+
  1 row selected (1.7 seconds)

  jdbc:hive2://> SHOW DATABASES LIKE 'my.*';
  jdbc:hive2://> DESCRIBE DATABASE default;
  +-------+--------------------+---------------------------+
  |db_name|      comment       |          location         |
  +-------+--------------------+---------------------------+
  |default|Default Hive database
  |hdfs://localhost:8020/user/hive/warehouse|
  +-------+--------------------+---------------------------+
  1 row selected (1.352 seconds)
  ```

- Use the database:

  `jdbc:hive2://> USE myhivebook;`

- Drop the empty database:

  `jdbc:hive2://> DROP DATABASE IF EXISTS myhivebook;`

  ## Note

Note that Hive keeps the database and the table in directory mode. In order to remove the parent directory, we need to remove the subdirectories first. By default, the database cannot be dropped if it is not empty, unless CASCADE is specified. CASCADE drops the tables in the database automatically before dropping the database.

- Drop the database with CASCADE:

```
jdbc:hive2://> DROP DATABASE IF EXISTS myhivebook CASCADE;
```

- Alter the database properties. The ALTER DATABASE statement can only apply to the table properties and roles (Hive 0.13.0 and later) on the table. The other metadata about the database cannot be changed:

```
jdbc:hive2://> ALTER DATABASE myhivebook
. . . . . . .> SET DBPROPERTIES ('edited-by' = 'Dayong');

jdbc:hive2://> ALTER DATABASE myhivebook
 SET OWNER user dayongd;
```

Note

SHOW and DESCRIBE

The SHOW and DESCRIBE keywords in Hive are used to show the definition information for most of the Hive objects, such as tables, partitions, and so on.

The SHOW statement supports a wide range of Hive objects, such as tables, tables' properties, table DDL, index, partitions, columns, functions, locks, roles, configurations, transactions, and compactions.

The DESCRIBE statement supports a small range of Hive objects, such as databases, tables, views, columns, and partitions. However, the DESCRIBE statement is able to provide more detailed information combined with the EXTENDED or FORMATTED keywords.

In this book, there is no single section to introduce SHOW and DESCRIBE, but we introduce their usage in line with other HQL through the remaining chapters.

Hive internal and external tables

The concept of a table in Hive is very similar to the table in the relational database. Each table associates with a directory configured in `${HIVE_HOME}/conf/hive-site.xml` in HDFS. By default, it is `/user/hive/warehouse` in HDFS. For example, `/user/hive/warehouse/employee` is created by Hive in HDFS for the `employee` table. All the data in the table will be kept in the directory. The Hive table is also referred to as internal or managed tables.

When there is data already in HDFS, an external Hive table can be created to describe the data. It is called `EXTERNAL` because the data in the external table is specified in the `LOCATION` properties instead of the default warehouse directory. When keeping data in the internal tables, Hive fully manages the life cycle of the table and data. This means the data is removed once the internal table is dropped. If the external table is dropped, the table metadata is deleted but the data is kept. Most of the time, an external table is preferred to avoid deleting data along with tables by mistake. The following are DDLs for Hive internal and external table examples:

- Show the database file's location and content of the `employee` internal table:

```
bash-4.1$ vi /home/hadoop/employee.txt
Michael|Montreal,Toronto|Male,30|DB:80|Product:Developer^DLead
Will|Montreal|Male,35|Perl:85|Product:Lead,Test:Lead
Shelley|New York|Female,27|Python:80|Test:Lead,COE:Architect
Lucy|Vancouver|Female,57|Sales:89,HR:94|Sales:Lead
```

- Create the internal table and load the data:

```
jdbc:hive2://> CREATE TABLE IF NOT EXISTS employee_internal
. . . . . . .> (
. . . . . . .>  name string,
. . . . . . .>  work_place ARRAY<string>,
. . . . . . .>  sex_age STRUCT<sex:string,age:int>,
. . . . . . .>  skills_score MAP<string,int>,
. . . . . . .>  depart_title MAP<STRING,ARRAY<STRING>>
. . . . . . .> )
. . . . . . .> COMMENT 'This is an internal table'
. . . . . . .> ROW FORMAT DELIMITED
. . . . . . .> FIELDS TERMINATED BY '|'
. . . . . . .> COLLECTION ITEMS TERMINATED BY ','
. . . . . . .> MAP KEYS TERMINATED BY ':'
. . . . . . .> STORED AS TEXTFILE;
No rows affected (0.149 seconds)

jdbc:hive2://> LOAD DATA LOCAL INPATH '/home/hadoop/employee.txt'
. . . . . . .> OVERWRITE INTO TABLE employee_internal;
```

- Create the external table and load the data:

```
jdbc:hive2://> CREATE EXTERNAL TABLE employee_external
. . . . . . .> (
. . . . . . .>  name string,
. . . . . . .>  work_place ARRAY<string>,
```

```
. . . . . . .>  sex_age STRUCT<sex:string,age:int>,
. . . . . . .>  skills_score MAP<string,int>,
. . . . . . .>  depart_title MAP<STRING,ARRAY<STRING>>
. . . . . . .> )
. . . . . . .> COMMENT 'This is an external table'
. . . . . . .> ROW FORMAT DELIMITED
. . . . . . .> FIELDS TERMINATED BY '|'
. . . . . . .> COLLECTION ITEMS TERMINATED BY ','
. . . . . . .> MAP KEYS TERMINATED BY ':'
. . . . . . .> STORED AS TEXTFILE
. . . . . . .> LOCATION '/user/dayongd/employee';
No rows affected (1.332 seconds)

jdbc:hive2://> LOAD DATA LOCAL INPATH '/home/hadoop/employee.txt'. . .
. . . .> OVERWRITE
 INTO TABLE employee_external;
```

Note

CREATE TABLE

The Hive table does not have constraints such as a database yet.

If the folder in the path does not exist in the `LOCATION` property, Hive will create that folder. If there is another folder inside the folder specified in the `LOCATION` property, Hive will NOT report errors when creating the table, but will report an error when querying the table.

A temporary table, which is automatically deleted at the end of the Hive session, is supported in Hive 0.14.0 by HIVE-7090 (https://issues.apache.org/jira/browse/HIVE-7090) through the `CREATE TEMPORARY TABLE` statement.

For the `STORE AS` property, it is set to `AS TEXTFILE` by default. Other file format values, such as `SEQUENCEFILE`, `RCFILE`, `ORC`, `AVRO` (since Hive 0.14.0), and `PARQUET` (since Hive 0.13.0) can also be specified.

- Create the table as select (CTAS):

```
jdbc:hive2://> CREATE TABLE ctas_employee
. . . . . . .> AS SELECT * FROM employee_external;
No rows affected (1.562 seconds)
```

Note

CTAS

`CTAS` copies the data as well as table definitions. The table created by `CTAS` is atomic; this means that other users do not see the table until all the query results are populated. `CTAS` has the following restrictions:

- The table created cannot be a partitioned table
- The table created cannot be an external table
- The table created cannot be a list bucketing table

A `CTAS` statement will trigger a map job for populating the data; even `SELECT *` itself

does not trigger any MapReduce job.

- CTAS with **Common Table Expression** (**CTE**) can be created as follows:

```
jdbc:hive2://> CREATE TABLE cte_employee AS
. . . . . . .> WITH r1 AS
. . . . . . .> (SELECT name FROM r2
. . . . . . .> WHERE name = 'Michael'),
. . . . . . .> r2 AS
. . . . . . .> (SELECT name FROM employee
. . . . . . .> WHERE sex_age.sex= 'Male'),
. . . . . . .> r3 AS
. . . . . . .> (SELECT name FROM employee
. . . . . . .> WHERE sex_age.sex= 'Female')
. . . . . . .> SELECT * FROM r1 UNION ALL select * FROM r3;
No rows affected (61.852 seconds)

jdbc:hive2://> SELECT * FROM cte_employee;
+---------------------------+
| cte_employee.name         |
+---------------------------+
| Michael                   |
| Shelley                   |
| Lucy                      |
+---------------------------+
3 rows selected (0.091 seconds)
```

Note

CTE

CTE is available since Hive 0.13.0. It is a temporary result set derived from a simple SELECT query specified in a WITH clause, followed by SELECT or INSERT keyword to operate this result set. The CTE is defined only within the execution scope of a single statement. One or more CTEs can be used in a nested or chained way with Hive keywords, such as the SELECT, INSERT, CREATE TABLE AS SELECT, or CREATE VIEW AS SELECT statements.

- Empty tables can be created in two ways as follows:

  1. Use CTAS as shown here:

     ```
     jdbc:hive2://> CREATE TABLE empty_ctas_employee AS
     . . . . . . .> SELECT * FROM employee_internal WHERE 1=2;
     No rows affected (213.356 seconds)
     ```

  2. Use LIKE as shown here:

     ```
     jdbc:hive2://> CREATE TABLE empty_like_employee
     . . . . . . .> LIKE employee_internal;
     No rows affected (0.115 seconds)
     ```

- Check the row counts for both tables:

```
jdbc:hive2://> SELECT COUNT(*) AS row_cnt
```

```
. . . . . . .> FROM empty_ctas_employee;
+----------+
| row_cnt  |
+----------+
| 0        |
+----------+
1 row selected (51.228 seconds)

jdbc:hive2://> SELECT COUNT(*) AS row_cnt
. . . . . . .> FROM empty_like_employee;
+----------+
| row_cnt  |
+----------+
| 0        |
+----------+
1 row selected (41.628 seconds)
```

Note

The LIKE way, which is faster, does not trigger a MapReduce job since it is metadata duplication only.

- The drop table's command removes the metadata completely and moves data to Trash or to the current directory if Trash is configured:

```
jdbc:hive2://> DROP TABLE IF EXISTS empty_ctas_employee;
No rows affected (0.283 seconds)

jdbc:hive2://> DROP TABLE IF EXISTS empty_like_employee;
No rows affected (0.202 seconds)
```

- The truncate table's command removes all the rows from a table that should be an internal table:

```
jdbc:hive2://> SELECT * FROM cte_employee;
+--------------------+
| cte_employee.name  |
+--------------------+
| Michael            |
| Shelley            |
| Lucy               |
+--------------------+
3 rows selected (0.158 seconds)

jdbc:hive2://> TRUNCATE TABLE cte_employee;
No rows affected (0.093 seconds)

--Table is empty after truncate
jdbc:hive2://> SELECT * FROM cte_employee;
+--------------------+
| cte_employee.name  |
+--------------------+
+--------------------+
No rows selected (0.059 seconds)
```

- Alter the table's statements to rename the table:

```
jdbc:hive2://> !table
+-----------+------------------+----------+-------------------------
+
|TABLE_SCHEM|     TABLE_NAME   | TABLE_TYPE| REMARKS
|
+-----------+------------------+----------+-------------------------
+
|default    | employee         | TABLE     | NULL
|
|default    | employee_internal| TABLE     | This is an internal table
|
|default    | employee_external| TABLE     | This is an external table
|
|default    | ctas_employee    | TABLE     | NULL
|
|default    | cte_employee     | TABLE     | NULL
|
+-----------+------------------+----------+--------------------------
+

jdbc:hive2://> ALTER TABLE cte_employee RENAME TO c_employee;
No rows affected (0.237 seconds)
```

- Alter the table's properties, such as comments:

```
jdbc:hive2://> ALTER TABLE c_employee
. . . . . . .> SET TBLPROPERTIES ('comment'='New name, comments');
No rows affected (0.239 seconds)

jdbc:hive2://> !table
+-----------+------------------+----------+-------------------------
+
|TABLE_SCHEM|     TABLE_NAME   | TABLE_TYPE| REMARKS
|
+-----------+------------------+----------+-------------------------
+
|default    | employee         | TABLE     | NULL
|
|default    | employee_internal| TABLE     | This is an internal table
|
|default    | employee_external| TABLE     | This is an external table
|
|default    | ctas_employee    | TABLE     | NULL
|
|default    | c_employee       | TABLE     | New name, comments
|
+-----------+------------------+----------+--------------------------
+
```

- Alter the table's delimiter through SERDEPROPERTIES:

```
jdbc:hive2://> ALTER TABLE employee_internal SET
. . . . . . .> SERDEPROPERTIES ('field.delim' = '$');
No rows affected (0.148 seconds)
```

- Alter the table's file format:

```
jdbc:hive2://> ALTER TABLE c_employee SET FILEFORMAT RCFILE;
No rows affected (0.235 seconds)
```

- Alter the table's location, which must be a full URI of HDFS:

```
jdbc:hive2://> ALTER TABLE c_employee
. . . . . . .> SET LOCATION
. . . . . . .> 'hdfs://localhost:8020/user/dayongd/employee';
No rows affected (0.169 seconds)
```

- Alter the table's enable/disable protection to NO_DROP, which prevents a table from being dropped, or OFFLINE, which prevents data (not metadata) in a table from being queried:

```
jdbc:hive2://> ALTER TABLE c_employee ENABLE NO_DROP;
jdbc:hive2://> ALTER TABLE c_employee DISABLE NO_DROP;
jdbc:hive2://> ALTER TABLE c_employee ENABLE OFFLINE;
jdbc:hive2://> ALTER TABLE c_employee DISABLE OFFLINE;
```

- Alter the table's concatenation to merge small files into larger files:

```
--Convert to the file format supported
jdbc:hive2://> ALTER TABLE c_employee SET FILEFORMAT ORC;
No rows affected (0.160 seconds)

--Concatenate files
jdbc:hive2://> ALTER TABLE c_employee CONCATENATE;
No rows affected (0.165 seconds)

--Convert to the regular file format
jdbc:hive2://> ALTER TABLE c_employee SET FILEFORMAT TEXTFILE;
No rows affected (0.143 seconds)
```

Note

CONCATENATE

In Hive release 0.8.0, RCFile is added to support fast block-level merging of small RCFiles using the CONCATENATE command. In Hive release 0.14.0 ORC, the files that are added support fast stripe-level merging of small ORC files using the CONCATENATE command. Other file formats are not supported yet. In case of RCFiles, the merge happens at block level and ORC files merge at stripe level thereby avoiding the overhead of decompressing and decoding the data. MapReduce is triggered when performing concatenation.

- Alter the column's data type:

```
--Check column type before changes
jdbc:hive2://> DESC employee_internal;
+---------------+-----------------------------+----------+
|    col_name   |          data_type          | comment  |
+---------------+-----------------------------+----------+
| employee_name | string                      |          |
| work_place    | array<string>               |          |
| sex_age       | struct<sex:string,age:int>  |          |
| skills_score  | map<string,int>             |          |
```

```
| depart_title   | map<string,array<string>>    |          |
+---------------+----------------------------+---------+
5 rows selected (0.119 seconds)

--Change column type and order
jdbc:hive2://> ALTER TABLE employee_internal
. . . . . . .> CHANGE name employee_name string AFTER sex_age;
No rows affected (0.23 seconds)

--Verify the changes
jdbc:hive2://> DESC employee_internal;
+---------------+----------------------------+---------+
|   col_name    |         data_type          | comment |
+---------------+----------------------------+---------+
| work_place    | array<string>              |         |
| sex_age       | struct<sex:string,age:int> |         |
| employee_name | string                     |         |
| skills_score  | map<string,int>            |         |
| depart_title  | map<string,array<string>>  |         |
+---------------+----------------------------+---------+
5 rows selected (0.214 seconds)
```

- Alter the column's type and order:

```
jdbc:hive2://> ALTER TABLE employee_internal
. . . . . . .> CHANGE employee_name name string FIRST;
No rows affected (0.238 seconds)
--Verify the changes
jdbc:hive2://> DESC employee_internal;
+---------------+----------------------------+---------+
|   col_name    |         data_type          | comment |
+---------------+----------------------------+---------+
| name          | string                     |         |
| work_place    | array<string>              |         |
| sex_age       | struct<sex:string,age:int> |         |
| skills_score  | map<string,int>            |         |
| depart_title  | map<string,array<string>>  |         |
+---------------+----------------------------+---------+
5 rows selected (0.119 seconds)
```

- Add/replace columns:

```
--Add columns to the table
jdbc:hive2://> ALTER TABLE c_employee ADD COLUMNS (work string);
No rows affected (0.184 seconds)

--Verify the added columns
jdbc:hive2://> DESC c_employee;
+-----------+-----------+----------+
| col_name  | data_type | comment  |
+-----------+-----------+----------+
| name      | string    |          |
| work      | string    |          |
+-----------+-----------+----------+
2 rows selected (0.115 seconds)

--Replace all columns
```

```
jdbc:hive2://> ALTER TABLE c_employee
. . . . . . .> REPLACE COLUMNS (name string);
No rows affected (0.132 seconds)

--Verify the replaced all columns
jdbc:hive2://> DESC c_employee;
+-----------+------------+----------+
| col_name  | data_type  | comment  |
+-----------+------------+----------+
| name      | string     |          |
+-----------+------------+----------+
1 row selected (0.129 seconds)
```

Note

The ALTER command will only modify Hive's metadata, NOT the data. Users should make sure the actual data conforms with the metadata definition manually.

Hive partitions

By default, a simple query in Hive scans the whole Hive table. This slows down the performance when querying a large-size table. The issue could be resolved by creating Hive partitions, which is very similar to what's in the RDBMS. In Hive, each partition corresponds to a predefined partition column(s) and stores it as a subdirectory in the table's directory in HDFS. When the table gets queried, only the required partitions (directory) of data in the table are queried, so the I/O and time of query is greatly reduced. It is very easy to implement Hive partitions when the table is created and check the partitions created, as follows:

```
--
Create partitions when creating tables
jdbc:hive2://> CREATE TABLE employee_partitioned
. . . . . . .> (
. . . . . . .>   name string,
. . . . . . .>   work_place ARRAY<string>,
. . . . . . .>   sex_age STRUCT<sex:string,age:int>,
. . . . . . .>   skills_score MAP<string,int>,
. . . . . . .>   depart_title MAP<STRING,ARRAY<STRING>>
. . . . . . .> )
. . . . . . .> PARTITIONED BY (Year INT, Month INT)
. . . . . . .> ROW FORMAT DELIMITED
. . . . . . .> FIELDS TERMINATED BY '|'
. . . . . . .> COLLECTION ITEMS TERMINATED BY ','
. . . . . . .> MAP KEYS TERMINATED BY ':';
No rows affected (0.293 seconds)

--Show partitions
jdbc:hive2://> SHOW PARTITIONS employee_partitioned;
+------------+
| partition  |
+------------+
+------------+
No rows selected (0.177 seconds)
```

From the preceding result, we can see that the partition is not enabled automatically. We have to use `ALTER TABLE ADD PARTITION` to add partitions to a table. The `ADD PARTITION` command changes the table's metadata, but does not load data. If the data does not exist in the partition's location, queries will not return any results. To drop the partition including both data and metadata, use the `ALTER TABLE DROP PARTITION` statement as follows:

```
--Add multiple partitions
jdbc:hive2://> ALTER TABLE employee_partitioned ADD
. . . . . . .> PARTITION (year=2014, month=11)
. . . . . . .> PARTITION (year=2014, month=12);
No rows affected (0.248 seconds)

jdbc:hive2://> SHOW PARTITIONS employee_partitioned;
+--------------------+
|     partition      |
+--------------------+
```

```
| year=2014/month=11  |
| year=2014/month=12  |
+--------------------+
2 rows selected (0.108 seconds)

--Drop the partition
jdbc:hive2://> ALTER TABLE employee_partitioned
. . . . . . .> DROP IF EXISTS PARTITION (year=2014, month=11);

jdbc:hive2://> SHOW PARTITIONS employee_partitioned;
+--------------------+
|      partition     |
+--------------------+
| year=2014/month=12  |
+--------------------+
1 row selected (0.107 seconds)
```

To avoid manually adding partitions, dynamic partition insert (or multipartition insert) is designed for dynamically determining which partitions should be created and populated while scanning the input table. This part is introduced with more detail in [Chapter 5](#), *Data Manipulation*.

To load or overwrite data in partition, we can use the `LOAD` or `INSERT OVERWRITE` statements. The statement only overwrites the data in the specified partitions. Although partition columns are subdirectory names, we can query or specify them in the `SELECT` or `WHERE` statements to narrow down the result set. The following steps show how to load data to the partition table:

- Load data to the partition:

  ```
  jdbc:hive2://> LOAD DATA LOCAL INPATH
  . . . . . . .> '/home/dayongd/Downloads/employee.txt'
  . . . . . . .> OVERWRITE INTO TABLE employee_partitioned
  . . . . . . .> PARTITION (year=2014, month=12);
  No rows affected (0.96 seconds)
  ```

- Verify the data that is loaded:

  ```
  jdbc:hive2://> SELECT name, year, month FROM employee_partitioned;
  +----------+-------+--------+
  |   name   | year  | month  |
  +----------+-------+--------+
  | Michael  | 2014  | 12     |
  | Will     | 2014  | 12     |
  | Shelley  | 2014  | 12     |
  | Lucy     | 2014  | 12     |
  +----------+-------+--------+
  4 rows selected (37.451 seconds)
  ```

- The alter table/partition statement for file format, location, protections, and concatenation has the same syntax as the alter table statements and is shown here:

  ```
  ALTER TABLE table_name PARTITION partition_spec SET FILEFORMAT
  file_format;
  ALTER TABLE table_name PARTITION partition_spec SET LOCATION 'full
  ```

```
URI';
ALTER TABLE table_name PARTITION partition_spec ENABLE NO_DROP;
ALTER TABLE table_name PARTITION partition_spec ENABLE OFFLINE;
ALTER TABLE table_name PARTITION partition_spec DISABLE NO_DROP;
ALTER TABLE table_name PARTITION partition_spec DISABLE OFFLINE;
ALTER TABLE table_name PARTITION partition_spec CONCATENATE;
```

Hive buckets

Besides partition, bucket is another technique to cluster datasets into more manageable parts to optimize query performance. Different from partition, the bucket corresponds to segments of files in HDFS. For example, the `employee_partitioned` table from the previous section uses the year and month as the top-level partition. If there is a further request to use the `employee_id` as the third level of partition, it leads to many deep and small partitions and directories. For instance, we can bucket the `employee_partitioned` table using `employee_id` as the bucket column. The value of this column will be hashed by a user-defined number into buckets. The records with the same `employee_id` will always be stored in the same bucket (segment of files). By using buckets, Hive can easily and efficiently do sampling (see [Chapter 6](#), *Data Aggregation and Sampling*) and map side joins (see [Chapter 4](#), *Data Selection and Scope*). An example to create a bucket table is as follows:

```
--Prepare another dataset and table for bucket table
jdbc:hive2://> CREATE TABLE employee_id
. . . . . . .> (
. . . . . . .>   name string,
. . . . . . .>   employee_id int,
. . . . . . .>   work_place ARRAY<string>,
. . . . . . .>   sex_age STRUCT<sex:string,age:int>,
. . . . . . .>   skills_score MAP<string,int>,
. . . . . . .>   depart_title MAP<string,ARRAY<string>>
. . . . . . .> )
. . . . . . .> ROW FORMAT DELIMITED
. . . . . . .> FIELDS TERMINATED BY '|'
. . . . . . .> COLLECTION ITEMS TERMINATED BY ','
. . . . . . .> MAP KEYS TERMINATED BY ':';
No rows affected (0.101 seconds)

jdbc:hive2://> LOAD DATA LOCAL INPATH
. . . . . . .> '/home/dayongd/Downloads/employee_id.txt'
. . . . . . .> OVERWRITE INTO TABLE employee_id
No rows affected (0.112 seconds)

--Create the buckets table
jdbc:hive2://> CREATE TABLE employee_id_buckets
. . . . . . .> (
. . . . . . .>   name string,
. . . . . . .>   employee_id int,
. . . . . . .>   work_place ARRAY<string>,
. . . . . . .>   sex_age STRUCT<sex:string,age:int>,
. . . . . . .>   skills_score MAP<string,int>,
. . . . . . .>   depart_title MAP<string,ARRAY<string >>
. . . . . . .> )
. . . . . . .> CLUSTERED BY (employee_id) INTO 2 BUCKETS
. . . . . . .> ROW FORMAT DELIMITED
. . . . . . .> FIELDS TERMINATED BY '|'
. . . . . . .> COLLECTION ITEMS TERMINATED BY ','
. . . . . . .> MAP KEYS TERMINATED BY ':';
No rows affected (0.104 seconds)
```

Note

Bucket numbers

To define the proper number of buckets, we should avoid having too much or too little of data in each bucket. A better choice is somewhere near two blocks of data. For example, we can plan 512 MB of data in each bucket, if the Hadoop block size is 256 MB. If possible, use 2N as the number of buckets.

Bucketing has close dependency on the underlying data loaded. To properly load data to a bucket table, we need to either set the maximum number of reducers to the same number of buckets specified in the table creation (for example, 2) or enable enforce bucketing as follows:

```
jdbc:hive2://> set map.reduce.tasks = 2;
No rows affected (0.026 seconds)

jdbc:hive2://> set hive.enforce.bucketing = true;
No rows affected (0.002 seconds)
```

To populate the data to the bucket table, we cannot use LOAD keywords such as what was done in the regular tables since LOAD does not verify the data against the metadata. Instead, INSERT should be used to populate the bucket table as follows:

```
jdbc:hive2://> INSERT OVERWRITE TABLE employee_id_buckets
. . . . . . .> SELECT * FROM employee_id;
No rows affected (75.468 seconds)

--Verify the buckets in the HDFS
-bash-4.1$ hdfs dfs -ls /user/hive/warehouse/employee_id_buckets
Found 2 items
-rwxrwxrwx   1 hive hive        900 2014-11-02 10:54
/user/hive/warehouse/employee_id_buckets/000000_0
-rwxrwxrwx   1 hive hive        582 2014-11-02 10:54
/user/hive/warehouse/employee_id_buckets/000001_0
```

Hive views

In Hive, views are logical data structures that can be used to simplify queries by either hiding the complexities such as joins, subqueries, and filters or by flatting the data. Unlike some RDBMS, Hive views do not store data or get materialized. Once the Hive view is created, its schema is frozen immediately. Subsequent changes to the underlying tables (for example, adding a column) will not be reflected in the view's schema. If an underlying table is dropped or changed, subsequent attempts to query the invalid view will fail, as follows:

```
jdbc:hive2://> CREATE VIEW employee_skills
. . . . . . .> AS
. . . . . . .> SELECT name, skills_score['DB'] AS DB,
. . . . . . .> skills_score['Perl'] AS Perl,
. . . . . . .> skills_score['Python'] AS Python,
. . . . . . .> skills_score['Sales'] as Sales,
. . . . . . .> skills_score['HR'] as HR
. . . . . . .> FROM employee;
No rows affected (0.253 seconds)
```

When creating views, there is no MapReduce job triggered at all since this is only a metadata change. However, a proper MapReduce job will be triggered when querying the view. Use `SHOW CREATE TABLE` or `DESC FORMATTED TABLE` to display the `CREATE VIEW` statement that created a view. The following are other Hive view DDLs:

- Alter the views' properties:

  ```
  jdbc:hive2://> ALTER VIEW employee_skills
  . . . . . . .> SET TBLPROPERTIES ('comment' = 'This is a view');
  No rows affected (0.19 seconds)
  ```

- Redefine views:

  ```
  jdbc:hive2://> ALTER VIEW employee_skills AS
  . . . . . . .> SELECT * from employee ;
  No rows affected (0.17 seconds)
  ```

- Drop views:

  ```
  jdbc:hive2://> DROP VIEW employee_skills;
  No rows affected (0.156 seconds)
  ```

Summary

After going through this chapter, we are able to define and use various data types in Hive. We should know how to create, alter, and drop tables, partitions, and views in Hive and how to use external tables, internal tables, partitions, buckets, and views in Hive.

In the next chapter, we will dive into the details of querying data by Hive.

Chapter 4. Data Selection and Scope

This chapter is about how to discover the data by querying the data, linking the data, and limiting the data ranges or scopes. The chapter mainly covers the syntax and usage of Hive `SELECT`, `WHERE`, `LIMIT`, `JOIN`, and `UNION ALL` to operate datasets.

In this chapter we will cover the following topics:

- The `SELECT` statement
- The common `JOIN` statement
- The special `JOIN` (`MAPJOIN`) statement
- The set operation statement (`UNION ALL`)

The SELECT statement

The most common use case of using Hive is to query the data in Hadoop. To achieve this, we need to write and execute the `SELECT` statement in Hive. The typical work done by the `SELECT` statement is to project the rows meeting query conditions specified in the `WHERE` clause after the target table and return the result set. The `SELECT` statement is quite often used with `FROM`, `DISTINCT`, `WHERE`, and `LIMIT` keywords. We will introduce them through examples as follows.

The `SELECT *` statement here means all the columns in the table are selected. By default, all rows are returned including duplicated rows. If the `DISTINCT` keyword is used, only unique rows from the table are selected and returned. The `LIMIT` keyword is used to limit the number of rows returned randomly. In addition, `SELECT *` scans the whole table/file without triggering MapReduce jobs, so it runs faster than `SELECT <column_name>`. Since Hive 0.10.0, the simple `SELECT` statements, such as `SELECT <column_name> FROM <table_name> LIMIT n`, can also avoid triggering the MapReduce job if the Hive fetch task conversion is enabled by setting `hive.fetch.task.conversion = more`.

The following tasks can be done:

- Query all or specific columns in the table:

```
jdbc:hive2://> SELECT * FROM employee;
+-------+-----------------+----------+---------------+---------------------------+
| name  |    work_place   | sex_age  |  skills_score |    depart_title           |
+-------+-----------------+----------+---------------+---------------------------+
|Michael|[Montreal,Toronto]|[Male,30] |{DB=80}        |{Product=[Developer,Lead]} |
|Will   |[Montreal]        |[Male,35] |{Perl=85}      |{Test=[Lead],Product=[Lead]} |
|Shelley|[New York]        |[Female,27]|{Python=80}   |{Test=[Lead],COE=[Architect]}|
|Lucy   |[Vancouver]       |[Female,57]|{Sales=89,HR=94}|{Sales=[Lead]}            |
+-------+-----------------+----------+---------------+---------------------------+
4 rows selected (0.677 seconds)

jdbc:hive2://> SELECT name FROM employee;
+----------+
|   name   |
+----------+
| Michael  |
| Will     |
| Shelley  |
| Lucy     |
+----------+
4 rows selected (162.452 seconds)
```

- Select a unique value of the specified column:

```
jdbc:hive2://> SELECT DISTINCT name FROM employee LIMIT 2;
+----------+
|   name   |
+----------+
| Lucy     |
| Michael  |
+----------+
2 rows selected (71.125 seconds)
```

- Enable fetch and verify the performance improvement:

```
jdbc:hive2://> SET hive.fetch.task.conversion=more;
No rows affected (0.002 seconds)

jdbc:hive2://> SELECT name FROM employee;
+----------+
|   name   |
+----------+
| Michael  |
| Will     |
| Shelley  |
| Lucy     |
+----------+
4 rows selected (0.242 seconds)
```

Besides LIMIT, WHERE is another generic condition clause to limit the returned result set. The WHERE condition can be any Boolean expression or user-defined functions comparing to table or partition columns:

```
jdbc:hive2://> SELECT name, work_place FROM employee
. . . . . . .> WHERE name = 'Michael';
+----------+--------------------------+
|   name   |        work_place        |
+----------+--------------------------+
| Michael  | ["Montreal","Toronto"]   |
+----------+--------------------------+
1 row selected (38.107 seconds)
```

Multiple SELECT statements can work together to build a complex query using nest or subqueries, such as JOIN and UNION. The following are a few examples to use nest/subqueries. Subqueries can be used in the format of WITH (also referred to as CTE since Hive 0.13.0), after the FROM or WHERE statement. When using subqueries, an alias should be given for the subquery (see t1 in the following example). Or else, Hive will report exceptions. The different uses of SELECT statements are as follows:

- Nested SELECT using CTE can be implemented as follows:

```
jdbc:hive2://> WITH t1 AS (
. . . . . . .> SELECT * FROM employee
. . . . . . .> WHERE sex_age.sex = 'Male')
. . . . . . .> SELECT name, sex_age.sex AS sex FROM t1;
+----------+-------+
|   name   |  sex  |
```

```
+----------+-------+
| Michael  | Male  |
| Will     | Male  |
+----------+-------+
2 rows selected (38.706 seconds)
```

- Nested `SELECT` after the `FROM` statement can be implemented as follows:

```
jdbc:hive2://> SELECT name, sex_age.sex AS sex
. . . . . . .> FROM
. . . . . . .> (
. . . . . . .>   SELECT * FROM employee
. . . . . . .>   WHERE sex_age.sex = 'Male'
. . . . . . .> ) t1;
+----------+-------+
|   name   |  sex  |
+----------+-------+
| Michael  | Male  |
| Will     | Male  |
+----------+-------+
2 rows selected (48.198 seconds)
```

The Hive subquery in the `WHERE` clause can be used with `IN`, `NOT IN`, `EXIST`, or `NOT EXIST` as follows. If the alias (see the following example for the `employee` table) is not specified before columns (name) in the `WHERE` condition, Hive will report the error *Correlating expression cannot contain unqualified column references.* This is a limitation of the Hive subquery. A subquery that uses `EXIST` or `NOT EXIST` must refer to both inner and outer expression. This is similar to the `JOIN` table, which is introduced later. This is not supported by the `IN` and `NOT IN` clause.

```
jdbc:hive2://> SELECT name, sex_age.sex AS sex
. . . . . . .> FROM employee a
. . . . . . .> WHERE a.name IN
. . . . . . .> (SELECT name FROM employee
. . . . . . .> WHERE sex_age.sex = 'Male'
. . . . . . .> );
+----------+-------+
|   name   |  sex  |
+----------+-------+
| Michael  | Male  |
| Will     | Male  |
+----------+-------+
2 rows selected (54.644 seconds)

jdbc:hive2://> SELECT name, sex_age.sex AS sex
. . . . . . .> FROM employee a
. . . . . . .> WHERE EXISTS
. . . . . . .> (SELECT * FROM employee b
. . . . . . .> WHERE a.sex_age.sex = b.sex_age.sex
. . . . . . .> AND b.sex_age.sex = 'Male'
. . . . . . .> );
+----------+-------+
|   name   |  sex  |
+----------+-------+
| Michael  | Male  |
```

```
| Will     | Male  |
+---------+-------+
2 rows selected (69.48 seconds)
```

There are additional restrictions for subqueries used in WHERE clauses:

- Subqueries can only appear on the right-hand side of the WHERE clauses
- Nested subqueries are not allowed
- The IN and NOT IN statement supports only one column

The INNER JOIN statement

Hive `JOIN` is used to combine rows from two or more tables together. Hive supports common `JOIN` operations such as what's in the RDBMS, for example, `JOIN`, `LEFT OUTER JOIN`, `RIGHT OUTER JOIN`, `FULL OUTER JOIN`, and `CROSS JOIN`. However, Hive only supports equal `JOIN` instead of unequal `JOIN`, because unequal `JOIN` is difficult to be converted to MapReduce jobs.

The `INNER JOIN` in Hive uses `JOIN` keywords, which return rows meeting the `JOIN` conditions from both left and right tables. The `INNER JOIN` keyword can also be omitted by comma-separated table names since Hive 0.13.0. See the following examples to show various inner `JOIN` statements in Hive:

- Prepare another table to join and load data:

```
jdbc:hive2://> CREATE TABLE IF NOT EXISTS employee_hr
. . . . . . .> (
. . . . . . .>   name string,
. . . . . . .>   employee_id int,
. . . . . . .>   sin_number string,
. . . . . . .>   start_date date
. . . . . . .> )
. . . . . . .> ROW FORMAT DELIMITED
. . . . . . .> FIELDS TERMINATED BY '|'
. . . . . . .> STORED AS TEXTFILE;
No rows affected (1.732 seconds)

jdbc:hive2://> LOAD DATA LOCAL INPATH
. . . . . . .> '/home/Dayongd/employee_hr.txt'
. . . . . . .> OVERWRITE INTO TABLE employee_hr;
No rows affected (0.635 seconds)
```

- Perform inner `JOIN` between two tables with equal `JOIN` conditions:

```
jdbc:hive2://> SELECT emp.name, emph.sin_number
. . . . . . .> FROM employee emp
. . . . . . .> JOIN employee_hr emph ON emp.name = emph.name;
+-----------+------------------+
| emp.name  | emph.sin_number  |
+-----------+------------------+
| Michael   | 547-968-091      |
| Will      | 527-948-090      |
| Lucy      | 577-928-094      |
+-----------+------------------+
3 rows selected (71.083 seconds)
```

- The `JOIN` operation can be performed among more tables (three tables in this case), as follows:

```
jdbc:hive2://> SELECT emp.name, empi.employee_id, emph.sin_number
. . . . . . .> FROM employee emp
. . . . . . .> JOIN employee_hr emph ON emp.name = emph.name
. . . . . . .> JOIN employee_id empi ON emp.name = empi.name;
+-----------+-------------------+-----------------+
```

```
| emp.name | empi.employee_id | emph.sin_number |
+----------+------------------+-----------------+
| Michael  | 100              | 547-968-091     |
| Will     | 101              | 527-948-090     |
| Lucy     | 103              | 577-928-094     |
+----------+------------------+-----------------+
3 rows selected (67.933 seconds)
```

- Self-join is a special `JOIN` where one table joins itself. When doing such joins, a different alias should be given to distinguish the same table:

```
jdbc:hive2://> SELECT emp.name
. . . . . . .> FROM employee emp
. . . . . . .> JOIN employee emp_b
. . . . . . .> ON emp.name = emp_b.name;
+----------+
| emp.name |
+----------+
| Michael  |
| Will     |
| Shelley  |
| Lucy     |
+----------+
4 rows selected (59.891 seconds)
```

- Implicit join is a `JOIN` operation without using the `JOIN` keyword. It is supported since Hive 0.13.0:

```
jdbc:hive2://> SELECT emp.name, emph.sin_number
. . . . . . .> FROM employee emp, employee_hr emph
. . . . . . .> WHERE emp.name = emph.name;
+----------+-----------------+
| emp.name | emph.sin_number |
+----------+-----------------+
| Michael  | 547-968-091     |
| Will     | 527-948-090     |
| Lucy     | 577-928-094     |
+----------+-----------------+
3 rows selected (47.241 seconds)
```

- The `JOIN` operation uses different columns in join conditions and will create an additional MapReduce:

```
jdbc:hive2://> SELECT emp.name, empi.employee_id, emph.sin_number
. . . . . . .> FROM employee emp
. . . . . . .> JOIN employee_hr emph ON emp.name = emph.name
. . . . . . .> JOIN employee_id empi ON emph.employee_id =
empi.employee_id;
+----------+------------------+-----------------+
| emp.name | empi.employee_id | emph.sin_number |
+----------+------------------+-----------------+
| Michael  | 100              | 547-968-091     |
| Will     | 101              | 527-948-090     |
| Lucy     | 103              | 577-928-094     |
+----------+------------------+-----------------+
3 rows selected (49.785 seconds)
```

Note

If `JOIN` uses different columns in the join conditions, it will request additional job stages to complete the join. If the `JOIN` operation uses the same column in the join conditions, Hive will join on this condition using one stage.

When `JOIN` is performed between multiple tables, the MapReduce jobs are created to process the data in the HDFS. Each of the jobs is called a stage. Usually, it is suggested for `JOIN` statements to put the big table right at the end for better performance as well as avoiding **Out Of Memory** (**OOM**) exceptions, because the last table in the sequence is streamed through the reducers where the others are buffered in the reducer by default. Also, a hint, such as `/*+STREAMTABLE (table_name)*/`, can be specified to tell which table is streamed as follows:

```
jdbc:hive2://> SELECT /*+ STREAMTABLE(employee_hr) */
. . . . . . .> emp.name, empi.employee_id, emph.sin_number
. . . . . . .> FROM employee emp
. . . . . . .> JOIN employee_hr emph ON emp.name = emph.name
. . . . . . .> JOIN employee_id empi ON emph.employee_id =
empi.employee_id;
```

The OUTER JOIN and CROSS JOIN statements

Besides `INNER JOIN`, Hive also supports regular `OUTER JOIN` and `FULL JOIN`. The logic of such `JOIN` is the same to what's in the RDBMS. The following table summarizes the differences of a common `JOIN`:

Common JOIN type	Logic	Rows returned (assume table_m has m rows and table_n has n rows)
`table_m` `JOIN` `table_n`	This returns all rows matched in both tables.	m ∩ n
`table_m` `LEFT` `[OUTER]` `JOIN` `table_n`	This returns all rows in the left table and matched rows in the right table. If there is no match in the right table, return null in the right table.	m
`table_m` `RIGHT` `[OUTER]` `JOIN` `table_n`	This returns all rows in the right table and matched rows in the left table. If there is no match in the left table, return null in the left table.	n
`table_m` `FULL` `[OUTER]` `JOIN` `table_n`	This returns all rows in both the tables and matched rows in both the tables. If there is no match in the left or right table, return null instead.	m + n - m ∩ n
`table_m` `CROSS` `JOIN` `table_n`	This returns all row combinations in both the tables to produce a Cartesian product.	m * n

The following examples demonstrate `OUTER JOIN`:

```
jdbc:hive2://> SELECT emp.name, emph.sin_number
. . . . . . .> FROM employee emp
. . . . . . .> LEFT JOIN employee_hr emph ON emp.name = emph.name;
+-----------+------------------+
| emp.name  | emph.sin_number  |
+-----------+------------------+
| Michael   | 547-968-091      |
| Will      | 527-948-090      |
| Shelley   | NULL             |
| Lucy      | 577-928-094      |
+-----------+------------------+
4 rows selected (39.637 seconds)
```

```
jdbc:hive2://> SELECT emp.name, emph.sin_number
. . . . . . .> FROM employee emp
. . . . . . .> RIGHT JOIN employee_hr emph ON emp.name = emph.name;
+-----------+------------------+
| emp.name  | emph.sin_number  |
+-----------+------------------+
| Michael   | 547-968-091      |
| Will      | 527-948-090      |
| NULL      | 647-968-598      |
| Lucy      | 577-928-094      |
+-----------+------------------+
4 rows selected (34.485 seconds)

jdbc:hive2://> SELECT emp.name, emph.sin_number
. . . . . . .> FROM employee emp
. . . . . . .> FULL JOIN employee_hr emph ON emp.name = emph.name;
+-----------+------------------+
| emp.name  | emph.sin_number  |
+-----------+------------------+
| Lucy      | 577-928-094      |
| Michael   | 547-968-091      |
| Shelley   | NULL             |
| NULL      | 647-968-598      |
| Will      | 527-948-090      |
+-----------+------------------+
5 rows selected (64.251 seconds)
```

The CROSS JOIN statement, which is available since Hive 0.10.0, does not have the JOIN condition. The CROSS JOIN statement can also be written using JOIN without condition or with the *always true* condition, such as 1 = 1. The following three ways of writing CROSS JOIN produce the same result set:

```
jdbc:hive2://> SELECT emp.name, emph.sin_number
. . . . . . .> FROM employee emp
. . . . . . .> CROSS JOIN employee_hr emph;

jdbc:hive2://> SELECT emp.name, emph.sin_number
. . . . . . .> FROM employee emp
. . . . . . .> JOIN employee_hr emph;

jdbc:hive2://> SELECT emp.name, emph.sin_number
. . . . . . .> FROM employee emp
. . . . . . .> JOIN employee_hr emph on 1=1;

+-----------+------------------+
| emp.name  | emph.sin_number  |
+-----------+------------------+
| Michael   | 547-968-091      |
| Michael   | 527-948-090      |
| Michael   | 647-968-598      |
| Michael   | 577-928-094      |
| Will      | 547-968-091      |
| Will      | 527-948-090      |
| Will      | 647-968-598      |
| Will      | 577-928-094      |
```

```
| Shelley   | 547-968-091      |
| Shelley   | 527-948-090      |
| Shelley   | 647-968-598      |
| Shelley   | 577-928-094      |
| Lucy      | 547-968-091      |
| Lucy      | 527-948-090      |
| Lucy      | 647-968-598      |
| Lucy      | 577-928-094      |
+----------+------------------+
16 rows selected (34.924 seconds)
```

In addition, JOIN always happens before WHERE. If possible, push conditions such as the JOIN conditions rather than WHERE conditions to filter the result set after JOIN immediately. What's more, JOIN is NOT commutative! It is always left associative no matter whether they are LEFT JOIN or RIGHT JOIN.

Although Hive does not support unequal JOIN explicitly, there are workarounds using CROSS JOIN and WHERE conditions mentioned in the following example:

```
jdbc:hive2://> SELECT emp.name, emph.sin_number
. . . . . . .> FROM employee emp
. . . . . . .> JOIN employee_hr emph ON emp.name <> emph.name;

Error: Error while compiling statement: FAILED: SemanticException [Error
10017]: Line 1:77 Both left and right aliases encountered in JOIN 'name'
(state=42000,code=10017)

jdbc:hive2://> SELECT emp.name, emph.sin_number
. . . . . . .> FROM employee emp
. . . . . . .> CROSS JOIN employee_hr emph WHERE emp.name <> emph.name;
+----------+------------------+
| emp.name | emph.sin_number  |
+----------+------------------+
| Michael  | 527-948-090      |
| Michael  | 647-968-598      |
| Michael  | 577-928-094      |
| Will     | 547-968-091      |
| Will     | 647-968-598      |
| Will     | 577-928-094      |
| Shelley  | 547-968-091      |
| Shelley  | 527-948-090      |
| Shelley  | 647-968-598      |
| Shelley  | 577-928-094      |
| Lucy     | 547-968-091      |
| Lucy     | 527-948-090      |
| Lucy     | 647-968-598      |
+----------+------------------+
13 rows selected (35.016 seconds)
```

Special JOIN – MAPJOIN

The `MAPJOIN` statement means doing the `JOIN` operation only by map without the reduce job. The `MAPJOIN` statement reads all the data from the small table to memory and broadcasts to all maps. During the map phase, the `JOIN` operation is performed by comparing each row of data in the big table with small tables against the join conditions. Because there is no reduce needed, the `JOIN` performance is improved. When the `hive.auto.convert.join` setting is set to `true`, Hive automatically converts the `JOIN` to `MAPJOIN` at runtime if possible instead of checking the map join hint. In addition, `MAPJOIN` can be used for unequal joins to improve performance since both `MAPJOIN` and `WHERE` are performed in the map phase. The following is an example of `MAPJOIN` that is enabled by query hint:

```
jdbc:hive2://> SELECT /*+ MAPJOIN(employee) */ emp.name, emph.sin_number
. . . . . . .> FROM employee emp
. . . . . . .> CROSS JOIN employee_hr emph WHERE emp.name <> emph.name;
```

The `MAPJOIN` operation does not support the following:

- The use of `MAPJOIN` after `UNION ALL`, `LATERAL VIEW`, `GROUP BY`/`JOIN`/`SORT BY`/`CLUSTER BY`/`DISTRIBUTE BY`
- The use of `MAPJOIN` before `UNION`, `JOIN`, and another `MAPJOIN`

The bucket map join is a special type of `MAPJOIN` that uses bucket columns (the column specified by `CLUSTERED BY` in the `CREATE` table statement) as the join condition. Instead of fetching the whole table as done by the regular map join, bucket map join only fetches the required bucket data. To enable bucket map join, we need to set `hive.optimize.bucketmapjoin = true` and make sure the buckets number is a multiple of each other. If both tables joined are sorted and bucketed with the same number of buckets, a sort-merge join can be performed instead of caching all small tables in the memory. The following additional settings are needed to enable this behavior:

```
SET hive.optimize.bucketmapjoin = true;
SET hive.optimize.bucketmapjoin.sortedmerge = true;
SET
hive.input.format=org.apache.hadoop.hive.ql.io.BucketizedHiveInputFormat;
```

The `LEFT SEMI JOIN` statement is also a type of `MAPJOIN`. Before Hive supports `IN`/`EXIST`, `LEFT SEMI JOIN` is used to implement such a request as shown in the following example. The restriction of using `LEFT SEMI JOIN` is that the right-hand side table should only be referenced in the join condition, but not in `WHERE` or `SELECT` clauses.

```
jdbc:hive2://> SELECT a.name
. . . . . . .> FROM employee a
. . . . . . .> WHERE EXISTS
. . . . . . .> (SELECT * FROM employee_id b
. . . . . . .> WHERE a.name = b.name);

jdbc:hive2://> SELECT a.name
. . . . . . .> FROM employee a
```

```
. . . . . . .> LEFT SEMI JOIN employee_id b
. . . . . . .> ON a.name = b.name;
+----------+
|  a.name  |
+----------+
| Michael  |
| Will     |
| Shelley  |
| Lucy     |
+----------+
4 rows selected (35.027 seconds)
```

Set operation – UNION ALL

To operate the result set vertically, Hive only supports UNION ALL right now. And, the result set of UNION ALL keeps duplicates if any. Before Hive 0.13.0, UNION ALL can only be used in the subquery. Since Hive 0.13.0, UNION ALL can also be used in top-level queries. The following are examples of the UNION ALL statements:

- Check the name column in the employee_hr and employee table:

```
jdbc:hive2://> SELECT name FROM employee_hr;
+----------+
|   name   |
+----------+
| Michael  |
| Will     |
| Steven   |
| Lucy     |
+----------+
4 rows selected (0.116 seconds)

jdbc:hive2://> SELECT name FROM employee;
+----------+
|   name   |
+----------+
| Michael  |
| Will     |
| Shelley  |
| Lucy     |
+----------+
4 rows selected (0.049 seconds)
```

- Use UNION on the name column from both tables, including duplications:

```
jdbc:hive2://> SELECT a.name
. . . . . . .> FROM employee a
. . . . . . .> UNION ALL
. . . . . . .> SELECT b.name
. . . . . . .> FROM employee_hr b;
+-----------+
| _u1.name  |
+-----------+
| Michael   |
| Will      |
| Shelley   |
| Lucy      |
| Michael   |
| Will      |
| Steven    |
| Lucy      |
+-----------+
8 rows selected (39.93 seconds)
```

For other set operations supported by RDBMS, such as UNION, INTERCEPT, and MINUS, we can use SELECT with the WHERE condition to implement them as follows:

- Implement `UNION` between two tables without duplications:

```
jdbc:hive2://> SELECT DISTINCT name
. . . . . . .> FROM
. . . . . . .> (
. . . . . . .>     SELECT a.name AS name
. . . . . . .>     FROM employee a
. . . . . . .>     UNION ALL
. . . . . . .>     SELECT b.name AS name
. . . . . . .>     FROM employee_hr b
. . . . . . .> ) union_set;
+----------+
|   name   |
+----------+
| Lucy     |
| Michael  |
| Shelley  |
| Steven   |
| Will     |
+----------+
5 rows selected (100.366 seconds)
```

Note

The subquery alias (such as union_set in this example) must be given to avoid a Hive syntax error.

- The `employee` table implements `INTERCEPT` on `employee_hr` using `JOIN`:

```
jdbc:hive2://> SELECT a.name
. . . . . . .> FROM employee a
. . . . . . .> JOIN employee_hr b
. . . . . . .> ON a.name = b.name;
+----------+
|  a.name  |
+----------+
| Michael  |
| Will     |
| Lucy     |
+----------+
3 rows selected (44.862 seconds)
```

- The `employee` table implements `MINUS` on `employee_hr` using `OUTER JOIN`:

```
jdbc:hive2://> SELECT a.name
. . . . . . .> FROM employee a
. . . . . . .> LEFT JOIN employee_hr b
. . . . . . .> ON a.name = b.name
. . . . . . .> WHERE b.name IS NULL;
+----------+
|  a.name  |
+----------+
| Shelley  |
+----------+
1 row selected (36.841 seconds)
```

Summary

In this chapter, you learned to use `SELECT` statements to discover the data you need. Then, we introduced Hive operations to link different datasets from vertical or horizontal directions using `JOIN` or `UNION ALL`. After going through this chapter, we should be able to use the `SELECT` statement with different `WHERE` conditions, `LIMIT`, `DISTINCT`, and complex subqueries. We should be able to understand and use different types of `JOIN` statements to link the different datasets horizontally and `UNION ALL` to combine the different datasets vertically.

In the next chapter, we will talk about the details of exchange, order, and transforming data as well as transactions in Hive.

Chapter 5. Data Manipulation

The ability to manipulate data is a critical capability in big data analysis. Manipulating data is the process of exchanging, moving, sorting, and transforming the data. This technique is used in many situations, such as cleaning data, searching patterns, creating trends, and so on. Hive offers various query statements, keywords, operators, and functions to carry out data manipulation.

In this chapter, we will cover the following topics:

- Data exchange using `LOAD`, `INSERT`, `IMPORT`, and `EXPORT`
- Order and sort
- Operators and functions
- Transaction

Data exchange – LOAD

To *move* data in Hive, it uses the `LOAD` keyword. Move here means the original data is moved to the target table/partition and does not exist in the original place anymore. The following is an example of how to move data to the Hive table or partition from local or HDFS files. The `LOCAL` keyword specifies where the files are located in the host. If the `LOCAL` keyword is not specified, the files are loaded from the full **Uniform Resource Identifier** (**URI**) specified after `INPATH` or the value from the `fs.default.name` Hive property by default. The path after `INPATH` can be a relative path or an absolute path. The path either points to a file or a folder (all files in the folder) to be loaded, but the subfolder is not allowed in the path specified. If the data is loaded into a partition table, the partition column must be specified. The `OVERWRITE` keyword is used to decide whether to append or replace the existing data in the target table/partition.

The following are the examples to load files into Hive tables:

- Load local data to the Hive table:

```
jdbc:hive2://> LOAD DATA LOCAL INPATH
. . . . . . .> '/home/dayongd/Downloads/employee_hr.txt'
. . . . . . .> OVERWRITE INTO TABLE employee_hr;
No rows affected (0.436 seconds)
```

- Load local data to the Hive partition table:

```
jdbc:hive2://> LOAD DATA LOCAL INPATH
. . . . . . .> '/home/dayongd/Downloads/employee.txt'
. . . . . . .> OVERWRITE INTO TABLE employee_partitioned
. . . . . . .> PARTITION (year=2014, month=12);
No rows affected (0.772 seconds)
```

- Load HDFS data to the Hive table using the default system path:

```
jdbc:hive2://> LOAD DATA INPATH
. . . . . . .> '/user/dayongd/employee/employee.txt'
. . . . . . .> OVERWRITE INTO TABLE employee;
No rows affected (0.453 seconds)
```

- Load HDFS data to the Hive table with full URI:

```
jdbc:hive2://> LOAD DATA INPATH
. . . . . . .>
'hdfs://[dfs_host]:8020/user/dayongd/employee/employee.txt'
. . . . . . .> OVERWRITE INTO TABLE employee;
No rows affected (0.297 seconds)
```

Data exchange – INSERT

To extract the data from Hive tables/ partitions, we can use the `INSERT` keyword. Like RDBMS, Hive supports inserting data by selecting data from other tables. This is a very common way to populate a table from existing data. The basic `INSERT` statement has the same syntax as a relational database's `INSERT`. However, Hive has improved its `INSERT` statement by supporting `OVERWRITE`, multiple `INSERT`, dynamic partition `INSERT`, as well as using `INSERT` to files. The following are a few examples:

- The following is a regular `INSERT` from the `SELECT` statement:

```
--Check the target table, which is empty.
jdbc:hive2://> SELECT name, work_place, sex_age
. . . . . . .> FROM employee;
+-------------+-------------------+---------------+
|employee.name|employee.work_place|employee.sex_age|
+-------------+-------------------+---------------+
+-------------+-------------------+---------------+
No rows selected (0.115 seconds)


--Populate data from SELECT
jdbc:hive2://> INSERT INTO TABLE employee
. . . . . . .> SELECT * FROM ctas_employee;
No rows affected (31.701 seconds)


--Verify the data loaded
jdbc:hive2://> SELECT name, work_place, sex_age FROM employee;
+-------------+---------------------+------------------------+
|employee.name|  employee.work_place |     employee.sex_age   |
+-------------+---------------------+------------------------+
| Michael     |["Montreal","Toronto"]|{"sex":"Male","age":30} |
| Will        |["Montreal"]         |{"sex":"Male","age":35} |
| Shelley     |["New York"]         |{"sex":"Female","age":27}|
| Lucy        |["Vancouver"]        |{"sex":"Female","age":57}|
+-------------+---------------------+------------------------+
4 rows selected (0.12 seconds)
```

- Insert data from the CTE statement:

```
jdbc:hive2://> WITH a AS (SELECT * FROM ctas_employee )
. . . . . . .> FROM a
. . . . . . .> INSERT OVERWRITE TABLE employee
. . . . . . .> SELECT *;
No rows affected (30.1 seconds)
```

- Run multiple `INSERT` by only scanning the source table once:

```
jdbc:hive2://> FROM ctas_employee
. . . . . . .> INSERT OVERWRITE TABLE employee
. . . . . . .> SELECT *
. . . . . . .> INSERT OVERWRITE TABLE employee_internal
. . . . . . .> SELECT * ;
No rows affected (27.919 seconds)
```

The `INSERT OVERWRITE` statement will replace the data in the target table/partition while `INSERT INTO` will append data.

When inserting data to the partitions, we need to specify the partition columns. Instead of specifying static values for static partitions, Hive also supports dynamically giving partition values. Dynamic partitions are useful when the data volume is large and we don't know what will be the partition values. For example, the date is dynamically used as partition columns.

Dynamic partition is not enabled by default. We need to set the following properties to make it work:

```
jdbc:hive2://> SET hive.exec.dynamic.partition=true;
No rows affected (0.002 seconds)
```

By default, the user must specify at least one static partition column. This is to avoid accidentally overwriting partitions. To disable this restriction, we can set the partition mode to `nonstrict` from the default strict mode before inserting into dynamic partitions as follows:

```
jdbc:hive2://> SET hive.exec.dynamic.partition.mode=nonstrict;
No rows affected (0.002 seconds)

jdbc:hive2://> INSERT INTO TABLE employee_partitioned
. . . . . . .> PARTITION(year, month)
. . . . . . .> SELECT name, array('Toronto') as work_place,
. . . . . . .> named_struct("sex","Male","age",30) as sex_age,
. . . . . . .> map("Python",90) as skills_score,
. . . . . . .> map("R&D",array('Developer')) as depart_title,
. . . . . . .> year(start_date) as year, month(start_date) as month
. . . . . . .> FROM employee_hr eh
. . . . . . .> WHERE eh.employee_id = 102;
No rows affected (29.024 seconds)
```

Note

Complex type constructors are used in the preceding example to assign a constant value to a complex data type column.

The Hive `INSERT` to files statement is the opposite operation for `LOAD`. It extracts the data from `SELECT` statements to local or HDFS files. However, it only supports the `OVERWRITE` keyword, not `INTO`. This means we cannot append data extracted to the existing files. By default, the columns are separated by `^A` and rows are separated by newlines. Since Hive 0.11.0, row separators can be specified. The following are a few examples to insert data to files:

- We can insert to local files with default row separators. In some recent version of Hadoop, the local directory path only works for a directory level less than two. We may need to set `hive.insert.into.multilevel.dirs=true` to get this fixed:

  ```
  jdbc:hive2://> INSERT OVERWRITE LOCAL DIRECTORY '/tmp/output1'
  ```

```
. . . . . . .> SELECT * FROM employee;
No rows affected (30.859 seconds)
```

Note

By default, many partial files could be created by the reducer when doing `INSERT`. To merge them into one, we can use HDFS commands, as shown in the following example:

```
hdfs dfs –getmerge hdfs://<host_name>:8020/user/dayongd/output
/tmp/test
```

- Insert to local files with specified row separators:

```
jdbc:hive2://> INSERT OVERWRITE LOCAL DIRECTORY '/tmp/output2'
. . . . . . .> ROW FORMAT DELIMITED FIELDS TERMINATED BY ','
. . . . . . .> SELECT * FROM employee;
No rows affected (31.937 seconds)

--Verify the separator
vi /tmp/output2/000000_0
Michael,Montreal^BToronto,Male^B30,DB^C80,Product^CDeveloper^DLead
Will,Montreal,Male^B35,Perl^C85,Product^CLead^BTest^CLead
Shelley,New York,Female^B27,Python^C80,Test^CLead^BCOE^CArchitect
Lucy,Vancouver,Female^B57,Sales^C89^BHR^C94,Sales^CLead
```

- Fire multiple `INSERT` statements from the same table `SELECT` statement:

```
jdbc:hive2://> FROM employee
. . . . . . .> INSERT OVERWRITE DIRECTORY '/user/dayongd/output'
. . . . . . .> SELECT *
. . . . . . .> INSERT OVERWRITE DIRECTORY '/user/dayongd/output1'
. . . . . . .> SELECT * ;
No rows affected (25.4 seconds)
```

Note

Besides the Hive `INSERT` statement, Hive and HDFS shell commands can also be used to extract data to local or remote files with both append and overwrite support. The `hive -e 'quoted_hql_string'` or `hive -f <hql_filename>` commands can execute a Hive query statement or query file. Linux redirect operators and piping can be used with these commands to redirect result sets. The following are a few examples:

- Append to local files:

```
$ hive -e 'select * from employee' >> test
```

- Overwrite to local files:

```
$ hive -e 'select * from employee' > test
```

- Append to HDFS files:

```
$ hive -e 'select * from employee'|hdfs dfs -appendToFile -
/user/dayongd/output2/test
```

- Overwrite to HDFS files:

```
$ hive -e 'select * from employee'|hdfs dfs -put -f - /user/dayongd/output2/test
```

Data exchange – EXPORT and IMPORT

When working with Hive, sometimes we need to migrate data among different environments. Or we may need to back up some data. Since Hive 0.8.0, `EXPORT` and `IMPORT` statements are available to support the import and export of data in HDFS for data migration or backup/restore purposes.

The `EXPORT` statement will export both data and metadata from a table or partition. Metadata is exported in a file called `_metadata`. Data is exported in a subdirectory called data:

```
jdbc:hive2://> EXPORT TABLE employee TO '/user/dayongd/output3';
No rows affected (0.19 seconds)
```

After `EXPORT`, we can manually copy the exported files to other Hive instances or use Hadoop `distcp` commands to copy to other HDFS clusters. Then, we can import the data in the following manner:

- Import data to a table with the same name. It throws an error if the table exists:

  ```
  jdbc:hive2://> IMPORT FROM '/user/dayongd/output3';
  Error: Error while compiling statement: FAILED: SemanticException
  [Error 10119]: Table exists and contains data files
  (state=42000,code=10119)
  ```

- Import data to a new table:

  ```
  jdbc:hive2://> IMPORT TABLE empolyee_imported FROM
  . . . . . . .> '/user/dayongd/output3';
  No rows affected (0.788 seconds)
  ```

- Import data to an external table, where the `LOCATION` property is optional:

  ```
  jdbc:hive2://> IMPORT EXTERNAL TABLE empolyee_imported_external
  . . . . . . .> FROM '/user/dayongd/output3'
  . . . . . . .> LOCATION '/user/dayongd/output4' ;
  No rows affected (0.256 seconds)
  ```

- Export and import partitions:

  ```
  jdbc:hive2://> EXPORT TABLE employee_partitioned partition
  . . . . . . .> (year=2014, month=11) TO '/user/dayongd/output5';
  No rows affected (0.247 seconds)

  jdbc:hive2://> IMPORT TABLE employee_partitioned_imported
  . . . . . . .> FROM '/user/dayongd/output5';
  No rows affected (0.14 seconds)
  ```

ORDER and SORT

Another aspect to manipulate data in Hive is to properly order or sort the data or result sets to clearly identify the important facts, such as top N values, maximum, minimum, and so on.

There are the following keywords used in Hive to order and sort data:

- `ORDER BY (ASC|DESC)`: This is similar to the RDBMS `ORDER BY` statement. A sorted order is maintained across all of the output from every reducer. It performs the global sort using only one reducer, so it takes a longer time to return the result. Usage with `LIMIT` is strongly recommended for `ORDER BY`. When `hive.mapred.mode = strict` (by default, `hive.mapred.mode = nonstrict`) is set and we do not specify `LIMIT`, there are exceptions. This can be used as follows:

  ```
  jdbc:hive2://> SELECT name FROM employee ORDER BY NAME DESC;
  +----------+
  |   name   |
  +----------+
  | Will     |
  | Shelley  |
  | Michael  |
  | Lucy     |
  +----------+
  4 rows selected (57.057 seconds)
  ```

- `SORT BY (ASC|DESC)`: This indicates which columns to sort when ordering the reducer input records. This means it completes sorting before sending data to the reducer. The `SORT BY` statement does not perform a global sort and only makes sure data is locally sorted in each reducer unless we set `mapred.reduce.tasks=1`. In this case, it is equal to the result of `ORDER BY`. It can be used as follows:

  ```
  --Use more than 1 reducer
  jdbc:hive2://> SET mapred.reduce.tasks = 2;
  No rows affected (0.001 seconds)

  jdbc:hive2://> SELECT name FROM employee SORT BY NAME DESC;
  +----------+
  |   name   |
  +----------+
  | Shelley  |
  | Michael  |
  | Lucy     |
  | Will     |
  +----------+
  4 rows selected (54.386 seconds)

  --Use only 1 reducer
  jdbc:hive2://> SET mapred.reduce.tasks = 1;
  No rows affected (0.002 seconds)

  jdbc:hive2://> SELECT name FROM employee SORT BY NAME DESC;
  +----------+
  ```

```
|   name   |
+----------+
| Will     |
| Shelley  |
| Michael  |
| Lucy     |
+----------+
4 rows selected (46.03 seconds)
```

- DISTRIBUTE BY: Rows with matching column values will be partitioned to the same reducer. When used alone, it does not guarantee sorted input to the reducer. The DISTRIBUTE BY statement is similar to GROUP BY in RDBMS in terms of deciding which reducer to distribute the mapper output to. When using with SORT BY, DISTRIBUTE BY must be specified before the SORT BY statement. And, the column used to distribute must appear in the select column list. It can be used as follows:

```
jdbc:hive2://> SELECT name
. . . . . . .> FROM employee_hr DISTRIBUTE BY employee_id;
Error: Error while compiling statement: FAILED: SemanticException
[Error 10004]: Line 1:44 Invalid table alias or column reference
'employee_id': (possible column names are: name)
(state=42000,code=10004)

jdbc:hive2://> SELECT name, employee_id
. . . . . . .> FROM employee_hr DISTRIBUTE BY employee_id;
+----------+--------------+
|   name   | employee_id  |
+----------+--------------+
| Lucy     | 103          |
| Steven   | 102          |
| Will     | 101          |
| Michael  | 100          |
+----------+--------------+
4 rows selected (38.92 seconds)

--Used with SORT BY
jdbc:hive2://> SELECT name, employee_id
. . . . . . .> FROM employee_hr
. . . . . . .> DISTRIBUTE BY employee_id SORT BY name;
+----------+--------------+
|   name   | employee_id  |
+----------+--------------+
| Lucy     | 103          |
| Michael  | 100          |
| Steven   | 102          |
| Will     | 101          |
+----------+--------------+
4 rows selected (38.01 seconds)
```

- CLUSTER BY: This is a shorthand operator to perform DISTRIBUTE BY and SORT BY operations on the same group of columns. And, it is sorted locally in each reducer. The CLUSTER BY statement does not support ASC or DESC yet. Compared to ORDER BY, which is globally sorted, the CLUSTER BY operation is sorted in each distributed group. To fully utilize all the available reducers when doing a global sort, we can do

`CLUSTER BY` first and then `ORDER BY`. This can be used as follows:

```
jdbc:hive2://> SELECT name, employee_id
. . . . . . .> FROM employee_hr CLUSTER BY name;
+----------+--------------+
|   name   | employee_id  |
+----------+--------------+
| Lucy     | 103          |
| Michael  | 100          |
| Steven   | 102          |
| Will     | 101          |
+----------+--------------+
4 rows selected (39.791 seconds)
```

The difference between `ORDER BY` and `CLUSTER BY` can be seen in the following diagram:
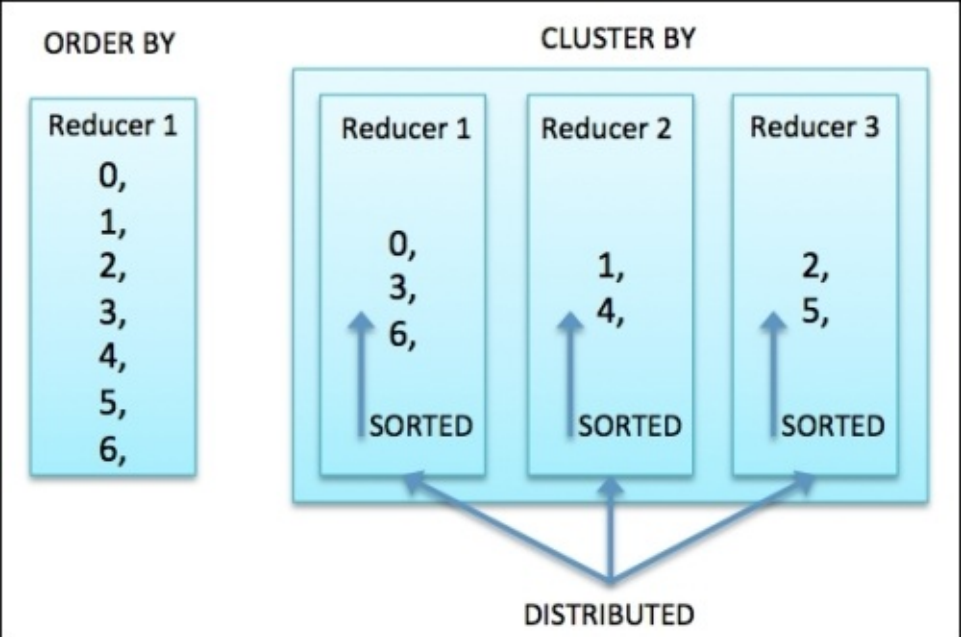
Operators and functions

To further manipulate data, we can also use expressions, operators, and functions in Hive to transform data. The Hive wiki (https://cwiki.apache.org/confluence/display/Hive/LanguageManual+UDF) has offered specifications for each expression and function, so we do not want to repeat all of them here except a few important usages or tips in this chapter.

Hive has defined relational operators, arithmetic operators, logical operators, complex type constructors, and complex type operators. For relational, arithmetic, and logical operators, they are similar to standard operators in SQL/Java. We do not repeat them again in this chapter. For operators on a complex data type, we have already introduced them in the *Understanding Hive data types* section of Chapter 3, *Data Definition and Description*, as well as the example for a dynamic partition insert in this chapter.

The functions in Hive are categorized as follows:

- **Mathematical functions**: These functions are mainly used to perform mathematical calculations, such as `RAND()` and `E()`.
- **Collection functions**: These functions are used to find the size, keys, and values for complex types, such as `SIZE(Array<T>)`.
- **Type conversion functions**: These are mainly `CAST` and `BINARY` functions to convert one type to the other.
- **Date functions**: These functions are used to perform date-related calculations, such as `YEAR(string date)` and `MONTH(string date)`.
- **Conditional functions**: These functions are used to check specific conditions with a defined value returned, such as `COALESCE`, `IF`, and `CASE WHEN`.
- **String functions**: These functions are used to perform string-related operations, such as `UPPER(string A)` and `TRIM(string A)`.
- **Aggregate functions**: These functions are used to perform aggregation (which is introduced in the next chapter for more details), such as `SUM()`, `COUNT(*)`.
- **Table-generating functions**: These functions transform a single input row into multiple output rows, such as `EXPLODE(MAP)` and `JSON_TUPLE(jsonString, k1, k2, …)`.
- **Customized functions**: These functions are created by Java code as extensions for Hive. They are introduced in Chapter 8, *Extensibility Considerations*.

To list Hive built-in functions/UDF, we can use the following commands in Hive CLI:

```
SHOW FUNCTIONS; --List all functions
DESCRIBE FUNCTION <function_name>; --Detail for specified function
DESCRIBE FUNCTION EXTENDED <function_name>; --Even more details
```

The following are a few examples and tips for using these functions:

- **Complex data type functions tips**: The `SIZE` type is used to calculate the size for `MAP`, `ARRAY`, or nested `MAP`/`ARRAY`. It returns `-1` if the size is unknown. It can be implemented as follows:

```
jdbc:hive2://> SELECT work_place, skills_score, depart_title
. . . . . . .> FROM employee;
+---------------------+-------------------+------------------------
----------+
|      work_place     |    skills_score   |     depart_title
|
+---------------------+-------------------+------------------------
----------+
|["Montreal","Toronto"]|{"DB":80}          |{"Product":
["Developer","Lead"]}   |
|["Montreal"]          |{"Perl":85}        |{"Product":
["Lead"],"Test":["Lead"]} |
|["New York"]          |{"Python":80}      |{"Test":["Lead"],"COE":
["Architect"]}|
|["Vancouver"]         |{"Sales":89,"HR":94}|{"Sales":["Lead"]}
|
+---------------------+-------------------+------------------------
----------+
4 rows selected (0.084 seconds)

jdbc:hive2://> SELECT SIZE(work_place) AS array_size,
. . . . . . .> SIZE(skills_score) AS map_size,
. . . . . . .> SIZE(depart_title) AS complex_size,
. . . . . . .> SIZE(depart_title["Product"]) AS nest_size
. . . . . . .> FROM employee;
+------------+----------+--------------+-----------+
| array_size | map_size | complex_size | nest_size |
+------------+----------+--------------+-----------+
| 2          | 1        | 1            | 2         |
| 1          | 1        | 2            | 1         |
| 1          | 1        | 2            | -1        |
| 1          | 2        | 1            | -1        |
+------------+----------+--------------+-----------+
4 rows selected (0.062 seconds)
```

The ARRAY_CONTAINS statement checks whether the array contains some values to return TRUE or FALSE. The SORT_ARRAY statement sorts the array in ascending order. These can be used as follows:

```
jdbc:hive2://> SELECT ARRAY_CONTAINS(work_place, 'Toronto')
. . . . . . .> AS is_Toronto,
. . . . . . .> SORT_ARRAY(work_place) AS sorted_array
. . . . . . .> FROM employee;
+------------+-------------------------+
| is_toronto |       sorted_array      |
+------------+-------------------------+
| true       | ["Montreal","Toronto"]  |
| false      | ["Montreal"]            |
| false      | ["New York"]            |
| false      | ["Vancouver"]           |
+------------+-------------------------+
4 rows selected (0.059 seconds)
```

- **Date function tips**: The FROM_UNIXTIME(UNIX_TIMESTAMP()) statement performs the same function as SYSDATE in Oracle. It dynamically returns the current date-time in

the Hive server, as follows:

```
jdbc:hive2://> SELECT
. . . . . . .> FROM_UNIXTIME(UNIX_TIMESTAMP()) AS current_time
. . . . . . .> FROM employee LIMIT 1;
+---------------------+
|     current_time    |
+---------------------+
| 2014-11-15 19:28:29 |
+---------------------+
1 row selected (0.047 seconds)
```

The `UNIX_TIMESTAMP()` statement can be used to compare two dates or can be used after `ORDER BY` to properly order the different string types of a date value, such as `ORDER BY UNIX_TIMESTAMP(string_date, 'dd-MM-yyyy')`. This can be used as follows:

```
--To compare the difference between two dates.
jdbc:hive2://> SELECT (UNIX_TIMESTAMP ('2015-01-21 18:00:00')
. . . . . . .> - UNIX_TIMESTAMP('2015-01-10 11:00:00'))/60/60/24
. . . . . . .> AS daydiff FROM employee LIMIT 1;
+--------------------+
|       daydiff      |
+--------------------+
| 11.291666666666666 |
+--------------------+
1 row selected (0.093 seconds)
```

The `TO_DATE` statement removes the hours, minutes, and seconds from a date. This is useful when we need to check whether the value of date-time type columns is within the data range, such as `WHERE TO_DATE(update_datetime) BETWEEN '2014-11-01' AND '2014-11-31'`. This can be used as follows:

```
jdbc:hive2://> SELECT TO_DATE(FROM_UNIXTIME(UNIX_TIMESTAMP()))
. . . . . . .> AS current_date FROM employee LIMIT 1;
+--------------+
| current_date |
+--------------+
| 2014-11-15   |
+--------------+
1 row selected (0.153 seconds)
```

- **CASE for different data types**: Before Hive 0.13.0, the data type after `THEN` or `ELSE` needed to be the same. Otherwise, it would give an exception, such as *The expression after ELSE should have the same type as those after THEN: "bigint" is expected but "int" is found*. The workaround is to use `IF`. In Hive 0.13.0, this gets fixed, as shown here:

```
jdbc:hive2://> SELECT
. . . . . . .> CASE WHEN 1 IS NULL THEN 'TRUE' ELSE 0 END
. . . . . . .> AS case_result FROM employee LIMIT 1;
+-------------+
| case_result |
+-------------+
```

```
| 0              |
+---------------+
1 row selected (0.063 seconds)
```

- **Parser and search tips**: The `LATERAL VIEW` statement is used with user-defined table generating functions such as `EXPLODE()` to flatten the map or array type of a column. The `explode` function can be used on both `ARRAY` and `MAP` with `LATERAL VIEW`. If even one of the columns exploded is `NULL`, the whole row is filtered out, such as the row of `Steven` in the following example. To avoid this, `OUTER LATERAL VIEW` can be used as follows since Hive 0.12.0:

```
--Prepare data
jdbc:hive2://> INSERT INTO TABLE employee
. . . . . . .> SELECT 'Steven' AS name, array(null) as work_place,
. . . . . . .> named_struct("sex","Male","age",30) as sex_age,
. . . . . . .> map("Python",90) as skills_score,
. . . . . . .> map("R&D",array('Developer')) as depart_title
. . . . . . .> FROM employee LIMIT 1;
No rows affected (28.187 seconds)

jdbc:hive2://> SELECT name, work_place, skills_score
. . . . . . .> FROM employee;
+----------+-----------------------+-----------------------+
|   name   |      work_place       |     skills_score      |
+----------+-----------------------+-----------------------+
| Michael  | ["Montreal","Toronto"] | {"DB":80}             |
| Will     | ["Montreal"]          | {"Perl":85}           |
| Shelley  | ["New York"]          | {"Python":80}         |
| Lucy     | ["Vancouver"]         | {"Sales":89,"HR":94}  |
| Steven   | NULL                  | {"Python":90}         |
+----------+-----------------------+-----------------------+
5 rows selected (0.053 seconds)

--LATERAL VIEW ignores the rows when EXPLORE returns NULL
jdbc:hive2://> SELECT name, workplace, skills, score
. . . . . . .> FROM employee
. . . . . . .> LATERAL VIEW explode(work_place) wp AS workplace
. . . . . . .> LATERAL VIEW explode(skills_score) ss
. . . . . . .> AS skills, score;
+----------+-----------+---------+--------+
|   name   | workplace | skills  | score  |
+----------+-----------+---------+--------+
| Michael  | Montreal  | DB      | 80     |
| Michael  | Toronto   | DB      | 80     |
| Will     | Montreal  | Perl    | 85     |
| Shelley  | New York  | Python  | 80     |
| Lucy     | Vancouver | Sales   | 89     |
| Lucy     | Vancouver | HR      | 94     |
+----------+-----------+---------+--------+
6 rows selected (24.733 seconds)

--OUTER LATERAL VIEW keeps rows when EXPLORE returns NULL
jdbc:hive2://> SELECT name, workplace, skills, score
. . . . . . .> FROM employee
. . . . . . .> LATERAL VIEW OUTER explode(work_place) wp
```

```
. . . . . . .> AS workplace
. . . . . . .> LATERAL VIEW explode(skills_score) ss
. . . . . . .> AS skills, score;
+---------+-----------+---------+--------+
|  name   | workplace | skills  | score  |
+---------+-----------+---------+--------+
| Michael | Montreal  | DB      | 80     |
| Michael | Toronto   | DB      | 80     |
| Will    | Montreal  | Perl    | 85     |
| Shelley | New York  | Python  | 80     |
| Lucy    | Vancouver | Sales   | 89     |
| Lucy    | Vancouver | HR      | 94     |
| Steven  | None      | Python  | 90     |
+---------+-----------+---------+--------+
7 rows selected (24.573 seconds)
```

The REVERSE statement can be used to reverse the order of each letter in a string. The SPLIT statement can be used to tokenize the string using a specified tokenizer. The following is an example of using them to get the filename from a Linux path:

```
jdbc:hive2://> SELECT
. . . . . . .> reverse(split(reverse('/home/user/employee.txt'),'/')
[0])
. . . . . . .> AS linux_file_name FROM employee LIMIT 1;
+------------------+
| linux_file_name  |
+------------------+
| employee.txt     |
+------------------+
1 row selected (0.1 seconds)
```

Whereas reverse outputs each element in an array or map as separate rows, collect_set and collect_list does the opposite by returning a set with elements from each row. The collect_set statement will remove duplications from the result, but collect_list does not. This is shown here:

```
jdbc:hive2://> SELECT collect_set(work_place[0])
. . . . . . .> AS flat_workplace0 FROM employee;
+-------------------------------------+
|            flat_workplace0          |
+-------------------------------------+
| ["Vancouver","Montreal","New York"] |
+-------------------------------------+
1 row selected (43.455 seconds)

jdbc:hive2://> SELECT collect_list(work_place[0])
. . . . . . .> AS flat_workplace0 FROM employee;
+-----------------------------------------------+
|                flat_workplace0                |
+-----------------------------------------------+
| ["Montreal","Montreal","New York","Vancouver"] |
+-----------------------------------------------+
1 row selected (45.488 seconds)
```

- **Virtual columns**: Virtual columns are special function type of columns in Hive.

Right now, Hive offers two virtual columns: INPUT__FILE__NAME and BLOCK__OFFSET__INSIDE__FILE. The INPUT__FILE__NAME function is the input file's name for a mapper task. The BLOCK__OFFSET__INSIDE__FILE function is the current global file position or current block's file offset if the file is compressed. The following are examples to use virtual columns to know the place where the data is physically located in the HDFS, especially for bucketed and partitioned tables:

```
jdbc:hive2://> SELECT INPUT__FILE__NAME,
. . . . . . .> BLOCK__OFFSET__INSIDE__FILE AS OFFSIDE
. . . . . . .> FROM employee_id_buckets;
+----------------------------------------------------------+----------+
|               input__file__name                          | offside  |
+----------------------------------------------------------+----------+
| hdfs://hive_warehouse_URI/employee_id_buckets/000000_0   | 0        |
| hdfs://hive_warehouse_URI/employee_id_buckets/000000_0   | 55       |
| hdfs://hive_warehouse_URI/employee_id_buckets/000000_0   | 120      |
| hdfs://hive_warehouse_URI/employee_id_buckets/000000_0   | 175      |
| hdfs://hive_warehouse_URI/employee_id_buckets/000000_0   | 240      |
| hdfs://hive_warehouse_URI/employee_id_buckets/000000_0   | 295      |
| hdfs://hive_warehouse_URI/employee_id_buckets/000000_0   | 360      |
| hdfs://hive_warehouse_URI/employee_id_buckets/000000_0   | 415      |
| hdfs://hive_warehouse_URI/employee_id_buckets/000000_0   | 480      |
| hdfs://hive_warehouse_URI/employee_id_buckets/000000_0   | 535      |
| hdfs://hive_warehouse_URI/employee_id_buckets/000000_0   | 592      |
| hdfs://hive_warehouse_URI/employee_id_buckets/000000_0   | 657      |
| hdfs://hive_warehouse_URI/employee_id_buckets/000000_0   | 712      |
| hdfs://hive_warehouse_URI/employee_id_buckets/000000_0   | 769      |
| hdfs://hive_warehouse_URI/employee_id_buckets/000000_0   | 834      |
| hdfs://hive_warehouse_URI/employee_id_buckets/000001_0   | 0        |
| hdfs://hive_warehouse_URI/employee_id_buckets/000001_0   | 57       |
| hdfs://hive_warehouse_URI/employee_id_buckets/000001_0   | 122      |
| hdfs://hive_warehouse_URI/employee_id_buckets/000001_0   | 177      |
| hdfs://hive_warehouse_URI/employee_id_buckets/000001_0   | 234      |
| hdfs://hive_warehouse_URI/employee_id_buckets/000001_0   | 291      |
| hdfs://hive_warehouse_URI/employee_id_buckets/000001_0   | 348      |
| hdfs://hive_warehouse_URI/employee_id_buckets/000001_0   | 405      |
| hdfs://hive_warehouse_URI/employee_id_buckets/000001_0   | 462      |
| hdfs://hive_warehouse_URI/employee_id_buckets/000001_0   | 517      |
+----------------------------------------------------------+----------+
25 rows selected (0.073 seconds)

jdbc:hive2://> SELECT INPUT__FILE__NAME FROM employee_partitioned;
+-------------------------------------------------------------------------+
|                          input__file__name                              |
+-------------------------------------------------------------------------+
|hdfs://warehouse_URI/employee_partitioned/year=2010/month=1/000000_0
|hdfs://warehouse_URI/employee_partitioned/year=2012/month=11/000000_0
|hdfs://warehouse_URI/employee_partitioned/year=2014/month=12/employee.
txt
|hdfs://warehouse_URI/employee_partitioned/year=2014/month=12/employee.
txt
```

```
|hdfs://warehouse_URI/employee_partitioned/year=2014/month=12/employee.
txt
|hdfs://warehouse_URI/employee_partitioned/year=2014/month=12/employee.
txt
|hdfs://warehouse_URI/employee_partitioned/year=2015/month=01/000000_0
|hdfs://warehouse_URI/employee_partitioned/year=2015/month=01/000000_0
|hdfs://warehouse_URI/employee_partitioned/year=2015/month=01/000000_0
|hdfs://warehouse_URI/employee_partitioned/year=2015/month=01/000000_0
+----------------------------------------------------------------------
---+
10 rows selected (0.47 seconds)
```

- **Functions not mentioned in the Hive wiki**: The following are the functions not mentioned in the Hive wiki:

```
--Functions to check for null values
jdbc:hive2://> SELECT work_place, isnull(work_place) is_null,
. . . . . . .> isnotnull(work_place) is_not_null FROM employee;
+-------------------------+---------+-------------+
|        work_place       | is_null | is_not_null |
+-------------------------+---------+-------------+
| ["Montreal","Toronto"]  | false   | true        |
| ["Montreal"]            | false   | true        |
| ["New York"]            | false   | true        |
| ["Vancouver"]           | false   | true        |
| NULL                    | true    | false       |
+-------------------------+---------+-------------+
5 rows selected (0.058 seconds)

--assert_true, throw an exception if 'condition' is not true.
jdbc:hive2://> SELECT assert_true(work_place IS NULL)
. . . . . . .> FROM employee;
Error: java.io.IOException:
org.apache.hadoop.hive.ql.metadata.HiveException: ASSERT_TRUE():
assertion failed. (state=,code=0)

--elt(n, str1, str2, ...),returns the n-th string
jdbc:hive2://> SELECT elt(2,'New York','Montreal','Toronto')
. . . . . . .> FROM employee LIMIT 1;
+-----------+
|    _c0    |
+-----------+
| Montreal  |
+-----------+
1 row selected (0.055 seconds)

--Return the name of current_database since Hive 0.13.0
jdbc:hive2://> SELECT current_database();
+----------+
|    _c0   |
+----------+
| default  |
+----------+
1 row selected (0.057 seconds)
```

Transactions

Before Hive version 0.13.0, Hive does not support row-level transactions. As a result, there is no way to update, insert, or delete rows of data. Hence, data overwrite can only happen on tables or partitions. This makes Hive very difficult when dealing with concurrent read/write and data-cleaning use cases.

Since Hive version 0.13.0, Hive fully supports row-level transactions by offering full **Atomicity, Consistency, Isolation, and Durability** (**ACID**) to Hive. For now, all the transactions are autocommuted and only support data in the **Optimized Row Columnar** (**ORC**) file (available since Hive 0.11.0) format and in bucketed tables.

The following configuration parameters must be set appropriately to turn on transaction support in Hive:

```
SET hive.support.concurrency = true;
SET hive.enforce.bucketing = true;
SET hive.exec.dynamic.partition.mode = nonstrict;
SET hive.txn.manager = org.apache.hadoop.hive.ql.lockmgr.DbTxnManager;
SET hive.compactor.initiator.on = true;
SET hive.compactor.worker.threads = 1;
```

The SHOW TRANSACTIONS command is added since Hive 0.13.0 to show currently open and aborted transactions in the system:

```
jdbc:hive2://> SHOW TRANSACTIONS;
+-----------------+--------------------+-------+-----------+
|      txnid      |        state       | user  |   host    |
+-----------------+--------------------+-------+-----------+
| Transaction ID  | Transaction State  | User  | Hostname  |
+-----------------+--------------------+-------+-----------+
1 row selected (15.209 seconds)
```

Since Hive 0.14.0, the INSERT VALUE, UPDATE, and DELETE commands are added to operate rows with the following syntax:

```
INSERT INTO TABLE tablename [PARTITION (partcol1[=val1], partcol2[=val2]
...)]
VALUES values_row [, values_row …];

UPDATE tablename SET column = value [, column = value…] [WHERE expression]

DELETE FROM tablename [WHERE expression]
```

Summary

In this chapter, we covered how to exchange data between Hive and files using the `LOAD`, `INSERT`, `IMPORT`, and `EXPORT` keywords. Then, we introduced the different Hive ordering and sorting options. We also covered some commonly used tips using Hive functions. Finally, we provided an overview of row-level transactions that are newly supported since Hive 0.13.0. After going through this chapter, we should be able to import or export data to Hive. We should be experienced in using different types of ordering and sorting keywords, Hive functions, and transactions.

In the next chapter, we'll look at the different ways of carrying out data aggregations and sampling in Hive.

Chapter 6. Data Aggregation and Sampling

This chapter is about how to aggregate and sample data in Hive. It firstly covers the usage of several aggregation functions, analytic functions working with `GROUP BY` and `PARTITION BY`, and windowing clauses. Then, it introduces different ways of sampling data in Hive.

In this chapter, we will cover the following topics:

- Basic aggregation
- Advanced aggregation
- Aggregation condition
- Analytic functions
- Sampling

Basic aggregation – GROUP BY

Data aggregation is any process to gather and express data in a summary form to get more information about particular groups based on specific conditions. Hive offers several built-in aggregate functions, such as MAX, MIN, AVG, and so on. Hive also supports advanced aggregation by using GROUPING SETS, ROLLUP, CUBE, analytic functions, and windowing.

The Hive basic built-in aggregate functions are usually used with the GROUP BY clause. If there is no GROUP BY clause specified, it aggregates over the whole table by default. Besides aggregate functions, all other columns that are selected must also be included in the GROUP BY clause. The following are a few examples using the built-in aggregate functions:

- Aggregation without GROUP BY columns:

```
jdbc:hive2://> SELECT count(*) AS row_cnt FROM employee;
+----------+
| row_cnt  |
+----------+
| 5        |
+----------+
1 row selected (60.709 seconds)
```

- Aggregation with GROUP BY columns:

```
jdbc:hive2://> SELECT sex_age.sex, count(*) AS row_cnt
. . . . . . .> FROM employee
. . . . . . .> GROUP BY sex_age.sex;
+--------------+----------+
| sex_age.sex  | row_cnt  |
+--------------+----------+
| Female       | 2        |
| Male         | 3        |
+--------------+----------+
2 rows selected (100.565 seconds)

--The column name selected is not group by columns
jdbc:hive2://> SELECT name, sex_age.sex, count(*) AS row_cnt
. . . . . . .> FROM employee GROUP BY sex_age.sex;
Error: Error while compiling statement: FAILED: SemanticException
[Error 10025]: Line 1:7 Expression not in GROUP BY key 'name'
(state=42000,code=10025)
```

If we have to select the columns that are not GROUP BY columns, one way is to use analytic functions, which are introduced later, to completely avoid using the GROUP BY clause. The other way is to use the collect_set function, which returns a set of objects with duplicate elements eliminated as follows:

```
--Find row count by sex and a sampled age for each sex
jdbc:hive2://> SELECT sex_age.sex,
. . . . . . .> collect_set(sex_age.age)[0] AS random_age,
. . . . . . .> count(*) AS row_cnt
. . . . . . .> FROM employee GROUP BY sex_age.sex;
```

```
+-------------+-------------+----------+
| sex_age.sex | random_age  | row_cnt  |
+-------------+-------------+----------+
| Female      | 27          | 2        |
| Male        | 35          | 3        |
+-------------+-------------+----------+
2 rows selected (48.15 seconds)
```

The aggregate function can be used with other aggregate functions in the same `select` statement. It can also be used with other functions, such as conditional functions, in the nested way. However, nested aggregate functions are not supported. See the following examples for more details:

- Multiple aggregate functions are called in the same `SELECT` statement, as follows:

  ```
  jdbc:hive2://> SELECT sex_age.sex, AVG(sex_age.age) AS avg_age,
  . . . . . . .> count(*) AS row_cnt
  . . . . . . .> FROM employee GROUP BY sex_age.sex;
  +-------------+---------------------+----------+
  | sex_age.sex |        avg_age      | row_cnt  |
  +-------------+---------------------+----------+
  | Female      | 42.0                | 2        |
  | Male        | 31.666666666666668  | 3        |
  +-------------+---------------------+----------+
  2 rows selected (98.857 seconds)
  ```

- These aggregate functions are used with `CASE WHEN`, as follows:

  ```
  jdbc:hive2://> SELECT sum(CASE WHEN sex_age.sex = 'Male'
  . . . . . . .> THEN sex_age.age ELSE 0 END)/
  . . . . . . .> count(CASE WHEN sex_age.sex = 'Male' THEN 1
  . . . . . . .> ELSE NULL END) AS male_age_avg FROM employee;
  +---------------------+
  |    male_age_avg     |
  +---------------------+
  | 31.666666666666668  |
  +---------------------+
  1 row selected (38.415 seconds)
  ```

- These aggregate functions are used with `COALESCE` and `IF`, as follows:

  ```
  jdbc:hive2://> SELECT
  . . . . . . .> sum(coalesce(sex_age.age,0)) AS age_sum,
  . . . . . . .> sum(if(sex_age.sex = 'Female',sex_age.age,0))
  . . . . . . .> AS female_age_sum FROM employee;
  +---------+---------------+
  | age_sum | female_age_sum|
  +---------+---------------+
  | 179     | 84            |
  +---------+---------------+
  1 row selected (42.137 seconds)
  ```

- Nested aggregate functions are not allowed, as shown here:

  ```
  jdbc:hive2://> SELECT avg(count(*)) AS row_cnt
  . . . . . . .> FROM employee;
  Error: Error while compiling statement: FAILED: SemanticException
  ```

```
    [Error 10128]: Line 1:11 Not yet supported place for UDAF 'count'
    (state=42000,code=10128)
```

Aggregate functions can also be used with the DISTINCT keyword to do aggregation on unique values:

```
jdbc:hive2://> SELECT count(DISTINCT sex_age.sex) AS sex_uni_cnt,
. . . . . . .> count(DISTINCT name) AS name_uni_cnt
. . . . . . .> FROM employee;
+--------------+---------------+
| sex_uni_cnt  | name_uni_cnt  |
+--------------+---------------+
| 2            | 5             |
+--------------+---------------+
1 row selected (35.935 seconds)
```

Note

When we use COUNT and DISTINCT together, Hive always ignores the setting (such as mapred.reduce.tasks = 20) for the number of reducers used and uses only one reducer. In this case, the single reducer becomes the bottleneck when processing big volumes of data. The workaround is to use the subquery as follows:

```
--Trigger single reducer during the whole processing
SELECT count(distinct sex_age.sex) AS sex_uni_cnt FROM employee;
--Use subquery to select unique value before aggregations for better
performance
SELECT count(*) AS sex_uni_cnt FROM (SELECT distinct sex_age.sex FROM
employee) a;
```

In this case, the first stage of the query implementing DISTINCT can use more than one reducer. In the second stage, the mapper will have less output just for the COUNT purpose since the data is already unique after implementing DISTINCT. As a result, the reducer will not be overloaded.

We may encounter a very special behavior when Hive deals with aggregation across columns with a NULL value. The entire row (if one column has NULL as a value in the row) will be ignored in the second row of the following example. To avoid this, we can use COALESCE to assign a default value when the column value is NULL. This can be done as follows:

```
--Create a table t for testing
jdbc:hive2://> CREATE TABLE t AS SELECT * FROM
. . . . . . .> (SELECT employee_id-99 AS val1,
. . . . . . .> (employee_id-98) AS val2 FROM employee_hr
. . . . . . .> WHERE employee_id <= 101
. . . . . . .> UNION ALL
. . . . . . .> SELECT null val1, 2 AS val2 FROM employee_hr
. . . . . . .> WHERE employee_id = 100) a;
No rows affected (0.138 seconds)

--Check the rows in the table created
jdbc:hive2://> SELECT * FROM t;
+---------+---------+
```

```
| t.val1  | t.val2  |
+---------+---------+
| 1       | 2       |
| NULL    | 2       |
| 2       | 3       |
+---------+---------+
3 rows selected (0.069 seconds)

--The 2nd row (NULL, 2) is ignored when doing sum(val1+val2)
jdbc:hive2://> SELECT sum(val1), sum(val1+val2)
. . . . . . .> FROM t;
+------+------+
| _c0  | _c1  |
+------+------+
| 3    | 8    |
+------+------+
1 row selected (57.775 seconds)

jdbc:hive2://> SELECT sum(coalesce(val1,0)),
. . . . . . .> sum(coalesce(val1,0)+val2) FROM t;
+------+------+
| _c0  | _c1  |
+------+------+
| 3    | 10   |
+------+------+
1 row selected (69.967 seconds)
```

The `hive.map.aggr` property controls aggregations in the `map` task. The default value for this setting is `false`. If it is set to `true`, Hive will do the first-level aggregation directly in the `map` task for better performance, but consume more memory:

```
jdbc:hive2://> SET hive.map.aggr=true;
No rows affected (0.002 seconds)
```

Advanced aggregation – GROUPING SETS

Hive has offered the `GROUPING SETS` keywords to implement advanced multiple `GROUP BY` operations against the same set of data. Actually, `GROUPING SETS` is a shorthand way of connecting several `GROUP BY` result sets with `UNION ALL`. The `GROUPING SETS` keyword completes all processes in one stage of jobs, which is more efficient than `GROUP BY` and `UNION ALL` having multiple stages. A blank set `()` in the `GROUPING SETS` clause calculates the overall aggregation. The following are a few examples to show the equivalence of `GROUPING SETS`. For better understanding, we can say that the outer level of `GROUPING SETS` defines on what data `UNION ALL` is to be implemented. The inner level defines on what data `GROUP BY` is to be implemented in each `UNION ALL`.

```
SELECT name, work_place[0] AS main_place,
count(employee_id) AS emp_id_cnt
FROM employee_id
GROUP BY name, work_place[0] GROUPING SETS((name, work_place[0]));
||
SELECT name, work_place[0] AS main_place,
count(employee_id) AS emp_id_cnt
FROM employee_id
GROUP BY name, work_place[0]

SELECT name, work_place[0] AS main_place,
count(employee_id) AS emp_id_cnt
FROM employee_id
GROUP BY name, work_place[0] GROUPING SETS(name, work_place[0]);
||
SELECT name, NULL AS main_place, count(employee_id) AS emp_id_cnt
FROM employee_id
GROUP BY name
UNION ALL
SELECT NULL AS name, work_place[0] AS main_place,
count(employee_id) AS emp_id_cnt
FROM employee_id
GROUP BY work_place[0];

SELECT name, work_place[0] AS main_place,
count(employee_id) AS emp_id_cnt
FROM employee_id
GROUP BY name, work_place[0]
GROUPING SETS((name, work_place[0]), name);
||
SELECT name, work_place[0] AS main_place,
count(employee_id) AS emp_id_cnt
FROM employee_id
GROUP BY name, work_place[0]
UNION ALL
SELECT name, NULL AS main_place, count(employee_id) AS emp_id_cnt
FROM employee_id
GROUP BY name;
```

```
SELECT name, work_place[0] AS main_place,
count(employee_id) AS emp_id_cnt
FROM employee_id
GROUP BY name, work_place[0]
GROUPING SETS((name, work_place[0]), name, work_place[0], ());
||
SELECT name, work_place[0] AS main_place,
count(employee_id) AS emp_id_cnt
FROM employee_id
GROUP BY name, work_place[0]
UNION ALL
SELECT name, NULL AS main_place, count(employee_id) AS emp_id_cnt
FROM employee_id
GROUP BY name
UNION ALL
SELECT NULL AS name, work_place[0] AS main_place,
count(employee_id) AS emp_id_cnt
FROM employee_id
GROUP BY work_place[0]
UNION ALL
SELECT NULL AS name, NULL AS main_place,
count(employee_id) AS emp_id_cnt
FROM employee_id;
```

However, the GROUPING SETS operation still has unresolved issues when working with columns referred by a table or record type alias (see Apache Jira HIVE-6950 at https://issues.apache.org/jira/browse/HIVE-6950). This is shown here:

```
jdbc:hive2://> SELECT sex_age.sex, sex_age.age,
. . . . . . .> count(name) AS name_cnt
. . . . . . .> FROM employee
. . . . . . .> GROUP BY sex_age.sex, sex_age.age
. . . . . . .> GROUPING SETS((sex_age.sex, sex_age.age));
Error: Error while compiling statement: FAILED: ParseException line 1:131
missing ) at ',' near '<EOF>'
line 1:145 extraneous input ')' expecting EOF near '<EOF>'
(state=42000,code=40000)
```

Advanced aggregation – ROLLUP and CUBE

The ROLLUP statement enables a SELECT statement to calculate multiple levels of aggregations across a specified group of dimensions. The ROLLUP statement is a simple extension to the GROUP BY clause with high efficiency and minimal overhead to a query. Compared to GROUPING SETS that creates specified levels of aggregations, ROLLUP creates $n+1$ levels of aggregations, where n is the number of grouping columns. First, it calculates the standard aggregate values specified in the GROUP BY clause. Then, it creates higher-level subtotals, moving from right to left through the list of combinations of grouping columns, as shown in the following example:

GROUP BY a,b,c WITH ROLLUP

This is equivalent to the following:

GROUP BY a,b,c GROUPING SETS ((a,b,c),(a,b),(a),())

The CUBE statement takes a specified set of grouping columns and creates aggregations for all of their possible combinations. If n columns are specified for CUBE, there will be 2^n combinations of aggregations returned, as shown in the following example:

GROUP BY a,b,c WITH CUBE

This is equivalent to the following:

GROUP BY a,b,c GROUPING SETS ((a,b,c),(a,b),(b,c),(a,c),(a),(b),(c),())

The GROUPING__ID function works as an extension to distinguish entire rows from each other. It accepts one or more columns and returns the decimal equivalent of the BIT vector for each column specified after GROUP BY. The returned decimal number is converted from a binary of 1s and 0s, which represents whether the column is aggregated (value is not NULL) in the row. The order of columns starts from counting the nearest column from GROUP BY. In the following example, the first column is start_date:

```
jdbc:hive2://> SELECT GROUPING__ID,
. . . . . . .> BIN(CAST(GROUPING__ID AS BIGINT)) AS bit_vector,
. . . . . . .> name, start_date, count(employee_id) emp_id_cnt
. . . . . . .> FROM employee_hr
. . . . . . .> GROUP BY start_date, name
. . . . . . .> WITH CUBE ORDER BY start_date;
```

grouping__id	bit_vector	name	start_date	emp_id_cnt
2	10	Steven	NULL	1
2	10	Michael	NULL	1
2	10	Lucy	NULL	1
0	0	NULL	NULL	4
2	10	Will	NULL	1
3	11	Lucy	2010-01-03	1
1	1	NULL	2010-01-03	1

```
| 1              | 1           | NULL     | 2012-11-03  | 1          |
| 3              | 11          | Steven   | 2012-11-03  | 1          |
| 1              | 1           | NULL     | 2013-10-02  | 1          |
| 3              | 11          | Will     | 2013-10-02  | 1          |
| 1              | 1           | NULL     | 2014-01-29  | 1          |
| 3              | 11          | Michael  | 2014-01-29  | 1          |
+---------------+------------+---------+------------+-----------+
13 rows selected (136.708 seconds)
```

Aggregation condition – HAVING

Since Hive 0.7.0, HAVING is added to support the conditional filtering of GROUP BY results. By using HAVING, we can avoid using a subquery after GROUP BY. The following is an example:

```
jdbc:hive2://> SELECT sex_age.age FROM employee
. . . . . . .> GROUP BY sex_age.age HAVING count(*)<=1;
+--------------+
| sex_age.age  |
+--------------+
| 57           |
| 27           |
| 35           |
+--------------+
3 rows selected (74.376 seconds)
```

If we do not use HAVING, we can use a subquery for instance as follows:

```
jdbc:hive2://> SELECT a.age
. . . . . . .> FROM
. . . . . . .> (SELECT count(*) as cnt, sex_age.age
. . . . . . .> FROM employee GROUP BY sex_age.age
. . . . . . .> ) a WHERE a.cnt<=1;
+--------+
| a.age  |
+--------+
| 57     |
| 27     |
| 35     |
+--------+
3 rows selected (87.298 seconds)
```

Analytic functions

Analytic functions, available since Hive 0.11.0, are a special group of functions that scan the multiple input rows to compute each output value. Analytic functions are usually used with `OVER`, `PARTITION BY`, `ORDER BY`, and the windowing specification. Different from the regular aggregate functions used with the `GROUP BY` clause that is limited to one result value per group, analytic functions operate on windows where the input rows are ordered and grouped using flexible conditions expressed through an `OVER PARTITION` clause. Though analytic functions give aggregate results, they do not group the result set. They return the group value multiple times with each record. The analytic functions offer great flexibility and functionalities than the regular `GROUP BY` clause and make special aggregations in Hive easier and powerful. The syntax for the analyze function is as follows:

```
Function (arg1,..., argn) OVER ([PARTITION BY <...>] [ORDER BY <....>]
[<window_clause>])
```

The `Function (arg1,..., argn)` can be any function in the following list with examples:

- **Standard aggregations**: This can be either `COUNT()`, `SUM()`, `MIN()`, `MAX()`, or `AVG()`.
- `RANK`: It ranks items in a group, such as finding the top *N* rows for specific conditions.
- `DENSE_RANK`: It is similar to `RANK`, but leaves no gaps in the ranking sequence when there are ties. For example, if we rank a match using `DENSE_RANK` and had two players tie for second place, we would see that the two players were in second place and that the next person is ranked as third. However, the `RANK` function would also rank two people in second place, but the next person would be in fourth place.
- `ROW_NUMBER`: It assigns a unique sequence number starting from 1 to each row according to the partition and order specification.
- `CUME_DIST`: It computes the number of rows whose value is smaller or equal to the value of the total number of rows divided by the current row.
- `PERCENT_RANK`: It is similar to `CUME_DIST`, but it uses rank values rather than row counts in its numerator as *total number of rows - 1* divided by *current rank - 1*. Therefore, it returns the percent rank of a value relative to a group of values.
- `NTILE`: It divides an ordered dataset into number of buckets and assigns an appropriate bucket number to each row. It can be used to divide rows into equal sets and assign a number to each row.
- `LEAD`: The `LEAD` function, `lead(value_expr[,offset[,default]])`, is used to return data from the next row. The number (`value_expr`) of rows to lead can optionally be specified. If the number of rows (`offset`) to lead is not specified, the lead is one row by default. It returns `[,default]` or null when the default is not specified and the lead for the current row extends beyond the end of the window.
- `LAG`: The `LAG` function, `lag(value_expr[,offset[,default]])`, is used to access data from a previous row. The number (`value_expr`) of rows to lag can optionally be specified. If the number of rows (`offset`) to lag is not specified, the lag is one row by default. It returns `[,default]` or null when the default is not specified and the lag for the current row extends beyond the end of the window.

- `FIRST_VALUE`: It returns the first result from an ordered set.
- `LAST_VALUE`: It returns the last result from an ordered set. For `LAST_VALUE`, using the default windowing clause, the result can be a little unexpected. This is because the default windowing clause is `RANGE BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW`, which in this example means the current row will always be the last value. Changing the windowing clause to `RANGE BETWEEN UNBOUNDED PRECEDING AND UNBOUNDED FOLLOWING` gives us the result we probably expected (see the `last_value` column in the following examples).

The [`PARTITION BY <...>`] statement is similar to the `GROUP BY` clause. It divides the rows into groups containing identical values in one or more partitions by columns. These logical groups are known as partitions, which is not the same term used for partition tables. Omitting the `PARTITION BY` statement applies the analytic operation to all the rows in the table.

The [`ORDER BY <....>`] clause is like the `ORDER BY expr [ASC|DESC]` clause. The `ORDER BY` clause is the same as the regular `ORDER BY` clause. It makes sure the rows produced by the `PARTITION BY` clause are ordered by specifications, such as ascending or descending order. Right now, Hive only supports one `ORDER BY` column in this case. Otherwise, it will throw a semantic exception (see Apache Jira HIVE-4662 at https://issues.apache.org/jira/browse/HIVE-4662). The workaround is to use the `rows unbounded preceding` windowing clause (see `runningTotal2` column in the following examples):

- Prepare the table and data for demonstration:

```
jdbc:hive2://> CREATE TABLE IF NOT EXISTS employee_contract
. . . . . . .> (
. . . . . . .> name string,
. . . . . . .> dept_num int,
. . . . . . .> employee_id int,
. . . . . . .> salary int,
. . . . . . .> type string,
. . . . . . .> start_date date
. . . . . . .> )
. . . . . . .> ROW FORMAT DELIMITED
. . . . . . .> FIELDS TERMINATED BY '|'
. . . . . . .> STORED AS TEXTFILE;
No rows affected (0.282 seconds)

jdbc:hive2://> LOAD DATA LOCAL INPATH
. . . . . . .> '/home/dayongd/Downloads/employee_contract.txt'
. . . . . . .> OVERWRITE INTO TABLE employee_contract;
No rows affected (0.48 seconds)
```

- The regular aggregations are used as analytic functions, as follows:

```
jdbc:hive2://> SELECT name, dept_num, salary,
. . . . . . .> COUNT(*) OVER (PARTITION BY dept_num) AS row_cnt,
. . . . . . .> SUM(salary) OVER(PARTITION BY dept_num
. . . . . . .> ORDER BY dept_num) AS deptTotal,
. . . . . . .> SUM(salary) OVER(ORDER BY dept_num)
```

```
. . . . . . .> AS runningTotal1, SUM(salary)
. . . . . . .> OVER(ORDER BY dept_num, name rows unbounded
. . . . . . .> preceding) AS runningTotal2
. . . . . . .> FROM employee_contract
. . . . . . .> ORDER BY dept_num, name;
```

name	dept_num	salary	row_cnt	deptTotal	runningTotal1	runningTotal2
Lucy	1000	5500	5	24900	24900	5500
Michael	1000	5000	5	24900	24900	10500
Steven	1000	6400	5	24900	24900	16900
Will	1000	4000	5	24900	24900	24900
Will	1000	4000	5	24900	24900	20900
Jess	1001	6000	3	17400	42300	30900
Lily	1001	5000	3	17400	42300	35900
Mike	1001	6400	3	17400	42300	42300
Richard	1002	8000	3	20500	62800	50300
Wei	1002	7000	3	20500	62800	57300
Yun	1002	5500	3	20500	62800	62800

```
11 rows selected (359.918 seconds)
```

- Other analytic functions are used as follows:

```
jdbc:hive2://> SELECT name, dept_num, salary,
. . . . . . .> RANK() OVER (PARTITION BY dept_num ORDER BY salary)
. . . . . . .> AS rank,
. . . . . . .> DENSE_RANK()
. . . . . . .> OVER (PARTITION BY dept_num ORDER BY salary)
. . . . . . .> AS dense_rank, ROW_NUMBER() OVER () AS row_num,
. . . . . . .> ROUND((CUME_DIST() OVER (PARTITION BY dept_num
. . . . . . .> ORDER BY salary)), 1) AS cume_dist,
. . . . . . .> PERCENT_RANK() OVER(PARTITION BY dept_num
. . . . . . .> ORDER BY salary) AS percent_rank, NTILE(4)
. . . . . . .> OVER(PARTITION BY dept_num ORDER BY salary)
. . . . . . .> AS ntile
. . . . . . .> FROM employee_contract ORDER BY dept_num;
```

name	dept_num	salary	rank	dense_rank	row_num	cume_dist	percent_rank	ntile
Will	1000	4000	1	1	11	0.4	0.0	1
Will	1000	4000	1	1	10	0.4	0.0	1
Michael	1000	5000	3	2	9	0.6	0.5	2
Lucy	1000	5500	4	3	8	0.8	0.75	3
Steven	1000	6400	5	4	7	1.0	1.0	4
Lily	1001	5000	1	1	6	0.3	0.0	1
Jess	1001	6000	2	2	5	0.7	0.5	

```
| 2     |
|Mike    | 1001    | 6400 | 3 | 3           | 4       | 1.0        | 1.0
| 3     |
|Yun     | 1002    | 5500 | 1 | 1           | 3       | 0.3        | 0.0
| 1     |
|Wei     | 1002    | 7000 | 2 | 2           | 2       | 0.7        | 0.5
| 2     |
|Richard| 1002    | 8000 | 3 | 3           | 1       | 1.0        | 1.0
| 3     |
+-------+--------+------+----+----------+-------+--------+-----------
+-----+
11 rows selected (367.112 seconds)

jdbc:hive2://> SELECT name, dept_num, salary,
. . . . . . .> LEAD(salary, 2) OVER(PARTITION BY dept_num
. . . . . . .> ORDER BY salary) AS lead,
. . . . . . .> LAG(salary, 2, 0) OVER(PARTITION BY dept_num
. . . . . . .> ORDER BY salary) AS lag,
. . . . . . .> FIRST_VALUE(salary) OVER (PARTITION BY dept_num
. . . . . . .> ORDER BY salary) AS first_value,
. . . . . . .> LAST_VALUE(salary) OVER (PARTITION BY dept_num
. . . . . . .> ORDER BY salary) AS last_value_default,
. . . . . . .> LAST_VALUE(salary) OVER (PARTITION BY dept_num
. . . . . . .> ORDER BY salary
. . . . . . .> RANGE BETWEEN UNBOUNDED PRECEDING
. . . . . . .> AND UNBOUNDED FOLLOWING) AS last_value
. . . . . . .> FROM employee_contract ORDER BY dept_num;
+-------+--------+------+----+----+----------+----------------+-----
----+
| name  |dept_num|salary|lead|lag |first_value|last_value_default|
last_value
|
+-------+--------+------+----+----+----------+----------------+-----
----+
|Will   |1000    |4000  |5000|0   |4000      |4000            |6400
|
|Will   |1000    |4000  |5500|0   |4000      |4000            |6400
|
|Michael|1000    |5000  |6400|4000|4000      |5000            |6400
|
|Lucy   |1000    |5500  |NULL|4000|4000      |5500            |6400
|
|Steven |1000    |6400  |NULL|5000|4000      |6400            |6400
|
|Lily   |1001    |5000  |6400|0   |5000      |5000            |6400
|
|Jess   |1001    |6000  |NULL|0   |5000      |6000            |6400
|
|Mike   |1001    |6400  |NULL|5000|5000      |6400            |6400
|
|Yun    |1002    |5500  |8000|0   |5500      |5500            |8000
|
|Wei    |1002    |7000  |NULL|0   |5500      |7000            |8000
|
|Richard|1002    |8000  |NULL|5500|5500      |8000            |8000
|
```

```
+-------+-------+------+----+----+----------+----------------+-----
----+
```
11 rows selected (92.572 seconds)

The [`<window_clause>`] clause is used to further subpartition the result and apply the analytic functions. There are two types of windows: row type window and range type window.

Note

According to the article at https://issues.apache.org/jira/browse/HIVE-4797, the `RANK`, `NTILE`, `DENSE_RANK`, `CUME_DIST`, `PERCENT_RANK`, `LEAD`, `LAG`, and `ROW_NUMBER` functions do not support being used with a window clause yet.

For row type windows, the definition is in terms of row numbers before or after the current row. The general syntax of the row window clause is as follows:

`ROWS BETWEEN <start_expr> AND <end_expr>`

The `<start_expr>` can be any one of the following:

- `UNBOUNDED PRECEDING`
- `CURRENT ROW`
- `N PRECEDING` or `FOLLOWING`

The `<end_expr>` can be any one of the following:

- `UNBOUNDED FOLLOWING`
- `CURRENT ROW`
- `N PRECEDING` or `FOLLOWING`

The following are the window expressions:

- `BETWEEN … AND`: Use the `BETWEEN…AND` clause to specify the start point and end point for the window. The first expression (before `AND`) defines the start point and the second expression (after `AND`) defines the end point. If we omit `BETWEEN…AND` (such as `ROWS N PRECEDING` or `ROWS UNBOUNDED PRECEDING`), Hive considers it as the start point, and the end point defaults to the current row (see `win13` column in the upcoming examples).
- `N PRECEDING` or `FOLLOWING`: This indicates `N` rows before or after the current row.
- `UNBOUNDED PRECEDING`: This indicates that the window starts at the first row of the partition. This is the start point specification and cannot be used as an end point specification.
- `UNBOUNDED FOLLOWING`: This indicates that the window ends at the last row of the partition. This is the end point specification and cannot be used as a start point specification.
- `UNBOUNDED PRECEDING AND UNBOUNDED FOLLOWING`: This indicates the first and last row for every row, meaning all rows in the table (see `win12` column in the upcoming examples).
- `CURRENT ROW`: As a start point, `CURRENT ROW` specifies that the window begins at the current row or value depending on whether we have specified `ROW` or `RANGE` (RANGE

is introduced later in this chapter). In this case, the end point cannot be `N PRECEDING`. As an end point, `CURRENT ROW` specifies that the window ends at the current row or value depending on whether we have specified `ROW` or `RANGE`. In this case, the start point cannot be `N FOLLOWING`.

The following is a diagram that can help us understand the preceding definitions more clearly:
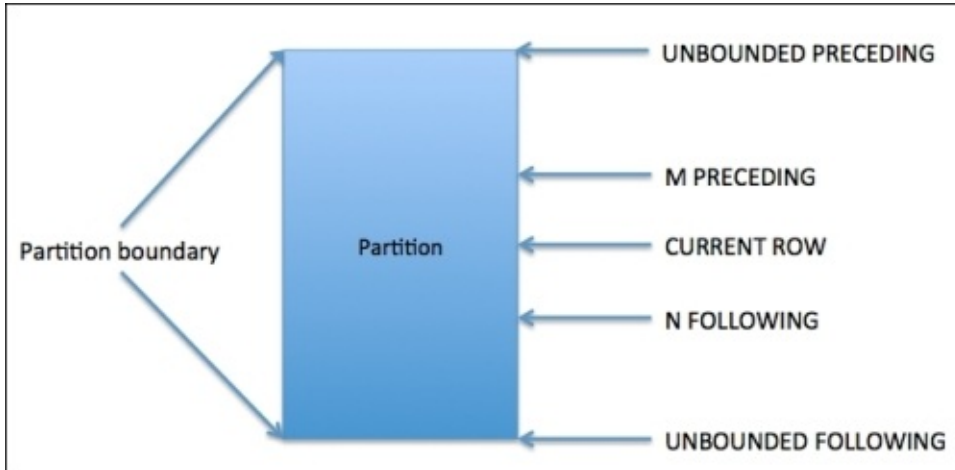
Window expression definition

The following examples implement the window expressions:

```
jdbc:hive2://> SELECT name, dept_num AS dept, salary AS sal,
. . . . . . .> MAX(salary) OVER (PARTITION BY dept_num ORDER BY
. . . . . . .> name ROWS
. . . . . . .> BETWEEN 2 PRECEDING AND CURRENT ROW) win1,
. . . . . . .> MAX(salary) OVER (PARTITION BY dept_num ORDER BY
. . . . . . .> name ROWS
. . . . . . .> BETWEEN 2 PRECEDING AND UNBOUNDED FOLLOWING) win2,
. . . . . . .> MAX(salary) OVER (PARTITION BY dept_num ORDER BY
. . . . . . .> name ROWS
. . . . . . .> BETWEEN 1 PRECEDING AND 2 FOLLOWING) win3,
. . . . . . .> MAX(salary) OVER (PARTITION BY dept_num ORDER BY
. . . . . . .> name ROWS
. . . . . . .> BETWEEN 1 PRECEDING AND 2 PRECEDING) win4,
. . . . . . .> MAX(salary) OVER (PARTITION BY dept_num ORDER BY
. . . . . . .> name ROWS
. . . . . . .> BETWEEN 1 FOLLOWING AND 2 FOLLOWING) win5,
. . . . . . .> MAX(salary) OVER (PARTITION BY dept_num ORDER BY
. . . . . . .> name ROWS
. . . . . . .> BETWEEN CURRENT ROW AND CURRENT ROW) win7,
. . . . . . .> MAX(salary) OVER (PARTITION BY dept_num ORDER BY
. . . . . . .> name ROWS
. . . . . . .> BETWEEN CURRENT ROW AND 1 FOLLOWING) win8,
. . . . . . .> MAX(salary) OVER (PARTITION BY dept_num ORDER BY
. . . . . . .> name ROWS
. . . . . . .> BETWEEN CURRENT ROW AND UNBOUNDED FOLLOWING) win9,
. . . . . . .> MAX(salary) OVER (PARTITION BY dept_num ORDER BY
. . . . . . .> name ROWS
```

```
. . . . . . .> BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW) win10,
. . . . . . .> MAX(salary) OVER (PARTITION BY dept_num ORDER BY
. . . . . . .> name ROWS
. . . . . . .> BETWEEN UNBOUNDED PRECEDING AND 1 FOLLOWING) win11,
. . . . . . .> MAX(salary) OVER (PARTITION BY dept_num ORDER BY
. . . . . . .> name ROWS BETWEEN UNBOUNDED PRECEDING AND UNBOUNDED
. . . . . . .> FOLLOWING) win12,
. . . . . . .> MAX(salary) OVER (PARTITION BY dept_num ORDER BY
. . . . . . .> name ROWS 2 PRECEDING) win13
. . . . . . .> FROM employee_contract
. . . . . . .> ORDER BY dept_num, name;
```

name	dept	sal	win1	win2	win3	win4	win5	win7	win8	win9	win10	win11	win12	win13
Lucy	1000	5500	5500	6400	6400	NULL	6400	5500	5500	6400	5500	5500	6400	5500
Michael	1000	5000	5500	6400	6400	NULL	6400	5000	6400	6400	5500	6400	6400	5500
Steven	1000	6400	6400	6400	6400	NULL	4000	6400	6400	6400	6400	6400	6400	6400
Will	1000	4000	6400	6400	4000	NULL	NULL	4000	4000	4000	6400	6400	6400	6400
Will	1000	4000	6400	6400	6400	NULL	4000	4000	4000	4000	6400	6400	6400	6400
Jess	1001	6000	6000	6400	6400	NULL	6400	6000	6000	6400	6000	6000	6400	6000
Lily	1001	5000	6000	6400	6400	NULL	6400	5000	6400	6400	6000	6400	6400	6000
Mike	1001	6400	6400	6400	6400	NULL	NULL	6400	6400	6400	6400	6400	6400	6400
Richard	1002	8000	8000	8000	8000	NULL	7000	8000	8000	8000	8000	8000	8000	8000
Wei	1002	7000	8000	8000	8000	NULL	5500	7000	7000	7000	8000	8000	8000	8000
Yun	1002	5500	8000	8000	7000	NULL	NULL	5500	5500	5500	8000	8000	8000	8000

```
11 rows selected (168.732 seconds)
```

From the preceding example, we can see that the win4 column is NULL. This is because the row specified by <start_expr> must be smaller than the row specified by <end_expr>. However, if we try to fix it by reordering it, especially when using the PRECEDING keyword, it reports the following exceptions and the same thing applies to UNBOUNDED PRECEDING. This is an issue (https://issues.apache.org/jira/browse/HIVE-9412) for Hive windowing right now:

```
jdbc:hive2://> SELECT name, dept_num, salary,
. . . . . . .> MAX(salary) OVER (PARTITION BY dept_num ORDER BY
. . . . . . .> name ROWS
. . . . . . .> BETWEEN 2 PRECEDING AND 1 PRECEDING) win4_alter
. . . . . . .> FROM employee_contract
```

```
. . . . . . .> ORDER BY dept_num, name;
Error: Error while compiling statement: FAILED: SemanticException Failed to
breakup Windowing invocations into Groups. At least 1 group must only
depend on input columns. Also check for circular dependencies.
Underlying error: Window range invalid, start boundary is greater than end
boundary: window(start=range(2 PRECEDING), end=range(1 PRECEDING))
(state=42000,code=40000)

jdbc:hive2://> SELECT name, dept_num, salary,
. . . . . . .> MAX(salary) OVER (PARTITION BY dept_num ORDER BY
. . . . . . .> name ROWS
. . . . . . .> BETWEEN UNBOUNDED PRECEDING AND 1 PRECEDING) win1
. . . . . . .> FROM employee_contract
. . . . . . .> ORDER BY dept_num, name;
Error: Error while compiling statement: FAILED: SemanticException End of a
WindowFrame cannot be UNBOUNDED PRECEDING (state=42000,code=40000)
```

In addition, windows can be defined in a separate WINDOW clause or referred by other windows, as follows:

```
jdbc:hive2://> SELECT name, dept_num, salary,
. . . . . . .> MAX(salary) OVER w1 AS win1,
. . . . . . .> MAX(salary) OVER w1 AS win2,
. . . . . . .> MAX(salary) OVER w1 AS win3
. . . . . . .> FROM employee_contract
. . . . . . .> ORDER BY dept_num, name
. . . . . . .> WINDOW
. . . . . . .> w1 AS (PARTITION BY dept_num ORDER BY name
. . . . . . .> ROWS BETWEEN 2 PRECEDING AND CURRENT ROW),
. . . . . . .> w2 AS w3,
. . . . . . .> w3 AS (PARTITION BY dept_num ORDER BY name
. . . . . . .> ROWS BETWEEN 1 PRECEDING AND 2 FOLLOWING);
+----------+-----------+---------+-------+-------+-------+
|   name   | dept_num  | salary  | win1  | win2  | win3  |
+----------+-----------+---------+-------+-------+-------+
| Lucy     | 1000      | 5500    | 5500  | 5500  | 5500  |
| Michael  | 1000      | 5000    | 5500  | 5500  | 5500  |
| Steven   | 1000      | 6400    | 6400  | 6400  | 6400  |
| Will     | 1000      | 4000    | 6400  | 6400  | 6400  |
| Will     | 1000      | 4000    | 6400  | 6400  | 6400  |
| Jess     | 1001      | 6000    | 6000  | 6000  | 6000  |
| Lily     | 1001      | 5000    | 6000  | 6000  | 6000  |
| Mike     | 1001      | 6400    | 6400  | 6400  | 6400  |
| Richard  | 1002      | 8000    | 8000  | 8000  | 8000  |
| Wei      | 1002      | 7000    | 8000  | 8000  | 8000  |
| Yun      | 1002      | 5500    | 8000  | 8000  | 8000  |
+----------+-----------+---------+-------+-------+-------+
11 rows selected (156.902 seconds)
```

Compared to row type windows in terms of rows, the range type windows are in terms of values before or after the current ORDER BY column, which must be a number or date type. For now, only one ORDER BY column is supported by range type windows.

```
jdbc:hive2://> SELECT name, salary, start_year,
. . . . . . .> MAX(salary) OVER (PARTITION BY dept_num ORDER BY
. . . . . . .> start_year RANGE
```

```
. . . . . . .> BETWEEN 2 PRECEDING AND CURRENT ROW) win1
. . . . . . .> FROM
. . . . . . .> (
. . . . . . .>   SELECT name, salary, dept_num,
. . . . . . .>   YEAR(start_date) AS start_year
. . . . . . .>   FROM employee_contract
. . . . . . .> ) a;
+----------+--------+------------+-------+
|   name   | salary | start_year | win1  |
+----------+--------+------------+-------+
| Lucy     | 5500   | 2010       | 5500  |
| Steven   | 6400   | 2012       | 6400  |
| Will     | 4000   | 2013       | 6400  |
| Will     | 4000   | 2014       | 6400  |
| Michael  | 5000   | 2014       | 6400  |
| Mike     | 6400   | 2013       | 6400  |
| Jess     | 6000   | 2014       | 6400  |
| Lily     | 5000   | 2014       | 6400  |
| Wei      | 7000   | 2010       | 7000  |
| Richard  | 8000   | 2013       | 8000  |
| Yun      | 5500   | 2014       | 8000  |
+----------+--------+------------+-------+
11 rows selected (92.035 seconds)
```

Note

If we omit the windowing clause entirely, the default window is `RANGE BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW`.

Sampling

When data volume is extra large, we may need to find a subset of data to speed up data analysis. Here it comes to a technique used to select and analyze a subset of data in order to identify patterns and trends. In Hive, there are three ways of sampling data: random sampling, bucket table sampling, and block sampling.

Random sampling uses the `RAND()` function and `LIMIT` keyword to get the sampling of data as shown in the following example. The `DISTRIBUTE` and `SORT` keywords are used here to make sure the data is also randomly distributed among mappers and reducers efficiently. The `ORDER BY RAND()` statement can also achieve the same purpose, but the performance is not good:

```
SELECT * FROM <Table_Name> DISTRIBUTE BY RAND() SORT BY RAND()
LIMIT <N rows to sample>;
```

Bucket table sampling is a special sampling optimized for bucket tables as shown in the following syntax and example. The `colname` value specifies the column where to sample the data. The `RAND()` function can also be used when sampling is on the entire rows. If the sample column is also the `CLUSTERED BY` column, the `TABLESAMPLE` statement will be more efficient.

```
--Syntax
SELECT * FROM <Table_Name>
TABLESAMPLE(BUCKET <specified bucket number to sample> OUT OF <total number
of buckets> ON [colname|RAND()]) table_alias;

--An example
jdbc:hive2://> SELECT name FROM employee_id_buckets
. . . . . . .> TABLESAMPLE(BUCKET 1 OUT OF 2 ON rand()) a;
+----------+
|   name   |
+----------+
| Lucy     |
| Shelley  |
| Lucy     |
| Lucy     |
| Shelley  |
| Lucy     |
| Will     |
| Shelley  |
| Michael  |
| Will     |
| Will     |
| Will     |
| Will     |
| Will     |
| Lucy     |
+----------+
15 rows selected (0.07 seconds)
```

Block sampling allows Hive to randomly pick up *N* rows of data, percentage (*n

percentage) of data size, or *N* byte size of data. The sampling granularity is the HDFS block size. Its syntax and examples are as follows:

```
--Syntax
SELECT *
FROM <Table_Name> TABLESAMPLE(N PERCENT|ByteLengthLiteral|N ROWS) s;

-- ByteLengthLiteral
-- (Digit)+ ('b' | 'B' | 'k' | 'K' | 'm' | 'M' | 'g' | 'G')

--Sample by rows
jdbc:hive2://> SELECT name
. . . . . . .> FROM employee_id_buckets TABLESAMPLE(4 ROWS) a;
+----------+
|   name   |
+----------+
| Lucy     |
| Shelley  |
| Lucy     |
| Shelley  |
+----------+
4 rows selected (0.055 seconds)

--Sample by percentage of data size
jdbc:hive2://> SELECT name
. . . . . . .> FROM employee_id_buckets TABLESAMPLE(10 PERCENT) a;
+----------+
|   name   |
+----------+
| Lucy     |
| Shelley  |
| Lucy     |
+----------+
3 rows selected (0.061 seconds)

--Sample by data size
jdbc:hive2://> SELECT name
. . . . . . .> FROM employee_id_buckets TABLESAMPLE(3M) a;
+----------+
|   name   |
+----------+
| Lucy     |
| Shelley  |
| Lucy     |
| Shelley  |
| Lucy     |
| Shelley  |
| Lucy     |
| Shelley  |
| Lucy     |
| Will     |
| Shelley  |
| Lucy     |
| Will     |
| Shelley  |
| Michael  |
```

```
| Will     |
| Shelley  |
| Lucy     |
| Will     |
| Will     |
| Will     |
| Will     |
| Will     |
| Lucy     |
| Shelley  |
+----------+
25 rows selected (0.07 seconds)
```

Summary

In this chapter, we covered how to aggregate data using basic aggregation functions. Then, we introduced the advanced aggregations with `GROUPING SETS`, `ROLLUP`, and `CUBE`, as well as aggregation conditions using `HAVING`. We also covered the various analytic functions and windowing clauses. At the end of the chapter, we introduced three ways of sampling data in Hive. After going through this chapter, you should be able to do basic and advanced aggregations and data sampling in Hive.

In the next chapter, we'll talk about performance considerations in Hive.

Chapter 7. Performance Considerations

Although Hive is built to deal with big data, we still cannot ignore the importance of performance. Most of the time, a better Hive query can rely on the smart query optimizer to find the best execution strategy as well as the default setting best practice from vendor packages. However, as experienced users, we should learn more about the theory and practice of performance tuning in Hive, especially when working in a performance-based project or environment. In this chapter, we will start from utilities available in Hive to find potential issues causing poor performance. Then, we introduce the best practices of performance considerations in the areas of design, file format, compression, storage, query, and job.

In this chapter, we will cover the following topics:

- Performance utilities
- Design optimization
- Data file optimization
- Job and query optimization

Performance utilities

Hive provides the `EXPLAIN` and `ANALYZE` statements that can be used as utilities to check and identify the performance of queries.

The EXPLAIN statement

Hive provides an `EXPLAIN` command to return a query execution plan without running the query. We can use an `EXPLAIN` command for queries if we have a doubt or a concern about performance. The `EXPLAIN` command will help to see the difference between two or more queries for the same purpose. The syntax for `EXPLAIN` is as follows:

```
EXPLAIN [EXTENDED|DEPENDENCY|AUTHORIZATION] hive_query
```

The following keywords can be used:

- `EXTENDED`: This provides additional information for the operators in the plan, such as file pathname and abstract syntax tree.
- `DEPENDENCY`: This provides a JSON format output that contains a list of tables and partitions that the query depends on. It is available since HIVE 0.10.0.
- `AUTHORIZATION`: This lists all entities needed to be authorized including input and output to run the Hive query and authorization failures, if any. It is available since HIVE 0.14.0.

A typical query plan contains the following three sections. We will also have a look at an example later:

- **Abstract syntax tree** (**AST**): Hive uses a pacer generator called ANTLR (see http://www.antlr.org/) to automatically generate a tree of syntax for HQL. We can usually ignore this most of the time.
- **Stage dependencies**: This lists all dependencies and number of stages used to run the query.
- **Stage plans**: It contains important information, such as operators and sort orders, for running the job.

The following is what a typical query plan looks like. From the following example, we can see that the `AST` section is not shown since the `EXTENDED` keyword is not used with `EXPLAIN`. In the `STAGE DEPENDENCIES` section, both `Stage-0` and `Stage-1` are independent root stages. In the `STAGE PLANS` section, `Stage-1` has one map and reduce referred to by `Map Operator Tree` and `Reduce Operator Tree`. Inside each `Map/Reduce Operator Tree` section, all operators corresponding to Hive query keywords as well as expressions and aggregations are listed. The `Stage-0` stage does not have map and reduce. It is just a `Fetch` operation.

```
jdbc:hive2://> EXPLAIN SELECT sex_age.sex, count(*)
. . . . . . .> FROM employee_partitioned
. . . . . . .> WHERE year=2014 GROUP BY sex_age.sex LIMIT 2;
+-------------------------------------------------------------------
---+
|                              Explain
|
+-------------------------------------------------------------------
---+
| STAGE DEPENDENCIES:
|
```

```
| Stage-1 is a root stage
|
| Stage-0 is a root stage
|
|
|
| STAGE PLANS:
|
|    Stage: Stage-1
|
|      Map Reduce
|
|        Map Operator Tree:
|
|            TableScan
|
|              alias: employee_partitioned
|
|              Statistics: Num rows: 0 Data size: 227 Basic stats:PARTIAL
|
|                        Column stats: NONE
|
|              Select Operator
|
|                expressions: sex_age (type: struct<sex:string,age:int>)
|
|                outputColumnNames: sex_age
|
|                Statistics: Num rows: 0 Data size: 227 Basic stats:PARTIAL
|
|                        Column stats: NONE
|
|                Group By Operator
|
|                  aggregations: count()
|
|                  keys: sex_age.sex (type: string)
|
|                  mode: hash
|
|                  outputColumnNames: _col0, _col1
|
|                  Statistics: Num rows: 0 Data size: 227 Basic
stats:PARTIAL  |
|                        Column stats: NONE
|
|                  Reduce Output Operator
|
|                    key expressions: _col0 (type: string)
|
|                    sort order: +
|
|                    Map-reduce partition columns: _col0 (type: string)
|
|                    Statistics: Num rows: 0 Data size: 227 Basic
stats:PARTIAL|
```

```
|                          Column stats: NONE
|
|               value expressions: _col1 (type: bigint)
|
|        Reduce Operator Tree:
|
|          Group By Operator
|
|            aggregations: count(VALUE._col0)
|
|            keys: KEY._col0 (type: string)
|
|            mode: mergepartial
|
|            outputColumnNames: _col0, _col1
|
|            Statistics: Num rows: 0 Data size: 0 Basic stats: NONE
|
|                       Column stats: NONE
|
|            Select Operator
|
|              expressions: _col0 (type: string), _col1 (type: bigint)
|
|              outputColumnNames: _col0, _col1
|
|              Statistics: Num rows: 0 Data size: 0 Basic stats: NONE
|
|                         Column stats: NONE
|
|              Limit
|
|                Number of rows: 2
|
|                Statistics: Num rows: 0 Data size: 0 Basic stats: NONE
|
|                           Column stats: NONE
|
|                File Output Operator
|
|                  compressed: false
|
|                  Statistics: Num rows: 0 Data size: 0 Basic stats: NONE
|
|                             Column stats: NONE
|
|                  table:
|
|                     input format:
org.apache.hadoop.mapred.TextInputFormat  |
|      output
format:org.apache.hadoop.hive.ql.io.HiveIgnoreKeyTextOutputFormat|
|
serde:org.apache.hadoop.hive.serde2.lazy.LazySimpleSerDe| |
|
|    Stage: Stage-0
```

```
|
|      Fetch Operator
|
|         limit: 2
|
+----------------------------------------------------------------------
---+
53 rows selected (0.26 seconds)
```

The ANALYZE statement

Hive statistics are a collection of data that describe more details, such as the number of rows, number of files, and raw data size, on the objects in the Hive database. Statistics is a metadata of Hive data. Hive supports statistics at the table, partition, and column level. These statistics serve as an input to the Hive **Cost-Based Optimizer** (**CBO**), which is an optimizer to pick the query plan with the lowest cost in terms of system resources required to complete the query.

The statistics are gathered through the ANALYZE statement since Hive 0.10.0 on tables, partitions, and columns as given in the following examples:

```
jdbc:hive2://> ANALYZE TABLE employee COMPUTE STATISTICS;
No rows affected (27.979 seconds)

jdbc:hive2://> ANALYZE TABLE employee_partitioned
. . . . . . .> PARTITION(year=2014, month=12) COMPUTE STATISTICS;
No rows affected (45.054 seconds)

jdbc:hive2://> ANALYZE TABLE employee_id COMPUTE STATISTICS
. . . . . . .> FOR COLUMNS employee_id;
No rows affected (41.074 seconds)
```

Once the statistics are built, we can check the statistics by the DESCRIBE EXTENDED/FORMATTED statement. From the table/partition output, we can find the statistics information inside the parameters, such as parameters:{numFiles=1, COLUMN_STATS_ACCURATE=true, transient_lastDdlTime=1417726247, numRows=4, totalSize=227, rawDataSize=223}). The following is an example:

```
jdbc:hive2://> DESCRIBE EXTENDED employee_partitioned
. . . . . . .> PARTITION(year=2014, month=12);

jdbc:hive2://> DESCRIBE EXTENDED employee;
…
parameters:{numFiles=1, COLUMN_STATS_ACCURATE=true,
transient_lastDdlTime=1417726247, numRows=4, totalSize=227,
rawDataSize=223}).

jdbc:hive2://> DESCRIBE FORMATTED employee.name;
+--------+---------+---+---+---------+--------------+-----------+----------
-+
|col_name|data_type|min|max|num_nulls|distinct_count|avg_col_len|max_col_le
n|
+--------+---------+---+---+---------+--------------+-----------+----------
-+
| name   | string  |   |   | 0       | 5            | 5.6       | 7
|
+--------+---------+---+---+---------+--------------+-----------+----------
-+
+---------+----------+----------------+
|num_trues|num_falses|    comment     |
+---------+----------+----------------+
|         |          |from deserializer|
```

```
+--------+---------+----------------+
3 rows selected (0.116 seconds)
```

Hive statistics are persisted in the metastore to avoid computing them every time. For newly created tables and/or partitions, statistics are automatically computed by default if we enable the following setting:

```
jdbc:hive2://> SET hive.stats.autogather=ture;
```

Note

Hive logs

Logs provide useful information to find out how a Hive query/job runs. By checking the Hive logs, we can identify runtime problems and issues that may cause bad performance. There are two types of logs available in Hive: system log and job log.

The system log contains the Hive running status and issues. It is configured in `{HIVE_HOME}/conf/hive-log4j.properties`. The following three lines for Hive log can be found:

```
hive.root.logger=WARN,DRFA
hive.log.dir=/tmp/${user.name}
hive.log.file=hive.log
```

To modify the status, we can either modify the preceding lines in `hive-log4j.properties` (applies to all users) or set from the Hive CLI (only applies to the current user and current session) as follows:

```
hive --hiveconf hive.root.logger=DEBUG,console
```

The job log contains Hive query information and is saved at the same place, `/tmp/${user.name}`, by default as one file for each Hive user session. We can override it in `hive-site.xml` with the `hive.querylog.location` property. If a Hive query generates MapReduce jobs, those logs can also be viewed through the Hadoop JobTracker Web UI.

Design optimization

Design optimization covers several data layout and design strategies to improve performance.

Partition tables

Hive partitioning is one of the most effective methods to improve the query performance on larger tables. The query with partition filtering will only load the data in the specified partitions (subdirectories), so it can execute much faster than a normal query that filters by a non-partitioning field. The selection of partition key is always an important factor for performance. It should always be a low cardinal attribute to avoid many subdirectories overhead.

The following are some commonly used dimensions as partition keys:

- **Partitions by date and time**: Use date and time, such as year, month, and day (even hours), as partition keys when data is associated with the time dimension
- **Partitions by locations**: Use country, territory, state, and city as partition keys when data is location related
- **Partitions by business logics**: Use department, sales region, applications, customers, and so on as partitioned keys when data can be separated evenly by some business logic

Bucket tables

Similar to partitioning, a bucket table organizes data into separate files in the HDFS. Bucketing can speed up the data sampling in Hive with sampling on buckets. Bucketing can also improve the join performance if the join keys are also bucket keys because bucketing ensures that the key is present in a certain bucket. More details are given in the *Job and Query optimization* section in this chapter.

Index

Index is very common with RDBMS when we want to speed access to a column or set of columns. Hive supports index creation on tables/partitions since Hive 0.7.0. The index in Hive provides key-based data view and better data access for certain operations, such as WHERE, GROUP BY, and JOIN. We can use index is a cheaper alternative than full table scans. The command to create an index in Hive is straightforward as follows:

```
jdbc:hive2://> CREATE INDEX idx_id_employee_id
. . . . . . .> ON TABLE employee_id (employee_id)
. . . . . . .> AS 'COMPACT'
. . . . . . .> WITH DEFERRED REBUILD;
No rows affected (1.149 seconds)
```

In addition to the COMPACT keyword (refers to org.apache.hadoop.hive.ql.index.compact.CompactIndexHandler) used in the preceding example, Hive also supports BITMAP indexes since HIVE 0.8.0 for columns with less different values, as shown in the following example:

```
jdbc:hive2://> CREATE INDEX idx_sex_employee_id
. . . . . . .> ON TABLE employee_id (sex_age)
. . . . . . .> AS 'BITMAP'
. . . . . . .> WITH DEFERRED REBUILD;
No rows affected (0.251 seconds)
```

The WITH DEFERRED REBUILD keyword in the preceding example prevents the index from immediately being built. To build the index, we can issue ALTER…REBUILD commands as in the following example. When data in the base table changes, the ALTER…REBUILD command must be used to bring the index up to date. This is an atomic operation, so if the index rebuilt on a table that has been previously indexed failed, the state of index remains the same, as shown here:

```
jdbc:hive2://> ALTER INDEX idx_id_employee_id ON employee_id REBUILD;
No rows affected (111.413 seconds)
```

```
jdbc:hive2://> ALTER INDEX idx_sex_employee_id ON employee_id
. . . . . . .> REBUILD;
No rows affected (82.23 seconds)
```

Once the index is built, Hive will create a new index table for each index as follows:

```
jdbc:hive2://> !table
+----------+-----------------------------------------+-----------+-------
+
|TABLE_SCHEM|                  TABLE_NAME             |
TABLE_TYPE|REMARKS|
+----------+-----------------------------------------+-----------+-------
+
|default    |default__employee_id_idx_id_employee_id__ |INDEX_TABLE|NULL
|
|default    |default__employee_id_idx_sex_employee_id__|INDEX_TABLE|NULL
|
+----------+-----------------------------------------+-----------+-------
```

+

The index table will have name convention such as `default__tablename_indexname__`. It contains the indexed column, the `_bucketname` (typical file URI on HDFS), and `_offsets` (offsets for each rows). Then, this index table can be used where we need to query the indexed columns like a regular table, as shown here:

```
jdbc:hive2://> DESC default__employee_id_idx_id_employee_id__;
+--------------+----------------+----------+
|   col_name   |   data_type    | comment  |
+--------------+----------------+----------+
| employee_id  | int            |          |
| _bucketname  | string         |          |
| _offsets     | array<bigint>  |          |
+--------------+----------------+----------+
3 rows selected (0.135 seconds)
```

To drop an index, we can use the `DROP INDEX index_name ON table_name` statement as follows. However, we cannot drop the index table with a `DROP TABLE` statement:

```
jdbc:hive2://> DROP INDEX idx_sex_employee_id ON employee_id;
No rows affected (0.247 seconds)
```

Note

Since Hive 0.13.0, Hive includes the following new features for performance optimizations:

- **Tez**: Tez (http://tez.apache.org/) is an application framework built on Yarn that can execute complex **directed acyclic graphs** (**DAGs**) for general data-processing tasks. Tez further splits map and reduce jobs into smaller tasks and combines them in a flexible and efficient way for execution. Tez is considered a flexible and powerful successor to the MapReduce framework. To configure Hive to use Tez, we need to overwrite the following settings from the default MapReduce:

  ```
  SET hive.execution.engine=tez;
  ```

- **Vectorization**: Vectorization optimization processes a larger batch of data at the same time rather than one row at a time, thus significantly reducing computing overhead. Each batch consists of a column vector that is usually an array of primitive types. Operations are performed on the entire column vector, which improves the instruction pipelines and cache usage. Files must be stored in the **Optimized Row Columnar** (**ORC**) format in order to use vectorization. For more on vectorization, please refer to the Apache Hive wiki at https://cwiki.apache.org/confluence/display/Hive/Vectorized+Query+Execution. To enable vectorization, we need to do the following setting:

  ```
  SET hive.vectorized.execution.enabled=true;
  ```

Data file optimization

Data file optimization covers the performance improvement on the data files in terms of file format, compression, and storage.

File format

Hive supports `TEXTFILE`, `SEQUENCEFILE`, `RCFILE`, `ORC`, and `PARQUET` file formats. The three ways to specify the file format are as follows:

- `CREATE TABLE… STORE AS <File_Format>`
- `ALTER TABLE… [PARTITION partition_spec] SET FILEFORMAT <File_Format>`
- `SET hive.default.fileformat=<File_Format> --default fileformat for table`

Here, `<File_Type>` is `TEXTFILE`, `SEQUENCEFILE`, `RCFILE`, `ORC`, and `PARQUET`.

We can load a text file directly to a table with the `TEXTFILE` format. To load data to the table with other file formats, we need to load the data to a `TEXTFILE` format table first. Then, use `INSERT OVERWRITE TABLE <target_file_format_table> SELECT * FROM <text_format_source_table>` to convert and insert the data to the file format as expected.

The file formats supported by Hive and their optimizations are as follows:

- `TEXTFILE`: This is the default file format for Hive. Data is not compressed in the text file. It can be compressed with compression tools, such as GZip, Bzip2, and Snappy. However, these compressed files are not splittable as input during processing. As a result, it leads to running a single, huge map job to process one big file.
- `SEQUENCEFILE`: This is a binary storage format for key/value pairs. The benefit of a sequence file is that it is more compact than a text file and fits well with the MapReduce output format. Sequence files can be compressed on record or block level where block level has a better compression ratio. To enable block level compression, we need to do the following settings:

  ```
  jdbc:hive2://> SET hive.exec.compress.output=true;
  jdbc:hive2://> SET io.seqfile.compression.type=BLOCK;
  ```

  Unfortunately, both text and sequence files as a row level storage file format are not an optimal solution since Hive has to read a full row even if only one column is being requested. For instance, a hybrid row-columnar storage file format, such as `RCFILE`, `ORC`, and `PARQUET` implementation, is created to resolve this problem.

- `RCFILE`: This is short for **Record Columnar File**. It is a flat file consisting of binary key/value pairs that shares much similarity with a sequence file. The RCFile splits data horizontally into row groups. One or several groups are stored in an HDFS file. Then, RCFile saves the row group data in a columnar format by saving the first column across all rows, then the second column across all rows, and so on. This format is splittable and allows Hive to skip irrelevant parts of data and get the results faster and cheaper.
- `ORC`: This is short for **Optimized Row Columnar**. It is available since Hive 0.11.0. The `ORC` format can be considered an improved version of `RCFILE`. It provides a larger block size of 256 MB by default (`RCFILE` has 4 MB and `SEQUENCEFILE` has 1 MB) optimized for large sequential reads on HDFS for more throughput and fewer files to

reduce overload in the namenode. Different from `RCFILE` that relies on metastore to know data types, the `ORC` file understands the data types by using specific encoders so that it can optimize compression depending on different types. It also stores basic statistics, such as `MIN`, `MAX`, `SUM`, and `COUNT`, on columns as well as a lightweight index that can be used to skip blocks of rows that do not matter.

- `PARQUET`: This is another row columnar file format that has a similar design to that of `ORC`. What's more, Parquet has a wider range of support for the majority projects in the Hadoop ecosystem compared to `ORC` that only supports Hive and Pig. Parquet leverages the design best practices of Google's Dremel (see http://research.google.com/pubs/pub36632.html) to support the nested structure of data. Parquet is supported by a plugin since Hive 0.10.0 and has got native support since Hive 0.13.0.

Considering the maturity of Hive, it is suggested to use the `ORC` format if Hive is the main majority tool used in your Hadoop environment. If you use several tools in the Hadoop ecosystem, `PARQUET` is a better choice in terms of adaptability.

Note

Hadoop Archive File (**HAR**) is another type of file format to pack HDFS files into archives. This is an option (not a good option) for storing a large number of small-sized files in HDFS, as storing a large number of small-sized files directly in HDFS is not very efficient. However, HAR still has some limitations that make it unpopular, such as immutable archive process, not being splittable, and compatibility issues. For more information about HAR and archiving, please refer to the Apache Hive wiki at https://cwiki.apache.org/confluence/display/Hive/LanguageManual+Archiving.

Compression

Compression techniques in Hive can significantly reduce the amount of data transferring between mappers and reducers by proper intermediate output compression as well as output data size in HDFS by output compression. As a result, the overall Hive query will have better performance. To compress intermediate files produced by Hive between multiple MapReduce jobs, we need to set the following property (`false` by default) in the Hive CLI or the `hive-site.xml` file:

```
jdbc:hive2://> SET hive.exec.compress.intermediate=true
```

Then, we need to decide which compression codec to configure. A list of common codecs supported in Hadoop and Hive is as follows:

Compression	Codec	Extension	Splittable
Deflate	org.apache.hadoop.io.compress.DefaultCodec	`.deflate`	N
GZip	org.apache.hadoop.io.compress.GzipCodec	`.gz`	N
Bzip2	org.apache.hadoop.io.compress.BZip2Codec	`.gz`	Y
LZO	com.hadoop.compression.lzo.LzopCodec	`.lzo`	N
LZ4	org.apache.hadoop.io.compress.Lz4Codec	`.lz4`	N
Snappy	org.apache.hadoop.io.compress.SnappyCodec	`.snappy`	N

Hadoop has a default codec (`.deflate`). The compression ratio for GZip is higher as well as its CPU cost. Bzip2 is splittable, but splitting isn't supported by Hadoop until 1.1 (see https://issues.apache.org/jira/browse/HADOOP-4012). In addition, Bzip2 is too slow for compression considering its huge CPU cost. LZO files are not natively splittable. But we can preprocess them (using `com.hadoop.compression.lzo.LzoIndexer`) to create an index that determines the file splits. When it comes to the balance of CPU cost and compression ratio, LZ4 or Snappy do a better job. Since the majority of codec do not support split after compression, it is suggested to avoid compressing big files in HDFS.

The compression codec can be specified in either `mapred-site.xml`, `hive-site.xml`, or Hive CLI, as in the following example:

```
jdbc:hive2://> SET hive.intermediate.compression.codec=
. . . . . . .> org.apache.hadoop.io.compress.SnappyCodec
```

Intermediate compression will only save disk space for specific jobs that require multiple map and reduce jobs. For further saving of disk space, the actual Hive output files can be compressed. When the `hive.exec.compress.output` property is set to `true`, Hive will use the codec configured by the `mapred.map.output.compression.codec` property to compress the storage in HDFS as follows. These properties can be set in the `hive-site.xml` or in the Hive CLI.

```
jdbc:hive2://> SET hive.exec.compress.output=true

jdbc:hive2://> SET mapred.output.compression.codec=
. . . . . . .> org.apache.hadoop.io.compress.SnappyCodec
```

Storage optimization

The data, which is used or scanned frequently, can be identified as hot data. Usually, the query performance on the hot data is critical for overall performance. Increasing the data replication factor in HDFS (see the following example) for hot data could increase the chance of data being hit locally by Hive jobs and improve the performance. However, this is a trade-off for storage.

```
$ hdfs dfs -setrep -R -w 4 /user/hive/warehouse/employee
Replication 4 set: /user/hive/warehouse/employee/000000_0
```

On the other hand, too many files or redundancy could make namenode's memory exhausted, especially for lots of small files less than the HDFS block sizes. Hadoop itself already has some solutions to deal with too many small-file issues, such as the following:

- **Hadoop Archive and HAR**: These are toolkits to pack small files.
- **SequenceFile format**: This is a format to compress small files to bigger files.
- **CombineFileInputFormat**: A type of `InputFormat` to combine small files before map and reduce processing. It is the default `InputFormat` for Hive (see https://issues.apache.org/jira/browse/HIVE-2245).
- **HDFS federation**: It makes namenodes extensible and powerful to manage more files.

We can also leverage other tools in the Hadoop ecosystem if we have them installed, such as the following:

- HBase has a smaller block size and better file format to deal with smaller-file access issues
- Flume NG can be used as pipes to merge small files to big ones
- A scheduled offline file merge program to merge small files in HDFS or before loading them to HDFS

For Hive, we can do the following configurations for merging files of query results to avoid recreating small files:

- `hive.merge.mapfiles`: This merges small files at the end of a map-only job. By default, it is `true`.
- `hive.merge.mapredfiles`: This merges small files at the end of a MapReduce job. Set it to `true` since its default is `false`.
- `hive.merge.size.per.task`: This defines the size of merged files at the end of the job. The default value is 256,000,000.
- `hive.merge.smallfiles.avgsize`: This is the threshold for triggering file merge. The default value is 16,000,000.

When the average output file size of a job is less than the value specified by `hive.merge.smallfiles.avgsize`, and both `hive.merge.mapfiles` (for map-only jobs) and `hive.merge.mapredfiles` (for MapReduce jobs) are set to `true`, Hive will start an additional MapReduce job to merge the output files into big files.

Job and query optimization

Job and query optimization covers experience and skills to improve performance in the area of job-running mode, JVM reuse, job parallel running, and query optimizations in `JOIN`.

Local mode

Hadoop can run in standalone, pseudo-distributed, and fully distributed mode. Most of the time, we need to configure Hadoop to run in fully distributed mode. When the data to process is small, it is an overhead to start distributed data processing since the launching time of the fully distributed mode takes more time than the job processing time. Since Hive 0.7.0, Hive supports automatic conversion of a job to run in local mode with the following settings:

```
jdbc:hive2://> SET hive.exec.mode.local.auto=true; --default false
jdbc:hive2://> SET hive.exec.mode.local.auto.inputbytes.max=50000000;
jdbc:hive2://> SET hive.exec.mode.local.auto.input.files.max=5;
--default 4
```

A job must satisfy the following conditions to run in the local mode:

- The total input size of the job is lower than
  `hive.exec.mode.local.auto.inputbytes.max`
- The total number of map tasks is less than
  `hive.exec.mode.local.auto.input.files.max`
- The total number of reduce tasks required is 1 or 0

JVM reuse

By default, Hadoop launches a new JVM for each map or reduce job and runs the map or reduce task in parallel. When the map or reduce job is a lightweight job running only for a few seconds, the JVM startup process could be a significant overhead. The MapReduce framework (version 1 only, not Yarn) has an option to reuse JVM by sharing the JVM to run mapper/reducer serially instead of parallel. JVM reuse applies to map or reduce tasks in the same job. Tasks from different jobs will always run in a separate JVM. To enable the reuse, we can set the maximum number of tasks for a single job for JVM reuse using the `mapred.job.reuse.jvm.num.tasks` property. Its default value is 1:

```
jdbc:hive2://> SET mapred.job.reuse.jvm.num.tasks=5;
```

We can also set the value to –1 to indicate that all the tasks for a job will run in the same JVM.

Parallel execution

Hive queries commonly are translated into a number of stages that are executed by the default sequence. These stages are not always dependent on each other. Instead, they can run in parallel to save the overall job running time. We can enable this feature with the following settings:

```
jdbc:hive2://> SET hive.exec.parallel=true;—default false
jdbc:hive2://> SET hive.exec.parallel.thread.number=16;
-- default 8, it defines the max number for running in parallel
```

Parallel execution will increase the cluster utilization. If the utilization of a cluster is already very high, parallel execution will not help much in terms of overall performance.

Join optimization

We have already discussed optimization in different types of Hive joins in [Chapter 4](#), *Data Selection and Scope*. Here, we'll briefly review the key settings for join improvement.

Common join

The common join is also called reduce side join. It is a basic join in Hive and works for most of the time. For common joins, we need to make sure the big table is on the right-most side or specified by hit, as follows:

```
/*+ STREAMTABLE(stream_table_name) */.
```

Map join

Map join is used when one of the join tables is small enough to fit in the memory, so it is very fast but limited. Since Hive 0.7.0, Hive can convert map join automatically with the following settings:

```
jdbc:hive2://> SET hive.auto.convert.join=true; --default false
jdbc:hive2://> SET hive.mapjoin.smalltable.filesize=600000000;
--default 25M
jdbc:hive2://> SET hive.auto.convert.join.noconditionaltask=true;
--default false. Set to true so that map join hint is not needed

jdbc:hive2://> SET hive.auto.convert.join.noconditionaltask.size=10000000;
--The default value controls the size of table to fit in memory
```

Once autoconvert is enabled, Hive will automatically check if the smaller table file size is bigger than the value specified by `hive.mapjoin.smalltable.filesize`, and then Hive will convert the join to a common join. If the file size is smaller than this threshold, it will try to convert the common join into a map join. Once autoconvert join is enabled, there is no need to provide the map join hints in the query.

Bucket map join

Bucket map join is a special type of map join applied on the bucket tables. To enable bucket map join, we need to enable the following settings:

```
jdbc:hive2://> SET hive.auto.convert.join=true; --default false
jdbc:hive2://> SET hive.optimize.bucketmapjoin=true; --default false
```

In bucket map join, all the join tables must be bucket tables and join on buckets columns. In addition, the buckets number in bigger tables must be a multiple of the bucket number in the small tables.

Sort merge bucket (SMB) join

SMB is the join performed on the bucket tables that have the same sorted, bucket, and join condition columns. It reads data from both bucket tables and performs common joins (map and reduce triggered) on the bucket tables. We need to enable the following properties to use SMB:

```
jdbc:hive2://> SET hive.input.format=
. . . . . . .> org.apache.hadoop.hive.ql.io.BucketizedHiveInputFormat;
jdbc:hive2://> SET hive.auto.convert.sortmerge.join=true;
jdbc:hive2://> SET hive.optimize.bucketmapjoin=true;
jdbc:hive2://> SET hive.optimize.bucketmapjoin.sortedmerge=true;
jdbc:hive2://> SET hive.auto.convert.sortmerge.join.noconditionaltask=true;
```

Sort merge bucket map (SMBM) join

SMBM join is a special bucket join but triggers map-side join only. It can avoid caching all rows in the memory like map join does. To perform SMBM joins, the join tables must have the same bucket, sort, and join condition columns. To enable such joins, we need to enable the following settings:

```
jdbc:hive2://> SET hive.auto.convert.join=true;
jdbc:hive2://> SET hive.auto.convert.sortmerge.join=true
jdbc:hive2://> SET hive.optimize.bucketmapjoin=true;
jdbc:hive2://> SET hive.optimize.bucketmapjoin.sortedmerge=true;
jdbc:hive2://> SET hive.auto.convert.sortmerge.join.noconditionaltask=true;
jdbc:hive2://> SET
hive.auto.convert.sortmerge.join.bigtable.selection.policy=
org.apache.hadoop.hive.ql.optimizer.TableSizeBasedBigTableSelectorForAutoSM
J;
```

Skew join

When working with data that has a highly uneven distribution, the data skew could happen in such a way that a small number of compute nodes must handle the bulk of the computation. The following setting informs Hive to optimize properly if data skew happens:

```
jdbc:hive2://> SET hive.optimize.skewjoin=true;
--If there is data skew in join, set it to true. Default is false.

jdbc:hive2://> SET hive.skewjoin.key=100000;
--This is the default value. If the number of key is bigger than
--this, the new keys will send to the other unused reducers.
```

Note

Skew data could happen on the GROUP BY data too. To optimize it, we need to do the following settings to enable skew data optimization in the GROUP BY result:

```
SET hive.groupby.skewindata=true;
```

Once configured, Hive will first trigger an additional MapReduce job whose map output will randomly distribute to the reducer to avoid data skew.

For more information about Hive join optimization, please refer to the Apache Hive wiki available at
https://cwiki.apache.org/confluence/display/Hive/LanguageManual+JoinOptimization and
https://cwiki.apache.org/confluence/display/Hive/Skewed+Join+Optimization.

Summary

In this chapter, we first covered how to identify performance bottlenecks using the `EXPLAIN` and `ANALYZE` statements. Then, we spoke about the design optimization for performance when using tables, partition, and index. We also covered the data file optimization including file format, compression, and storage. At the end of this chapter, we discussed job and query optimization in Hive. After going through this chapter, we should be able to do performance troubleshooting and tuning in Hive.

In the next chapter, we'll talk about function extensions for Hive.

Chapter 8. Extensibility Considerations

Although Hive has many built-in functions, users sometimes will need power beyond that provided by built-in functions. For these instances, Hive offers the following three main areas where its functionalities can be extended:

- **User-defined function** (**UDF**): This provides a way to extend functionalities with an external function (mainly written in Java) that can be evaluated in HQL
- **Streaming**: This plugs in users' own customized mappers and reducers programs in the data streaming
- **SerDe**: This stands for serializers and deserializers and provides a way to serialize or deserialize a custom file format with files stored on HDFS

In this chapter, we'll talk about each of them in more detail.

User-defined functions

Hive defines the following three types of UDF:

- **UDFs**: These are regular user-defined functions that operate row-wise and output one result for one row, such as most built-in mathematic and string functions.
- **UDAFs**: These are user-defined aggregating functions that operate row-wise or group-wise and output one row or one row for each group as a result, such as the `MAX` and `COUNT` built-in functions.
- **UDTFs**: These are user-defined table-generating functions that also operate row-wise, but they produce multiple rows/tables as a result, such as the `EXPLODE` function. UDTF can be used either after `SELECT` or after the `LATERAL VIEW` statement.

  ### Note

  Since Hive is implemented in Java, UDFs should be written in Java as well. Since Java supports running code in other languages through the `javax.script` API (see http://docs.oracle.com/javase/6/docs/api/javax/script/package-summary.html), UDFs can be written in languages other than Java. In this book, we only focus on Java UDFs.

We'll start looking at the Java code template for each kind of function in more detail.

The UDF code template

The code template for a regular UDF is as follows:

```
package com.packtpub.hive.essentials.hiveudf;

import org.apache.hadoop.hive.ql.exec.UDF;
import org.apache.hadoop.hive.ql.exec.Description;
import org.apache.hadoop.hive.ql.udf.UDFType;

//Below are options or add more when needed
import org.apache.hadoop.io.Text;
import org.apache.commons.lang.StringUtils;

@Description(
name = "udf_name",
value = "_FUNC_(arg1, arg2, ... argN) - A short description for the
function",
extended = "This is more detail about the function, such as syntax,
examples."
)
@UDFType(deterministic = true, stateful = false)

public class udf_name extends UDF {
    public String evaluate(){
        /*
         * Do something here
         */
        return "return the udf result";
    }
    //override is supported
    public String evaluate(<Type_arg1> arg1,..., <Type_argN> argN){
        /*
         * Do something here
         */
        return "return the udf result";
    }
}
```

In the preceding template, the package definition and imports should be self-explanatory. We can import whatever is needed besides the top three mandatory libraries. The `@Description` annotation is a useful Hive specific annotation to provide usage information for the UDF in the Hive console. The information defined in the `value` property will be shown in the HQL `DESCRIBE FUNCTION` command. The information defined in the `extended` property will be shown in the HQL `DESCRIBE FUNCTION EXTENDED` command. The `@UDFType` annotation tells Hive what behavior to expect from the function. A deterministic UDF (`deterministic = true`) is a function that always gives the same result when passed the same arguments, such as `LENGTH(string input)`, `MAX()`, and so on. On the other hand, a non-deterministic (`deterministic = false`) UDF can return a different result for the same set of arguments, for example, `UNIX_TIMESTAMP()` returning the current timestamp in the default time zone. The stateful (`stateful = true`) property allows functions to keep some static variables available across rows, such as

`ROW_NUMBER()`, which assigns sequential numbers for all rows in a table.

All UDFs extend the Hive `UDF` class, so the UDF subclass must implement the `evaluate` method called by Hive. The `evaluate` method can be overridden for a different purpose. In this method, we can implement whatever logic and exception handling the design for the function using the Java Hadoop library and the Hadoop data type for MapReduce data serialization, such as `TEXT`, `DoubleWritable`, `INTWritable`, and so on.

The UDAF code template

In this section, we introduce the UDAF code template by extending it from the UDAF class. The code template is as follows:

```
package com.packtpub.hive.essentials.hiveudaf;

import org.apache.hadoop.hive.ql.exec.UDAF;
import org.apache.hadoop.hive.ql.exec.UDAFEvaluator;
import org.apache.hadoop.hive.ql.exec.Description;
import org.apache.hadoop.hive.ql.udf.UDFType;

@Description(
name = "udaf_name",
value = "_FUNC_(arg1, arg2, ... argN) - A short description for the
function",
extended = "This is more detail about the function, such as syntax,
examples."
)
@UDFType(deterministic = false, stateful = true)

public final class udaf_name extends UDAF {
  /**
   * The internal state of an aggregation function.
   *
   * Note that this is only needed if the internal state
   * cannot be represented by a primitive.
   *
   * The internal state can contain fields with types like
   * ArrayList<String> and HashMap<String,Double> if needed.
   */
  public static class UDAFState {
    private <Type_state1> state1;
    private <Type_stateN> stateN;
  }

  /**
   * The actual class for doing the aggregation. Hive will
   * automatically look for all internal classes of the UDAF
   * that implements UDAFEvaluator.
   */
  public static class UDAFExampleAvgEvaluator implements UDAFEvaluator {

    UDAFState state;

    public UDAFExampleAvgEvaluator() {
      super();
      state = new UDAFState();
      init();
    }

    /**
     * Reset the state of the aggregation.
     */
    public void init() {
```

```java
  /*
   * Examples for initializing state.
   */
  state.state1 = 0;
  state.stateN = 0;
}

/**
 * Iterate through one row of original data.
 *
 * The number and type of arguments need to be the same as we
 * call this UDAF from the Hive command line.
 *
 * This function should always return true.
 */
public boolean iterate(<Type_arg1> arg1,..., <Type_argN> argN)
{
  /*
   * Add logic here for how to do aggregation if there is
   * a new value to be aggregated.
   */
  return true;
}

/**
 * Called on the mapper side on different data nodes.
 * Terminate a partial aggregation and return the state.
 * If the state is a primitive, just return primitive Java
 * classes like Integer or String.
 */
public UDAFState terminatePartial() {
  /*
   * Check and return a partial result in expectations.
   */
  return state;
}

/**
 * Merge with a partial aggregation.
 *
 * This function should always have a single argument,
 * which has the same type as the return value of
 * terminatePartial().
 */
public boolean merge(UDAFState o) {
  /*
   * Define operations how to merge the result calculated
   * from all data nodes.
   */
  return true;
}

/**
 * Terminates the aggregation and returns the final result.
 */
public long terminate() {
```

```
        /*
         * Check and return final result in expectations.
         */
        return state.stateN;
    }
  }
}
```

A UDAF must be a subclass of `org.apache.hadoop.hive.ql.exec.UDAF` containing one or more nested static classes implementing `org.apache.hadoop.hive.ql.exec.UDAFEvaluator`. Make sure that the inner class that implements `UDAFEvaluator` is defined as public. Otherwise, Hive won't be able to use reflection and determine the `UDAFEvaluator` implementation. We should also implement the five required functions, `init`, `iterate`, `terminatePartial`, `merge`, and `terminate`, already described in the code comments.

Note

Both UDF and UDAF can also be implemented by extending from the `GenericUDF` and `GenericUDAFEvaluator` classes to avoid using Java reflection for better performance. And, these generic functions are actually extended by Hive's built-in UDFs implementations internally. Generic functions support complex data types, such as `MAP`, `ARRAY`, and `STRUCT`, as arguments, but the `UDF` and `UDAF` class do not. For more information about `GenericUDF`, please refer to the Apache Hive wiki at https://cwiki.apache.org/confluence/display/Hive/GenericUDFCaseStudy.

The UDTF code template

To implement UDTF, there is only one way by extending from
`org.apache.hadoop.hive.ql.exec.GenericUDTF`. There is no plain `UDTF` class. We need
to implement three methods: `initialize`, `process`, and `close`. The UDTF will call the
`initialize` method, which returns the information of the function output, such as data
type, number of output, and so on. Then, the `process` method is called to do core function
logic with arguments and forward the result. At the end, the `close` method will do a
proper cleanup, if needed. The code template for UDTF is as follows:

```
package com.packtpub.hive.essentials.hiveudtf;

import org.apache.hadoop.hive.ql.udf.generic.GenericUDTF;
import org.apache.hadoop.hive.ql.exec.Description;
import org.apache.hadoop.hive.ql.exec.UDFArgumentException;
import org.apache.hadoop.hive.ql.metadata.HiveException;
import org.apache.hadoop.hive.serde2.objectinspector.ObjectInspector;
import
org.apache.hadoop.hive.serde2.objectinspector.ObjectInspectorFactory;
import
org.apache.hadoop.hive.serde2.objectinspector.PrimitiveObjectInspector;
import org.apache.hadoop.hive.serde2.objectinspector.StructObjectInspector;
import
org.apache.hadoop.hive.serde2.objectinspector.primitive.PrimitiveObjectInsp
ectorFactory;

@Description(
name = "udtf_name",
value = "_FUNC_(arg1, arg2, ... argN) - A short description for the
function",
extended = "This is more detail about the function, such as syntax,
examples."
)
public class udtf_name extends GenericUDTF {
  private PrimitiveObjectInspector stringOI = null;
  /**
   * This method will be called exactly once per instance.
   * It performs any custom initialization logic we need.
   * It is also responsible for verifying the input types and
   * specifying the output types.
   */
  @Override
  public StructObjectInspector initialize(ObjectInspector[] args)
  throws UDFArgumentException {

    //Check number of arguments.
    if (args.length != 1) {
      throw new UDFArgumentException("The UDTF should take exactly one
argument");
    }
    /*
     * Check that the input ObjectInspector[] array contains a
     * single PrimitiveObjectInspector of the Primitive type,
     * such as String.
```

```java
     */
    if (args[0].getCategory() != ObjectInspector.Category.PRIMITIVE
        &&
       ((PrimitiveObjectInspector) args[0]).getPrimitiveCategory() !=
        PrimitiveObjectInspector.PrimitiveCategory.STRING) {
        throw new UDFArgumentException("The UDTF should take a string as a
parameter");
    }

    stringOI = (PrimitiveObjectInspector) args[0];
    /*
     * Define the expected output for this function, including
     * each alias and types for the aliases.
     */
    List<String> fieldNames = new ArrayList<String>(2);
    List<ObjectInspector> fieldOIs = new ArrayList<ObjectInspector>(2);
    fieldNames.add("alias1");
    fieldNames.add("alias2");

fieldOIs.add(PrimitiveObjectInspectorFactory.javaStringObjectInspector);
    fieldOIs.add(PrimitiveObjectInspectorFactory.javaIntObjectInspector);
    //Set up the output schema.
    return
ObjectInspectorFactory.getStandardStructObjectInspector(fieldNames,
fieldOIs);
  }

  /**
   * This method is called once per input row and generates
   * output. The "forward" method is used (instead of
   * "return") in order to specify the output from the function.
   */
  @Override
  public void process(Object[] record) throws HiveException {
   /*
    * We may need to convert the object to a primitive type
    * before implementing customized logic.
    */
    final String recStr = (String)
stringOI.getPrimitiveJavaObject(record[0]);

    //emit newly created structs after applying customized logic.
    forward(new Object[] {recStr, Integer.valueOf(1)});
  }

  /**
   * This method is for any cleanup that is necessary before
   * returning from the UDTF. Since the output stream has
   * already been closed at this point, this method cannot
   * emit more rows.
   */
  @Override
  public void close() throws HiveException {
    //Do nothing.
  }
}
```

Development and deployment

We'll go through the whole development and deployment steps using an example. Let's create a Hive function called `toUpper`, which will convert a string to uppercase using the following steps:

1. Download and install a Java IDE, such as Eclipse, from http://www.eclipse.org/downloads/packages/eclipse-ide-java-developers/lunasr1.
2. Start the IDE and create a Java project.
3. Right-click on the project to choose the **Build Path** | **Configure Build Path** | **Add External Jars** option. It will open a new window. Navigate to the directory having the library of Hive and Hadoop. Then, select and add all JAR files needed to import. We can also resolve library dependency automatically by using Maven (see http://maven.apache.org/) and the proper `pom.xml` file. How to configure a library repository in `pom.xml` files is usually well described in the Hadoop vendor package or Apache Hive and Hadoop help documents.
4. In the IDE, create the `toupper.java` file as follows, according to the UDF template mentioned previously:

```
package com.packtpub.hive.essentials.hiveudf;

import org.apache.hadoop.hive.ql.exec.UDF;
import org.apache.hadoop.io.Text;

class ToUpper extends UDF {

  public Text evaluate(Text input) {
    if(input == null) return null;
    return new Text(input.toString().toUpperCase());
  }
}
```

5. Now, export this project as a JAR file (or built by Maven) named as `toupper.jar`.
6. Copy this JAR file in a directory, such as `/home/dayongd/hive/lib/`, in a node of the Hive cluster.
7. Add the JAR to the Hive environment using one of the following options (option 3 or 4 is recommended):

   - **Option 1**: Run `ADD JAR /home/dayongd/hive/lib/toupper.jar` in the Hive CLI. This is only valid for the current session, but does not work for ODBC connections.
   - **Option 2**: Add `ADD JAR /home/dayongd/hive/lib/toupper.jar` in `/home/$USER/.hiverc` (we can create the file if it is not there). In this case, the file needs to be deployed to every node from where we might launch the Hive shell. This is only valid for the current session, but does not work for ODBC connections.
   - **Option 3**: Add the following configuration in the `hive-site.xml` file:

     ```
     <property>
     ```

```
<name>hive.aux.jars.path</name>
<value>file:///home/dayongd/hive/lib/toupper.jar</value>
</property>
```

- **Option 4**: Copy and paste the JAR file to the `/${HIVE_HOME}/auxlib/` folder (create it if it does not exist).

8. Create the function. We can create a temporary function that is only valid in the current Hive session as follows:

```
CREATE TEMPORARY FUNCTION toUpper AS
'com.packtpub.hive.essentials.hiveudf.toupper';
```

Note

Since Hive 0.13.0, we can use one command to add JAR and create permanent functions, which is registered to the megastore and can be referenced in a query without creating a temporary function in each session:

```
CREATE FUNCTION toUpper AS
'com.packtpub.hive.essentials.hiveudf.ToUpper' USING JAR
'hdfs:///path/to/jar';
```

9. Verify the function:

```
SHOW FUNCTIONS ToUpper;
DESCRIBE FUNCTION ToUpper;
DESCRIBE FUNCTION EXTENDED ToUpper;
```

10. Use the UDF in HQL:

```
SELECT toUpper(name) FROM employee LIMIT 1000;
```

11. Drop the function when needed:

```
DROP TEMPORARY FUNCTION IF EXISTS toUpper;
```

Streaming

Hive can also leverage the streaming feature in Hadoop to transform data in an alternative way. The streaming API opens an I/O pipe to an external process (script). Then, the process reads data from the standard input and writes the results out through the standard output. In Hive, we can use `TRANSFORM` clauses in HQL directly, and embed the mapper and the reducer scripts written in commands, shell scripts, Java, or other programming languages. Although streaming brings overhead by using serialization/deserialization between processes, it is a simpler coding mode for developers, especially non-Java developers. The syntax of the `TRANSFORM` clause is as follows:

```
FROM (
    FROM src
    SELECT TRANSFORM '(' expression (',' expression)* ')'
    (inRowFormat)?
    USING 'map_user_script'
    (AS colName (',' colName)*)?
    (outRowFormat)? (outRecordReader)?
    (CLUSTER BY?|DISTRIBUTE BY? SORT BY?) src_alias
 )
 SELECT TRANSFORM '(' expression (',' expression)* ')'
 (inRowFormat)?
 USING 'reduce_user_script'
 (AS colName (',' colName)*)?
 (outRowFormat)? (outRecordReader)?
```

By default, the `INPUT` values for the user script are the following:

- Columns transformed to `STRING` values
- Delimited by a tab
- `NULL` values converted to the literal string `N` (differentiates `NULL` values from empty strings)

By default, the `OUTPUT` values of the user script are the following:

- Treated as tab-separated `STRING` columns
- `N` will be reinterpreted as `NULL`
- The resulting `STRING` column will be cast to the data type specified in the table declaration

These defaults can be overridden with `ROW FORMAT`. An example of Hive streaming using the Python script `upper.py` is as follows:

```
#!/usr/bin/env python
'''
This is a script to upper all cases
'''
import sys

def main():
    try:
        for line in sys.stdin:
```

```
        n = line.strip()
        print n.upper()
    except:
        return None
if __name__ == "__main__":main()
```

Test the script, as follows:

```
$ echo "Will" | python upper.py
$ WILL
```

Call the script in the Hive CLI from HQL:

```
jdbc:hive2://> ADD FILE /home/dayongd/Downloads/upper.py;
jdbc:hive2://> SELECT TRANSFORM (name,work_place[0])
. . . . . . .> USING 'python upper.py' AS (CAP_NAME,CAP_PLACE)
. . . . . . .> FROM employee;
+-----------+-------------+
| cap_name  | cap_place   |
+-----------+-------------+
| MICHAEL   | MONTREAL    |
| WILL      | MONTREAL    |
| SHELLEY   | NEW YORK    |
| LUCY      | VANCOUVER   |
| STEVEN    | NULL        |
+-----------+-------------+
5 rows selected (30.101 seconds)
```

Note

The TRANSFORM command is not allowed when SQL standard-based authorization is configured, since Hive 0.13.0.

SerDe

SerDe stands for Serializer and Deserializer. It is the technology that Hive uses to process records and map them to column data types in Hive tables. To explain the scenario of using SerDe, we need to understand how Hive reads and writes data.

The process to read data is as follows:

1. Data is read from HDFS.
2. Data is processed by the `INPUTFORMAT` implementation, which defines the input data split and key/value records. In Hive, we can use `CREATE TABLE… STORED AS <FILE_FORMAT>` (see [Chapter 7](#), *Performance Considerations,* for available file formats) to specify which `INPUTFORMAT` it reads from.
3. The Java `Deserializer` class defined in SerDe is called to format the data into a record that maps to column and data types in a table.

For an example of reading data, we can use JSON SerDe to read the `TEXTFILE` format data from HDFS and translate each row of the JSON attribute and value to rows in Hive tables with the correct schema.

The process to write data is as follows:

1. Data (such as using an `INSERT` statement) to be written is translated by the `Serializer` class defined in SerDe to the format that the `OUTPUTFORMAT` class can read.
2. Data is processed by the `OUTPUTFORMAT` implementation, which creates the `RecordWriter` object. Similar to the `INPUTFORMAT` implementation, the `OUTPUTFORMAT` implementation is specified in the same way as a table where it writes the data.
3. The data is written to the table (data saved in the HDFS).

For an example of writing data, we can write a row-column of data to Hive tables using JSON SerDe, which translates data to a JSON text string saved to the HDFS.

Recent Hive versions uses the `org.apache.hadoop.hive.serde2` library, where `org.apache.hadoop.hive.serde` is the deprecated library. A list of commonly used SerDe in Hive is as follows:

- **LazySimpleSerDe**: The default built-in SerDe (`org.apache.hadoop.hive.serde2.lazy.LazySimpleSerDe`) that's used with the `TEXTFILE` format. It can be implemented as follows:

  ```
  jdbc:hive2://> CREATE TABLE test_serde_lz
  . . . . . . .> STORED AS TEXTFILE AS
  . . . . . . .> SELECT name from employee;
  No rows affected (32.665 seconds)
  ```

- **ColumnarSerDe**: This is the built-in SerDe used with the `RCFILE` format. It can be used as follows:

```
jdbc:hive2://> CREATE TABLE test_serde_cs
. . . . . .> ROW FORMAT SERDE
. . . . . .> 'org.apache.hadoop.hive.serde2.columnar.ColumnarSerDe'
. . . . . .> STORED AS RCFile AS
. . . . . .> SELECT name from employee;
No rows affected (27.187 seconds)
```

- **RegexSerDe**: This is the built-in Java regular expression SerDe to parse text files. It can be used as follows:

```
--Parse , seperate fields
jdbc:hive2://> CREATE TABLE test_serde_rex(
. . . . . .> name string,
. . . . . .> sex string,
. . . . . .> age string
. . . . . .> )
. . . . . .> ROW FORMAT SERDE
. . . . . .> 'org.apache.hadoop.hive.contrib.serde2.RegexSerDe'
. . . . . .> WITH SERDEPROPERTIES(
. . . . . .>   'input.regex' = '([^,]*),([^,]*),([^,]*)',
. . . . . .>   'output.format.string' = '%1$s %2$s %3$s'
. . . . . .> )
. . . . . .> STORED AS TEXTFILE;
No rows affected (0.266 seconds)
```

- **HBaseSerDe**: This is the built-in SerDe to enable Hive to integrate with HBase. We can store Hive tables in HBase by leveraging this SerDe. Make sure to have HBase installed before running the following query:

```
jdbc:hive2://> CREATE TABLE test_serde_hb(
. . . . . .> id string,
. . . . . .> name string,
. . . . . .> sex string,
. . . . . .> age string
. . . . . .> )
. . . . . .> ROW FORMAT SERDE
. . . . . .> 'org.apache.hadoop.hive.hbase.HBaseSerDe'
. . . . . .> STORED BY
. . . . . .> 'org.apache.hadoop.hive.hbase.HBaseStorageHandler'
. . . . . .> WITH SERDEPROPERTIES (
. . . . . .> "hbase.columns.mapping"=
. . . . . .> ":key,info:name,info:sex,info:age"
. . . . . .> )
. . . . . .> TBLPROPERTIES("hbase.table.name" = "test_serde");
No rows affected (0.387 seconds)
```

- **AvroSerDe**: This is the built-in SerDe that enables reading and writing Avro (see http://avro.apache.org/) data in Hive tables. Avro is a remote procedure call and data serialization framework. Since Hive 0.14.0, Avro-backed tables can simply be created by using the CREATE TABLE… STORED AS AVRO statement, as follows:

```
jdbc:hive2://> CREATE TABLE test_serde_avro(
. . . . . .> name string,
. . . . . .> sex string,
. . . . . .> age string
```

```
. . . . . . .> )
. . . . . . .> ROW FORMAT SERDE
. . . . . . .> 'org.apache.hadoop.hive.serde2.avro.AvroSerDe'
. . . . . . .> STORED AS INPUTFORMAT
. . . . . . .>
'org.apache.hadoop.hive.ql.io.avro.AvroContainerInputFormat'
. . . . . . .> OUTPUTFORMAT
. . . . . . .>
'org.apache.hadoop.hive.ql.io.avro.AvroContainerOutputFormat'
. . . . . . .>;
No rows affected (0.31 seconds)
```

- **ParquetHiveSerDe**: This is the built-in SerDe
  (`parquet.hive.serde.ParquetHiveSerDe`) that enables reading and writing the
  Parquet data format since Hive 0.13.0. It can be used as follows:

```
jdbc:hive2://> CREATE TABLE test_serde_parquet
. . . . . . .> STORED AS PARQUET AS
. . . . . . .> SELECT name from employee;
No rows affected (34.079 seconds)
```

- **OpenCSVSerDe**: This is the SerDe to read and write CSV data. It comes as a built-in
  SerDe since Hive 0.14.0. We can also install the implementation from other open
  source libraries, such as https://github.com/ogrodnek/csv-serde. It can be used as
  follows:

```
jdbc:hive2://> CREATE TABLE test_serde_csv(
. . . . . . .> name string,
. . . . . . .> sex string,
. . . . . . .> age string
. . . . . . .>)
. . . . . . .> ROW FORMAT SERDE
. . . . . . .> 'org.apache.hadoop.hive.serde2.OpenCSVSerde'
. . . . . . .> STORED AS TEXTFILE;
```

- **JSONSerDe**: This is a third-party SerDe to read and write JSON data records with
  Hive. Make sure to install it (from https://github.com/rcongiu/Hive-JSON-Serde)
  before running the following query:

```
jdbc:hive2://> CREATE TABLE test_serde_js(
. . . . . . .> name string,
. . . . . . .> sex string,
. . . . . . .> age string
. . . . . . .> )
. . . . . . .> ROW FORMAT SERDE 'org.openx.data.jsonserde.JsonSerDe'
. . . . . . .> STORED AS TEXTFILE;
No rows affected (0.245 seconds)
```

Hive also allows users to define a customized SerDe if none of these work for their data
format. For more information about custom SerDe, please refer to the Apache wiki at
https://cwiki.apache.org/confluence/display/Hive/DeveloperGuide#DeveloperGuide-
HowtoWriteYourOwnSerDe.

Summary

In this chapter, we introduced three main areas to extend Hive's functionalities. We also covered three user-defined functions in Hive as well as the coding template and deployment steps to guide your coding and deployment practice. Then, we talked about streaming in Hive to plug in your own code, which does not have to be Java code. At the end of this chapter, we discussed the available SerDe in Hive to parse different formats of data files when reading or writing data. After going through this chapter, we should be able to write basic UDFs, plug code in streamings, and use available SerDe in Hive.

In the next chapter, we'll talk about security considerations for Hive.

Chapter 9. Security Considerations

In most open source software, security is one of the most important areas, but always addressed at a later stage. As the main SQL-like interface of data in Hadoop, Hive must ensure that data is securely protected and accessed. For this reason, security in Hive is now considered as an integral and important part of the Hadoop ecosystem. The earlier version of Hive mainly relied on the HDFS for security. The security of Hive gradually became mature after HiveServer2 was released as an important milestone of the Hive server.

This chapter will discuss Hive security in the following areas:

- Authentication
- Authorization
- Encryption

Authentication

Authentication is the process of verifying the identity of a user by obtaining the user's credentials. Hive has offered authentication since HiveServer2. In the previous HiveServer, if we could access the host/port over the network, we could access the data. In this case, the Hive Metastore Server can be used to authenticate thrift clients using Kerberos. As mentioned in [Chapter 2](#), *Setting Up the Hive Environment,* it is strongly recommended to upgrade the Hive server to HiveServer2 in terms of security and reliability. In this section, we will briefly talk about authentication configurations in both Metastore Server and HiveServer2.

Note

Kerberos

Kerberos is a network authentication protocol developed by MIT as part of Project Athena. It uses time-sensitive tickets that are generated using symmetric key cryptography to securely authenticate a user in an unsecured network environment. Kerberos is derived from Greek mythology, where Kerberos was the three-headed dog that guarded the gates of Hades. The three-headed part refers to the three parties involved in the Kerberos authentication process: client, server, and **Key Distribution Center** (**KDC**). All clients and servers registered to KDC are known as a realm, which is typically the domain's DNS name in all caps. For more information, please refer to the MIT Kerberos website at http://web.mit.edu/kerberos/.

Metastore server authentication

To force clients to authenticate with the Hive Metastore server using Kerberos, we can set the following properties in the `hive-site.xml` file:

- Enable the **Simple Authentication and Security Layer** (**SASL**) framework to enforce client Kerberos authentication, as follows:

```
<property>
  <name>hive.metastore.sasl.enabled</name>
  <value>true</value>
  <description>If true, the metastore thrift interface will be secured
with SASL framework. Clients must authenticate with Kerberos.
</description>
</property>
```

- Specify the Kerberos keytab that is generated. Override the following example if we want to keep the file in another place. Make sure the file access permissions are set to 400 implying only read permission for the owner to avoid their identity being stolen by others:

```
<property>
  <name>hive.metastore.kerberos.keytab.file</name>
  <value>/etc/hive/conf/hive.keytab</value>
  <description>The sample path to the Kerberos Keytab file containing
the metastore thrift server's service principal.</description>
</property>
```

- Specify the Kerberos principal pattern string. The special string `_HOST` will be replaced automatically with the correct hostnames. The `YOUR-REALM.COM` value should be replaced by the actual realm name:

```
<property>
  <name>hive.metastore.kerberos.principal</name>
  <value>hive/_HOST@YOUR-REALM.COM</value>
  <description>The service principal for the metastore thrift server.
</description>
</property>
```

HiveServer2 authentication

HiveServer2 supports the following authentications. To configure HiveServer2 to use one of these authentication modes, we can set the proper properties in `hive_site.xml` as follows:

- **None authentication**: None authentication is what's in the default settings. "None" here means Hive allows anonymous access as shown in the following setting:

```
<property>
    <name>hive.server2.authentication</name>
    <value>NONE</value>
</property>
```

- **Kerberos authentication**: If Kerberos authentication is used, authentication is supported between the thrift client and HiveServer2, and between HiveServer2 and secure HDFS. To enable Kerberos authentication for HiveServer2, we can set the following properties by overriding the keytab path (if we want to keep the file in another place) as well as changing `YOUR-REALM.COM` to the actual realm name:

```
<property>
  <name>hive.server2.authentication</name>
  <value>KERBEROS</value>
</property>
<property>
  <name>hive.server2.authentication.kerberos.keytab</name>
  <value>/etc/hive/conf/hive.keytab</value>
</property>
<property>
  <name>hive.server2.authentication.kerberos.principal</name>
  <value>hive/_HOST@YOUR-REALM.COM</value>
</property>
```

Once Kerberos is enabled, the JDBC client (such as Beeline) must include the principal parameter in the JDBC connection string such as the following:

```
jdbc:hive2://HiveServer2HostName:10000/default;principal=hive/HiveServer2HostName@YOUR-REALM.COM
```

- **LDAP authentication**: To configure HiveServer2 to use user and password validation backed by LDAP (see http://tools.ietf.org/html/rfc4511), we can set the following properties:

```
<property>
  <name>hive.server2.authentication</name>
  <value>LDAP</value>
</property>
<property>
  <name>hive.server2.authentication.ldap.url</name>
  <value>LDAP_URL, such as ldap://ldaphost@company.com</value>
</property>
<property>
  <name>hive.server2.authentication.ldap.Domain</name>
  <value>Your Domain Name</value>
```

```
</property>
```

To configure with OpenLDAP, we can add the setting of `baseDN` instead of the `Domain` property as follows:

```
<property>
  <name>hive.server2.authentication.ldap.baseDN</name>
  <value>LDAP_BaseDN, such as ou=people,dc=packtpub,dc=com</value>
</property>
```

- **Pluggable custom authentication**: Pluggable custom authentication provides a custom authentication provider for HiveServer2. To enable it, configure the settings as follows:

```
<property>
  <name>hive.server2.authentication</name>
  <value>CUSTOM</value>
</property>
<property>
  <name>hive.server2.custom.authentication.class</name>
  <value>pluggable-auth-class-name</value>
  <description> Custom authentication class name, such as
   com.packtpub.hive.essentials.hiveudf.customAuthenticator
  </description>
</property>
```

Note

The pluggable authentication with a customized class did not work until the bug (see https://issues.apache.org/jira/browse/HIVE-4778) was fixed in Hive 0.13.0.

The following is a sample of a customized class that implements the `org.apache.hive.service.auth.PasswdAuthenticationProvider` interface. The overridden `Authenticate` method has the core logic of how to authenticate a username and password. Make sure to copy the compiled JAR file to `$HIVE_HOME/lib/` so that the preceding settings can work.

customAuthenticator.java
```
package com.packtpub.hive.essentials.hiveudf;

import java.util.Hashtable;
import javax.security.sasl.AuthenticationException;
import org.apache.hive.service.auth.PasswdAuthenticationProvider;

/*
 * The customized class for HiveServer2 authentication
 */

public class customAuthenticator implements
PasswdAuthenticationProvider {

  Hashtable<String, String> authHashTable = null;

  public customAuthenticator () {
      authHashTable = new Hashtable<String, String>();
```

```
      authHashTable.put("user1", "passwd1");
      authHashTable.put("user2", "passwd2");
  }

  @Override
  public void Authenticate(String user, String password)
          throws AuthenticationException {

    String storedPasswd = authHashTable.get(user);

    if (storedPasswd != null && storedPasswd.equals(password))
        return;

    throw new AuthenticationException("customAuthenticator Exception:
Invalid user");
  }
}
```

- **Pluggable Authentication Modules** (**PAM**) **authentication**: Since Hive 0.13.0, it supports PAM authentication, which provides the benefit of plugging existing authentication mechanisms to Hive. Configure the following settings to enable PAM authentication. For more information about how to install PAM, please refer to the *Setting Up HiveServer2* article in the Apache Hive wiki at https://cwiki.apache.org/confluence/display/Hive/Setting+Up+HiveServer2#SettingUp PluggableAuthenticationModules(PAM).

```
<property>
  <name>hive.server2.authentication</name>
  <value>PAM</value>
</property>
<property>
  <name>hive.server2.authentication.pam.services</name>
  <value>pluggable-auth-class-name</value>
  <description> Set this to a list of comma-separated PAM services that
will be used. Note that a file with the same name as the PAM service
must exist in /etc/pam.d.</description>
</property>
```

Authorization

Authorization in Hive is used to verify if a user has permission to perform a certain action, such as creating, reading, and writing data or metadata. Hive provides three authorization modes: legacy mode, storage-based mode, and SQL standard-based mode.

Legacy mode

This is the default authorization mode in Hive, providing column and row-level authorization through HQL statements. However, it is not a completely secure authorization mode and has a couple of limitations. It can be mainly used to prevent good users from accidentally doing bad things rather than preventing malicious users' operations. In order to enable the legacy authorization mode, we need to set the following properties in `hive-site.xml`:

```
<property>
  <name>hive.security.authorization.enabled</name>
  <value>true</value>
  <description>enables or disable the hive client authorization
  </description>
</property>
<property>
  <name>hive.security.authorization.createtable.owner.grants</name>
  <value>ALL</value>
  <description>the privileges automatically granted to the owner whenever a
table gets created. An example like "select, drop" will grant select and
drop privilege to the owner of the table.
  </description>
</property>
```

Since this is not a secure authorization mode, we will not discuss more details here. For more HQL support in the legacy authorization mode, please refer to the Apache Hive wiki at https://cwiki.apache.org/confluence/display/Hive/Hive+Default+Authorization+-+Legacy+Mode.

Storage-based mode

The storage-based authorization mode (since Hive 0.10.0) relies on the authorization provided by the storage layer HDFS, which provides both POSIX and ACL permissions (available since Hive 0.14.0; refer to https://issues.apache.org/jira/browse/HIVE-7583). The storage-based authorization is enabled in the Hive Metastore server having a single consistent view of metadata across other applications in the ecosystem. This mode checks Hive user permissions against the POSIX permissions on the corresponding file directories in HDFS. In addition to the POSIX permissions model, HDFS also provides access control lists described in ACLs on HDFS at http://hadoop.apache.org/docs/r2.4.0/hadoop-project-dist/hadoop-hdfs/HdfsPermissionsGuide.html#ACLs_Access_Control_Lists. Considering its implementation, the storage-based authorization mode only offers authorization at the level of Hive databases, tables, and partitions rather than column and row level. With dependency on the HDFS permissions, it lacks the flexibility to manage the authorization through HQL statements.

To enable storage-based authorization mode, we can set the following properties in the `hive-site.xml` file:

```
<property>
    <name>hive.security.authorization.enabled</name>
    <value>true</value>
    <description>enable or disable the hive client authorization
    </description>
</property>
<property>
    <name>hive.security.authorization.manager</name>

<value>org.apache.hadoop.hive.ql.security.authorization.StorageBasedAuthori
zationProvider</value>
    <description>The class name of the Hive client authorization manager.
</description>
</property>
<property>
    <name>hive.server2.enable.doAs</name>
    <value>true</value>
    <description>Allows Hive queries to be run by the user who submits the
query rather than the hive user.</description>
</property>
</property>
    <name>hive.metastore.pre.event.listeners</name>

<value>org.apache.hadoop.hive.ql.security.authorization.AuthorizationPreEve
ntListener</value>
    <description>This turns on metastore-side security.</description>
</property>
<property>
    <name>hive.security.metastore.authorization.manager</name>

<value>org.apache.hadoop.hive.ql.security.authorization.StorageBasedAuthori
zationProvider</value>
    <description>authenticator manager class name to be used in the
```

```
metastore for authentication.</description>
</property>
```

Note

Since Hive 0.14.0, storage-based authorization also authorizes read privileges on databases and tables by default through the `hive.security.metastore.authorization.auth.reads` property. For more information, please refer to https://issues.apache.org/jira/browse/HIVE-8221.

SQL standard-based mode

For fine-grained access control on a column and row level, we can use SQL standard-based mode available since Hive 0.13.0. It is similar to the SQL authorization by using the `GRANT` and `REVOKE` statements to control access through the HiveServer2 configuration. However, tools such as Hive CLI and Hadoop/HDFS/MapReduce commands do not access data through HiveServer2, so SQL standard-based mode cannot authorize their access. Therefore, it is recommended to use storage-based mode together with SQL standard-based mode authorization to authorize users who do not access from HiveServer2.

To enable SQL standard-based mode authorization, we can set the following properties in the `hive-site.xml` file:

```
<property>
    <name>hive.server2.enable.doAs</name>
    <value>false</value>
    <description>Allows Hive queries to be run by the user who submits the
query rather than the hive user. Need to turn if off for this SQL standard-
base mode</description>
</property>
<property>
    <name>hive.users.in.admin.role</name>
    <value>dayongd,administrator</value>
    <description>Comma-separated list of users assigned to the ADMIN role.
</description>
</property>
<property>
     <name>hive.security.authorization.enabled</name>
     <value>true</value>
</property>
<property>
     <name>hive.security.authorization.manager</name>

<value>org.apache.hadoop.hive.ql.security.authorization.plugin.sql</value>
</property>
<property>
     <name>hive.security.authenticator.manager</name>

<value>org.apache.hadoop.hive.ql.security.SessionStateUserAuthenticator</va
lue>
</property>
<property>
     <name>hive.metastore.uris</name>
     <value>" "</value>
     <description>" " (quotation marks surrounding a single empty space).
</description>
</property>
```

Before restarting HiveServer2, the users in the configured admin role must run the following command to make the admin role effective, and then restart HiveServer2:

```
jdbc:hive2://> GRANT admin TO USER dayongd;
```

The basic syntax to grant or revoke an authorization role or privilege is as follows:

```
GRANT <ROLENAME> TO <USERS> [ WITH ADMIN OPTION ];
REVOKE [ADMIN OPTION FOR] <ROLENAME> FROM <USERS>;
```

Here, the following parameters are used:

- `<ROLENAME>`: This can be a comma-separated name of roles
- `<USERS>`: This can be a user or a role
- `WITH ADMIN OPTION`: This makes sure that the user gets privileges to grant the role to other users/roles

Another example to grant or revoke an authorization is as follows:

```
GRANT <PRIVILEGE> ON <OBJECT> TO <USERS>;
REVOKE <PRIVILEGE> ON <OBJECT> FROM <USERS>;
```

Here, the following parameters are used:

- `<PRIVILEGE>`: This can be `INSERT`, `SELECT`, `UPDATE`, `DELETE`, or `ALL`
- `<USERS>`: This can be a user or a role
- `<OBJECT>`: This is a table or a view

For more examples of HQL statements to manage SQL standard-based authorization, please refer to the Apache Hive wiki at https://cwiki.apache.org/confluence/display/Hive/SQL+Standard+Based+Hive+Authorization+Configuration.

Note

Sentry

Sentry is a highly modular system for providing centralized, fine-grained, role-based authorization to both data and metadata stored on an Apache Hadoop cluster. It can be integrated with Hive to deliver advanced authorization controls. For more information about Sentry, please refer to http://incubator.apache.org/projects/sentry.html.

Encryption

For sensitive and legally protected data such as **personal identity information** (**PII**), it is required to store the data in encrypted format in the filesystem. However, Hive does not natively support encryption and decryption yet (see https://issues.apache.org/jira/browse/HIVE-5207).

Alternatively, we can look for third-party tools to encrypt and decrypt data after exporting it from Hive, but this requires additional postprocessing. The new HDFS encryption (see https://issues.apache.org/jira/browse/HDFS-6134) offers great transparent encryption and decryption of data on HDFS. It will satisfy our request if we want to encrypt the whole dataset in HDFS. However, it cannot be applied to the selected column and row level in the table of Hive, where most PII that is encrypted is only a part of raw data. In this case, the best solution for now is to use Hive UDF to plug in encryption and decryption implementations on selected columns or partial data in the Hive tables.

Sample UDF implementations for encryption and decryption using the AES encryption algorithm are as follows:

- `AESEncrypt.java`: The implementation is as follows:

  ```
  package com.packtpub.hive.essentials.hiveudf;

  import org.apache.hadoop.hive.ql.exec.UDF;
  import org.apache.hadoop.hive.ql.exec.Description;
  import org.apache.hadoop.hive.ql.udf.UDFType;

  @Description(
          name = "aesencrypt",
          value = "_FUNC_(str) - Returns encrypted string based on AES
  key.",
          extended = "Example:\n" +
          "  > SELECT aesencrypt(pii_info) FROM table_name;\n"
          )
  @UDFType(deterministic = true, stateful = false)
  /*
   * A Hive encryption UDF
   */
  public class AESEncrypt extends UDF {
      public String evaluate(String unencrypted) {
      String encrypted="";
       if(unencrypted != null) {
          try {
              encrypted = CipherUtils.encrypt(unencrypted);
          } catch (Exception e) {};
       }
       return encrypted;
      }
  }
  ```

- `AESDecrypt.java`: This can be implemented as follows:

  ```
  package com.packtpub.hive.essentials.hiveudf;
  ```

```java
import org.apache.hadoop.hive.ql.exec.UDF;
import org.apache.hadoop.hive.ql.exec.Description;
import org.apache.hadoop.hive.ql.udf.UDFType;

@Description(
        name = "aesdecrypt",
        value = "_FUNC_(str) - Returns unencrypted string based on
AES key.",
        extended = "Example:\n" +
        "  > SELECT aesdecrypt(pii_info) FROM table_name;\n"
        )
@UDFType(deterministic = true, stateful = false)
/*
 * A Hive decryption UDF
 */
public class AESDecrypt extends UDF {
    public String evaluate(String encrypted) {
    String unencrypted = new String(encrypted);
     if(encrypted != null) {
        try {
            unencrypted = CipherUtils.decrypt(encrypted);
        } catch (Exception e) {};
     }
     return unencrypted;
    }
}
```

- `CipherUtils.java`: This can be implemented as follows:

```java
package com.packtpub.hive.essentials.hiveudf;

import javax.crypto.Cipher;
import javax.crypto.spec.SecretKeySpec;
import org.apache.commons.codec.binary.Base64;
/*
 * The core encryption and decryption logic function
 */
public class CipherUtils
{
     //This is a secret key in terms of ASCII
    private static byte[] key = {
         0x75, 0x69, 0x69, 0x73, 0x40, 0x73, 0x41, 0x53, 0x65, 0x65,
0x72, 0x69, 0x74, 0x4b, 0x65, 0x75
    };

    public static String encrypt(String strToEncrypt)
    {
        try
        {
            //prepare algorithm
            Cipher cipher = Cipher.getInstance("AES/ECB/PKCS5Padding");
            final SecretKeySpec secretKey = new SecretKeySpec(key,
"AES");
            //initialize cipher for encryption
            cipher.init(Cipher.ENCRYPT_MODE, secretKey);
```

```
            //Base64.encodeBase64String that gives an ascii string
            final String encryptedString =
Base64.encodeBase64String(cipher.doFinal(strToEncrypt.getBytes()));
            return encryptedString.replaceAll("\r|\n", "");
        }
        catch (Exception e)
        {
            e.printStackTrace();
        }
        return null;

    }

    public static String decrypt(String strToDecrypt)
    {
        try
        {
            //prepare algorithm
            Cipher cipher = Cipher.getInstance("AES/ECB/PKCS5PADDING");
            final SecretKeySpec secretKey = new SecretKeySpec(key,
"AES");
            //initialize cipher for decryption
            cipher.init(Cipher.DECRYPT_MODE, secretKey);
            final String decryptedString = new
String(cipher.doFinal(Base64.decodeBase64(strToDecrypt)));
            return decryptedString;
        }
        catch (Exception e)
        {
                e.printStackTrace();

        }
        return null;
    }
}
```

Note

AES

Short for **Advanced Encryption Standard**, AES is a symmetric 128-bit block data encryption technique developed by Belgian cryptographers Joan Daemen and Vincent Rijmen. For more information, please refer to http://en.wikipedia.org/wiki/Advanced_Encryption_Standard.

To deploy the UDF and verify them, do the following:

```
jdbc:hive2://> ADD JAR /home/dayongd/Downloads/
. . . . . . .> hiveessentials-1.0-SNAPSHOT.jar;
No rows affected (0.002 seconds)

jdbc:hive2://> CREATE TEMPORARY FUNCTION aesdecrypt AS
. . . . . . .> 'com.packtpub.hive.essentials.hiveudf.AESDecrypt';
No rows affected (0.02 seconds)

jdbc:hive2://> CREATE TEMPORARY FUNCTION aesencrypt AS
```

```
. . . . . . .> 'com.packtpub.hive.essentials.hiveudf.AESEncrypt';
No rows affected (0.015 seconds)

jdbc:hive2://> SELECT aesencrypt('Will') AS encrypt_name
. . . . . . .> FROM employee LIMIT 1;
+---------------------------+
|        encrypt_name       |
+---------------------------+
| YGvo54QIahpb+CVOwv9OkQ==  |
+---------------------------+
1 row selected (34.494 seconds)

jdbc:hive2://> SELECT aesdecrypt('YGvo54QIahpb+CVOwv9OkQ==')
. . . . . . .> AS decrypt_name
. . . . . . .> FROM employee LIMIT 1;
+--------------+
| decrypt_name |
+--------------+
| Will         |
+--------------+
1 row selected (45.43 seconds)
```

Summary

In this chapter, we introduced three main areas for Hive security: authentication, authorization, and encryption. We covered the authentications in Metastore server and HiveServer2. Then, we talked about default, storage-based, and SQL standard-based authorization methods in HiveServer2. At the end of this chapter, we discussed the use of Hive UDF for encryption and decryption. After going through this chapter, we should clearly understand the different areas that will help us address Hive security.

In the next chapter, we'll talk about using Hive with other tools.

Chapter 10. Working with Other Tools

As one of the earliest and most popular SQL over Hadoop tools, Hive has many use cases of working with other tools to offer an end-to-end data intelligence solution. In this chapter, we will discuss the way Hive works with other big data tools in the following areas:

- JDBC / ODBC connector
- HBase
- Hue
- HCatalog
- Zookeeper
- Oozie
- Hive roadmap

JDBC / ODBC connector

JDBC/ODBC is one of the most common ways for Hive to work with other tools. Hadoop vendors, such as Cloudera and Hortonworks, offer free Hive JDBC/ODBC drivers so that Hive can be connected through these drivers; these can be found at the following links:

- For Cloudera, the link is http://www.cloudera.com/content/cloudera/en/downloads/connectors/hive.html
- For Hortonworks, the link is http://hortonworks.com/hdp/addons/

We can use these JDBC/ODBC connectors to connect Hive to tools such as the following:

- A command-line utility such as Beeline, mentioned in Chapter 2, *Setting Up the Hive Environment*
- Integrated development environment such as Oracle SQL Developer, mentioned in Chapter 2, *Setting Up the Hive Environment*
- Data extraction, transformation, loading, and integration tools, such as Talend Open Studio
- Business intelligence reporting tools, such as JasperReports and QlikView
- Data analysis tools such as Microsoft Excel 2013
- Data visualization tools such as Tableau

Since the setup of connectors is very straightforward, please refer to the websites of the preceding tools for more detailed instructions to connect to Hive.

HBase

HBase (see http://hbase.apache.org/) is a high-performance NoSQL key/value store on Hadoop. Hive has offered a storage handler mechanism to integrate with HBase by using the `HBaseStorageHandler` class that creates HBase tables managed by Hive. By integrating Hive with HBase, Hive users can leverage real-time transaction performance of HBase to do real-time big data analysis. Currently, the integration feature is still in progress, especially in the area of offering higher performance and snapshots support. There is another project called Phoenix (see http://phoenix.apache.org/), which provides basic SQL with higher-performance support over HBase.

An example of creating an HBase table in HQL is as follows:

```
CREATE TABLE hbase_table_sample(
id int,
value1 string,
value2 string,
map_value map<string, string>
)
STORED BY 'org.apache.hadoop.hive.hbase.HBaseStorageHandler'
WITH SERDEPROPERTIES ("hbase.columns.mapping" =
":key,cf1:val,cf2:val,cf3:")
TBLPROPERTIES ("hbase.table.name" = "table_name_in_hbase");
```

In this special `CREATE TABLE` statement, the `HBaseStorageHandler` class is delegating interaction with the HBase table with `HiveHBaseTableInputFormat` and `HiveHBaseTableOutputFormat`. The `hbase.columns.mapping` property is required to map each table column defined in the statement to the HBase table columns in order. For example, the ID, by order, maps to the HBase table's rowkey as `:key`. Sometimes, we may need to generate the proper rowkey columns using Hive UDFs if there is no existing column that can be used as a rowkey for the HBase table. The `value1` maps to the `val` column in the `cf1` column family in the HBase table. The Hive MAP data type can be used to access an entire column family. Each row can have a different set of columns, where the column names correspond to the map keys and the column values correspond to the map values, such as the `map_value` columns. The `hbase.table.name` property, which is optional, specifies the table name known by HBase. If it is not provided, the Hive and HBase table will have the same name, such as `hbase_table_sample`.

Note

For more information about configurations and features in progress about Hive-HBase integration, please refer to the Apache Hive wiki at https://cwiki.apache.org/confluence/display/Hive/HBaseIntegration.

Hue

Hue (see http://gethue.com/) is short for **Hadoop User Experience**. It is a web interface for making the Hadoop ecosystem easier to use. For Hive users, Hue offers a unified web interface for easily accessing both HDFS and Hive in an interactive environment. Hue can be installed alone or with the Hadoop vendor packages. In addition, Hue adds more programming-friendly features to Hive, such as the following:

- Highlights HQL keywords
- Autocompletes HQL query
- Offers live progress and logs for Hive and MapReduce jobs
- Submits several queries and checks progress later
- Browses data in Hive tables through a web user interface
- Navigates through the metadata
- Registers UDF and adds files/archives through a web user interface
- Saves, exports, and shares the query result
- Creates various charts from the query result

The following is a screenshot of the Hive editor interface in Hue:

Hue Hive editor user interface

HCatalog

HCatalog (see https://cwiki.apache.org/confluence/display/Hive/HCatalog) is a metadata management system for Hadoop data. It stores consistent schema information for Hadoop ecosystem tools, such as Pig, Hive, and MapReduce. By default, HCatalog supports data in the format of RCFile, CSV, JSON, SequenceFile, ORC file, and a customized format if `InputFormat`, `OutputFormat`, and SerDe are implemented. By using HCatalog, users are able to directly create, edit, and expose (via its REST API) metadata, which becomes effective immediately in all tools sharing the same piece of metadata. At first, HCatalog was a separate Apache project from Hive and was part of Apache Incubator, where most Apache projects first started. Eventually, HCatalog became a part of the Hive project in 2013 starting with Hive 0.11.0.

HCatalog is built on top of the Hive metastore and incorporates support for Hive DDL. It provides read and write interfaces and `HCatLoader` and `HCatStorer`, for Pig, by implementing Pig's load and store interfaces, respectively. HCatalog also provides an interface for MapReduce programs by using `HCatInputFormat` and `HCatOutputFormat`, which are very similar to other customized formats by implementing Hadoop's `InputFormat` and `OutputFormat`. HCatalog provides a REST API from a component called WebHCat so that HTTP requests can be made to access the metadata of Hadoop MapReduce/Yarn, Pig, Hive, and HCatalog DDL from other applications. There is no Hive-specific interface since HCatalog uses Hive's metastore. Therefore, HCatalog can define metadata for Hive directly through its CLI. The HCatalog CLI supports the HQL `SHOW`/`DESCRIBE` statement and the majority of Hive DDL, except the following statements, that require running MapReduce jobs:

- `CREATE TABLE… AS SELECT`
- `ALTER INDEX… REBUILD`
- `ALTER TABLE… CONCATENATE`
- `ALTER TABLE ARCHIVE/UNARCHIVE PARTITION`
- `ANALYZE TABLE… COMPUTE STATISTICS`
- `IMPORT/EXPORT`

ZooKeeper

ZooKeeper (see http://zookeeper.apache.org/) is a centralized service for configuration management and the synchronization of various aspects of naming and coordination. It manages a naming registry and effectively implements a system for managing the various statically and dynamically named objects in a hierarchical system. It also enables coordination and control to the shared resources, such as files and data, which are manipulated by multiple concurrent processes.

Unlike RDBMS, Hive does not natively support concurrency access and locking mechanisms. Hive relies on ZooKeeper for locking the shared resources since Hive 0.7.0. There are two types of locks provided by Hive through Zookeeper and they are as follows:

- **Shared lock**: This is acquired when a table/partition is read. The concurrent shared locks are allowed in Hive.
- **Exclusive lock**: This is acquired for all other operations that modify the table. For partition tables, only a shared lock is acquired if the change is only applicable to the newly-created partitions. An exclusive lock is acquired on the table if the change is applicable to all partitions. In addition, an exclusive lock on the table globally affects all partitions.

Any HQL must acquire proper locks before being allowed to perform corresponding lock-permitted operations.

To enable locking in Hive, we need to make sure ZooKeeper is installed and configured. Then, configure the following properties in Hive's `hive-site.xml` file:

```
<property>
  <name>hive.support.concurrency</name>
  <description>Enable Hive's Table Lock Manager Service</description>
  <value>true</value>
</property>

<property>
  <name>hive.zookeeper.quorum</name>
  <description>Comma separated Zookeeper quorum used by Hive's Table Lock
Manager. </description>
  <value>localhost.localdomain</value>
</property>
```

We can also set the following property to use the new lock manager for transactions support since Hive 0.13.0:

```
<property>
  <name>hive.txn.manager</name>
  <value>org.apache.hadoop.hive.ql.lockmgr.DbTxnManager</value>
</property>
```

Note

Once configured, we can further set locking properties, specified and detailed at https://cwiki.apache.org/confluence/display/Hive/Configuration+Properties#Configuration

Locks are either implicitly acquired/released from HQL or explicitly acquired/released using the `LOCK` and `UNLOCK` statements as follows:

```
--Lock table and specify lock type
jdbc:hive2://> LOCK TABLE employee shared;
No rows affected (1.328 seconds)

--Show the lock information on the specific tables
jdbc:hive2://> SHOW LOCKS employee EXTENDED;
+----------------------------------------------------------------------+-
----+
|                                tab_name                               |
mo  |
+----------------------------------------------------------------------+-
----+
| default@employee                                                     |
SHA |
|  LOCK_QUERYID:hive_20150105170303_792598b1-0ac8-4aad-aa4e-c4cdb0de6697  |
|
|  LOCK_TIME:1420495466554                                             |
|
|  LOCK_MODE:EXPLICIT                                                  |
|
|  LOCK_QUERYSTRING:LOCK TABLE employee shared                         |
|
+----------------------------------------------------------------------+-
----+
5 rows selected (0.576 seconds)

--Release the lock on the table
jdbc:hive2://> UNLOCK TABLE employee;
No rows affected (0.209 seconds)

--Show all locks in the database
jdbc:hive2://> SHOW LOCKS;
+----------+------+
| tab_name | mode |
+----------+------+
+----------+------+
No rows selected (0.529 seconds)

jdbc:hive2://> LOCK TABLE employee exclusive;
No rows affected (0.185 seconds)

jdbc:hive2://> SHOW LOCKS employee EXTENDED;
+----------------------------------------------------------------------+-
----+
|                                tab_name                               |
mo  |
+----------------------------------------------------------------------+-
----+
| default@employee                                                     |
EXC |
|  LOCK_QUERYID:hive_20150105170808_bbc6db18-e44a-49a1-bdda-3dc30b5c8cee  |
```

```
|
| LOCK_TIME:1420495807855                                                      |
|
| LOCK_MODE:EXPLICIT                                                           |
|
| LOCK_QUERYSTRING:LOCK TABLE employee exclusive                               |
|
+------------------------------------------------------------------------+-
----+
5 rows selected (0.578 seconds)

jdbc:hive2://> SELECT * FROM employee;
```

When the table acquires an exclusive lock, the preceding SELECT statement will wait for the lock and show nothing as a result set unless we unlock the table in the other session. From the Hive log, we can find the following information that specifies that the SELECT statement is waiting to get the read lock:

```
15/01/05 17:13:39 INFO ql.Driver: <PERFLOG method=acquireReadWriteLocks>
15/01/05 17:13:39 ERROR ZooKeeperHiveLockManager: conflicting lock present
for default@employee mode SHARED
```

Note

For more information about using ZooKeeper for Hive locks, please refer to the Apache Hive wiki at https://cwiki.apache.org/confluence/display/Hive/Locking.

Oozie

Oozie (see http://oozie.apache.org/) is an open source workflow coordination and schedule service to manage data processing jobs. Oozie workflow jobs are defined in a series of nodes in a **Directed Acyclical Graph** (**DAG**). Acyclical here means that there are no loops in the graph and all nodes in the graph flow in one direction without going back. Oozie workflows contain either the control flow node or action node:

- **Control flow node**: This either defines the start, end, and failed node in a workflow or controls the workflow execution path such as decision, fork, and join nodes.
- **Action node**: This defines the core data processing action job such as MapReduce, Hadoop filesystem, Hive, Pig, Java, Shell, e-mail, and Oozie subworkflows. Additional types of actions are also supported by developing extensions.

Oozie is a scalable, reliable, and extensible system. It can be parameterized for workflow submission and scheduled to run automatically. Therefore, Oozie is very suitable for lightweight data integration or maintenance jobs.

Hue offers very friendly and powerful support for Oozie through the Oozie editor. Creating and submitting an Oozie workflow of Hive actions from Hue is as straightforward as the following steps:

1. Log in to Hue and select from the top menu bar **Workflows** | **Editors** | **Workflows** to open **Workflow Manager**.
2. Click on the **Create** button to create a workflow.
3. Give a proper workflow name and save the workflow.
4. Once the workflow is saved, the Oozie editor window appears for further settings.
5. Drag a **Hive** action to the middle of the start and end nodes.
6. In the **Edit Node:** menu shown, the following settings are present. Provide proper settings as follows:

   - **Name**: Give a proper action name.
   - **Description**: This is where to describe the job. This is optional.
   - **Advanced**: This is for SLA monitoring. This is optional.
   - **Script name**: Choose the HQL scripts from HDFS for Hive action.
   - **Prepare**: Define actions, such as delete files or create folders, before running the script. This is optional.
   - **Parameters**: This defines the parameters to be taken when submitting the job (such as `${date}`). This is optional.
   - **Job properties**: This is where to set Hadoop/Hive properties. This is optional.
   - **Files**: This is where to select the files needed for the scripts. This is optional.
   - **Archives**: This is where to select the archive files such as UDF JARs. This is optional.
   - **Job XML**: Choose a copy of the `hive-site.xml` file of the Hive cluster from HDFS so that Oozie can connect to the Hive metastore.

7. Click on **Done** in the **Edit Node:** menu and then click on **Save** in **Workflow Editor**.

8. Click on **Submit** to submit the workflow. Then, the Hive action is triggered by the Oozie workflow successfully.

Hive roadmap

As it is the end of this chapter as well as of this book, the highlight of each Hive release milestone and future features expected are summarized as follows along with best wishes to the Hive communities for growing bigger and better in the near future:

- December 2011 – Hive 0.8.0

    - Added `Bitmap` indexes
    - Added the `TIMESTAMP` data type
    - Added the Hive Plugin Developer Kit to make plugin building and testing easier
    - Improved JDBC Driver and bug fixes

- April 2012 – Hive 0.9.0

    - Added the `CREATE OR REPLACE VIEW` statement
    - Added `NOT IN` and `NOT LIKE` support
    - Added the `BETWEEN` and `NULL` safe equality operator
    - Added `printf()`, `sort_array()`, and `concat_ws()` functions
    - Added a filter push-down from Hive into HBase for the key column
    - Combined multiple `UNION ALL` statements in one MapReduce job
    - Combined multiple `GROUP BY` statements on the same data with the same keys in one MapReduce job

- January 2013 – Hive 0.10.0

    - Added the `CUBE` and `ROLLUP` statements
    - Added better support for YARN
    - Added more information in the `EXPLAIN` statement
    - Added the `SHOW CREATE TABLE` statement
    - Added built-in support for reading/writing Avro data
    - Added improvements for skewed joins
    - Improved simple queries without running MapReduce jobs faster

- May 2013 – Hive 0.11.0 as Stinger Phase 1

    - Added ORC for better performance
    - Added analytic and windows functions
    - Added HCatalog as part of Hive
    - Added `GROUP BY` column positions
    - Improved data types and added the `DECIMAL` data type
    - Improved joins for broadcast and SMB joins
    - Implemented HiveServer2

- October 2013 – Hive 0.12.0 as Stinger Phase 2

    - Added `VARCHAR` and `DATE` support
    - Added parallel `ORDER BY` to Hive
    - Added more improvements for `ORC`, such as predicate push-down
    - Added a correlation optimizer

- Added support for `GROUP BY` on the `STRUCT` type
    - Added support for the outer lateral view
    - Pushed `LIMIT` down to mappers

- April 2014 – Hive 0.13.0 as Stinger Phase 3 Final

    - Added `DECIMAL` and `CHAR` data types
    - Added support for running jobs on Tez
    - Added a vectorized query engine
    - Added support for subqueries for `IN`, `NOT IN`, `EXISTS`, and `NOT EXISTS`
    - Added support for permanent functions
    - Added support for common table expressions
    - Added SQL standard-based authentication

- November 2014 – Hive 0.14.0 as Stinger.next Phase 1

    - Added transactions with ACID semantics
    - Added a **Cost Base Optimizer** (**CBO**)
    - Added the `CREATE TEMPORARY TABLE` statement
    - Added support for the `STORED AS AVRO` in the `CREATE TABLE` statement
    - Added `skipTrash` configuration for the `DROP TABLE` statement
    - Added `AccumuloStorageHandle`
    - Used Tez autoparallelism in Hive

- February 2015 – Hive 1.0.0

    - Moved to a 1.x.y release naming structure
    - Made HiveMetaStoreClient a public API
    - Removal of HiveServer1
    - Switched to Tez 0.5.2

- Future

    - Offer subsecond queries with **Live Long And Process** (**LLAP**)
    - Offer Hive over Spark
    - Support SQL 2011 analytics
    - Support cross-geo queries
    - Offer materialized views
    - Offer workload management via YARN and LLAP integration
    - Hive as a unified data query tool

Summary

In this final chapter, we introduced some big data tools, which can work with Hive through JDBC or ODBC integration, such as HBase, Hue, HCatalog, ZooKeeper, and Oozie. Then, we reviewed the key releases of Hive from 0.8.0 to 1.0.0, as well as the exciting features expected in the future. After going through this chapter, we should understand how to use other big data tools with Hive to provide end-to-end data intelligence solutions.

Index

A

B

C

D

E

F

G

H

I

J

K

- Kerberos
  - about / [Authentication](#)
- Kerberos authentication / [HiveServer2 authentication](#)
- Key Distribution Center (KDC) / [Authentication](#)

L

M

N

O

P

- parallel execution, job and query optimization / [Parallel execution](#)
- ParquetHiveSerDe / [SerDe](#)
- parser and search tips / [Operators and functions](#)
- PARTITION BY statement / [Analytic functions](#)
- partitions
    - about / [Hive partitions](#)
- partition tables
    - by date and time / [Partition tables](#)
    - by locations / [Partition tables](#)
    - by business logics / [Partition tables](#)
- personal identity information (PII)
    - about / [Encryption](#)
- Phoenix
    - URL / [HBase](#)
- Pluggable Authentication Modules (PAM) authentication / [HiveServer2 authentication](#)
- pluggable custom authentication / [HiveServer2 authentication](#)
- PostgreSQL
    - URL / [Installing Hive from Apache](#)
- Presto
    - URL / [A short history](#)
- primitive type conversion / [Data type conversions](#)
- Processing Elements (PE) / [Batch, real-time, and stream processing](#)

R

S

T

U

V

W

Y

Z