



C o m m u n i t y E x p e r i e n c e D i s t i l l e d

Learning Cloudera Impala

Perform interactive, real-time in-memory analytics on large amounts of data using the massive parallel processing engine Cloudera Impala

Avkash Chauhan

[PACKT] open source*
PUBLISHING community experience distilled

Learning Cloudera Impala

Perform interactive, real-time in-memory analytics on large amounts of data using the massive parallel processing engine Cloudera Impala

Avkash Chauhan



BIRMINGHAM - MUMBAI

Learning Cloudera Impala

Copyright © 2013 Packt Publishing

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

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

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

First published: December 2013

Production Reference: 1181213

Published by Packt Publishing Ltd.
Livery Place
35 Livery Street
Birmingham B3 2PB, UK.

ISBN 978-1-78328-127-5

www.packtpub.com

Cover Image by Vivek Sinha (vs@viveksinha.com)

Credits

Author

Avkash Chauhan

Project Coordinator

Sherin Padayatty

Reviewers

Salman Ahmed

Charles Menguy

Proofreader

Lawrence A. Herman

Acquisition Editors

Pramila Balan

Joanne Fitzpatrick

Indexer

Monica Ajmera Mehta

Commissioning Editor

Sharvari Tawde

Graphics

Ronak Dhruv

Yuvraj Mannari

Technical Editors

Kapil Hemnani

Faisal Siddiqui

Production Coordinator

Arvindkumar Gupta

Cover Work

Arvindkumar Gupta

Copy Editors

Alisha Aranha

Roshni Banerjee

Mradula Hegde

Dipti Kapadia

Aditya Nair

Deepa Nambiar

Adithi Shetty

About the Author

Avkash Chauhan is a software technology veteran with more than 12 years of industry experience in various disciplines such as embedded engineering, cloud computing, big data analytics, data processing, and data visualization. He has an extensive global work experience with Fortune 100 companies worldwide. He has spent the last eight years at Microsoft before moving on to Silicon Valley to work with a big data and analytics start-up. He started his career as an embedded engineer; and during his eight-year long gig at Microsoft, he worked on Windows CE, Windows Phone, Windows Azure, and HDInsight. He spent several years working with the Windows Azure team to develop world-class cloud technology, and his last project was Apache Hadoop on Windows Azure, also known as HDInsight. He worked on the HDInsight project since its incubation at Microsoft, and helped its early development and then deployment on cloud. For the past three years, he has been working on big data- and Hadoop-related technologies by developing applications to make Hadoop easy to use for large- and mid-market companies. He is a prolific blogger and very active on the social networking sites. You can directly contact him through the following:

- **LinkedIn:** <https://www.linkedin.com/in/avkashchauhan>
- **Blog:** <http://cloudcelebrity.wordpress.com/>
- **Twitter:** @avkashchauhan

I would like to thank my wife, two little kids, family, and friends for their continuous love and immense support in completing this book.

About the Reviewer

Charles Menguy is a software engineer working in New York City for Adobe Systems, whose primary focus is dealing with enormous amounts of data. He holds a Master's degree in Computer Science, with a major in Artificial Intelligence. He is passionate about all things related to big data, data science, and cloud computing. As a certified Hadoop developer from Cloudera, he has been working with various technologies in the Hadoop stack. He contributes back to the community by being an avid user of StackOverflow.

You can add him to your LinkedIn contacts at <http://www.linkedin.com/in/charlesmenguy/>, write to him at menguy.charles@gmail.com, or learn more about him at <http://cmenguy.github.io/>.

www.PacktPub.com

Support files, eBooks, discount offers and more

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

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

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



<http://PacktLib.PacktPub.com>

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

Why Subscribe?

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

Free Access for Packt account holders

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

Table of Contents

Preface	1
Chapter 1: Getting Started with Impala	7
Impala requirements	9
Dependency on Hive for Impala	10
Dependency on Java for Impala	10
Hardware dependency	10
Networking requirements	11
User account requirements	11
Installing Impala	11
Installing Impala with Cloudera Manager	11
Installing Impala without Cloudera Manager	13
Configuring Impala after installation	14
Starting Impala	15
Stopping Impala	16
Restarting Impala	16
Upgrading Impala	16
Upgrading Impala using parcels with Cloudera Manager	17
Upgrading Impala using packages with Cloudera Manager	17
Upgrading Impala without Cloudera Manager	18
Impala core components	18
Impala daemon	19
Impala statestore	19
Impala metadata and metastore	20
The Impala programming interface	20
The Impala execution architecture	21
Working with Apache Hive	21
Working with HDFS	22
Working with HBase	22

Impala security	22
Authorization	23
The SELECT privilege	23
The INSERT privilege	23
The ALL privilege	23
Authentication through Kerberos	24
Auditing	24
Impala security guidelines for a higher level of protection	25
Summary	26
Chapter 2: The Impala Shell Commands and Interface	27
Using Cloudera Manager for Impala	27
Launching Impala shell	29
Connecting impala-shell to the remotely located impalad daemon	30
Impala-shell command-line options with brief explanations	30
General command-line options	31
Connection-specific options	32
Query-specific options	33
Secure connectivity-specific options	34
Impala-shell command reference	34
General commands	35
Query-specific commands	36
Table- and database-specific commands	38
Summary	38
Chapter 3: The Impala Query Language and Built-in Functions	39
Impala SQL language statements	40
Database-specific statements	41
The CREATE DATABASE statement	41
The DROP DATABASE statement	41
The SHOW DATABASES statement	42
Using database-specific query sentence in an example	42
Table-specific statements	43
The CREATE TABLE statement	43
The CREATE EXTERNAL TABLE statement	44
The ALTER TABLE statement	44
The DROP TABLE statement	45
The SHOW TABLES statement	45
The DESCRIBE statement	45
The INSERT statement	47
The SELECT statement	47
Internal and external tables	48
Data types	48
Operators	52
Functions	55

Clauses	57
Query-specific SQL statements in Impala	60
Defining VIEWS in Impala	61
Loading data from HDFS using the LOAD DATA statement	62
Comments in Impala SQL statements	62
Built-in function support in Impala	63
The type conversion function	65
Unsupported SQL statements in Impala	65
Summary	66
Chapter 4: Impala Walkthrough with an Example	67
Creating an example scenario	67
Example dataset one – automobiles (automobiles.txt)	68
Example dataset two – motorcycles (motorcycles.txt)	68
Data and schema considerations	69
Commands for loading data into Impala tables	69
HDFS specific commands	69
Loading data into the Impala table from HDFS	70
Launching the Impala shell	72
Database and table specific commands	72
SQL queries against the example database	74
SQL join operation with the example database	77
Using various types of SQL statements	77
Summary	79
Chapter 5: Impala Administration and Performance Improvements	81
Impala administration	81
Administration with Cloudera Manager	82
The Impala statestore UI	84
Impala High Availability	84
Single point of failure in Impala	85
Improving performance	85
Enabling block location tracking	85
Enabling native checksumming	86
Enabling Impala to perform short-circuit read on DataNode	86
Adding more Impala nodes to achieve higher performance	87
Optimizing memory usage during query execution	87
Query execution dependency on memory	87
Using resource isolation	87
Testing query performance	88
Benchmarking queries	88

Verifying data locality	88
Choosing an appropriate file format and compression type for better performance	89
Fine-tuning Impala performance	90
Partitioning	90
Join queries	90
Table and column statistics	91
Summary	92
Chapter 6: Troubleshooting Impala	93
Troubleshooting various problems	93
Impala configuration-related issues	93
The block locality issue	94
Native checksumming issues	94
Various connectivity issues	94
Connectivity between Impala shell and Impala daemon	94
ODBC/JDBC-specific connectivity issues	95
Query-specific issues	96
Issues specific to User Access Control (UAC)	97
Platform-specific issues	97
Impala port mapping issues	97
HDFS-specific problems	98
Input file format-specific issues	98
Using Cloudera Manager to troubleshoot problems	98
Impala log analysis using Cloudera Manager	99
Using the Impala web interface for monitoring and troubleshooting	101
Using the Impala statestore web interface	102
Using the Impala Maintenance Mode	103
Checking Impala events	104
Summary	104
Chapter 7: Advanced Impala Concepts	105
Impala and MapReduce	105
Impala and Hive	106
Key differences between Impala and Hive	106
Impala and Extract, Transform, Load (ETL)	106
Why Impala is faster than Hive in query processing	107
Impala processing strategy	108
Impala and HBase	108
Using Impala to query HBase tables	109
File formats and compression types supported in Impala	110
Processing different file and compression types in Impala	111
The regular text file format with Impala tables	113

The Avro file format with Impala tables	114
The RCFile file format with Impala tables	114
The SequenceFile file format with Impala tables	115
The Parquet file format with Impala tables	115
The unsupported features in Impala	116
Impala resources	117
Summary	117
Appendix: Technology Behind Impala and Integration with	
Third-party Applications	119
Technology behind Impala	119
Data visualization using Impala	120
Tableau and Impala	121
Microsoft Excel and Impala	122
Microstrategy and Impala	123
Zoomdata and Impala	124
Real-time query with Impala on Hadoop	125
Real-time query subscriptions with Impala	125
What is new in Impala 1.2.0 (Beta)	126
Index	127

Preface

The changing landscape of Big Data and tools created for a relevant understanding of it have become very crucial in today's tech industry. The ability to understand and familiarize with such tools allow individuals to creatively and intelligently take decisions with precision. If you've always wanted to crunch billions of rows of raw data on Hadoop in a couple of seconds, Cloudera Impala is, hands down, the top choice for you. Cloudera Impala provides a way to ingest various formats of data stored on Hadoop and provides a query engine to process it for gaining extremely important insight.

In this book, *Learning Cloudera Impala*, you are going to learn everything you need to know about Cloudera Impala so that you can start your project. The book covers Cloudera Impala from installation, administration, and query processing, all the way up to connectivity with other third-party applications. With this book in your hand, you will find yourself empowered to play with your data in Hadoop, and getting insight from your data will look like an interesting game to you.

What this book covers

Chapter 1, Getting Started with Impala, covers information on Impala, its core components, and its inner workings in details. We will cover the Impala execution architecture, including daemon and statestore, and how they interact together with the other components. Impala metadata and metastore are also discussed here to explain how Impala maintains its information. Finally, we will study various ways to interface Impala.

Chapter 2, The Impala Shell Commands and Interface, explains the various command options to interact with Impala, mainly using command-line references. In this chapter, we have covered the Impala command-line interface, explaining various ways Impala shell can connect to Impala daemon. Once the connection between Impala shell and *impalad* is established, we can use the various commands we discussed to connect to Impala.

Chapter 3, The Impala Query Language and Built-in Functions, teaches us how to make great use of Impala shell to interact with data by using the Impala Query Language, which is based on SQL, while providing a great degree of compatibility with HiveQL. Hive statements are based on SQL statements, and because Impala statements are based on SQL, we will learn several similarities and differences between them. Along with the Impala Query Language, we will also learn various Impala built-in functions using great examples.

Chapter 4, Impala Walkthrough with an Example, covers most of the learning from the previous chapter in detail. This way you can see a real-world scenario used with Impala and understand how and where to use Impala statements in real-world applications. I have created this detailed example by first creating automobile-specific datasets, and then using most of the SQL statements with the built-in functions we discussed in the previous chapter.

Chapter 5, Impala Administration and Performance Improvements, covers two important topics, Impala administration and performance improvements. Within the Impala administration section, I will first show you how you can administer Impala using Cloudera Manager. After that, I will teach you how to verify Impala-specific information for its correctness using a debugging web server. We will see Impala logs and Impala daemons through the statestore UI. The next part of Impala admin is about Impala High Availability, where we will learn the key traits for keeping Impala running in the event of a problem.

Chapter 6, Troubleshooting Impala, teaches you how to troubleshoot various Impala issues in different categories. Besides troubleshooting, in the latter part, I will show you how to utilize Impala logging to learn more about Impala execution, query processing, and possible issues. My objective is to provide you with some critical information on troubleshooting and log analysis, so you can manage the Impala cluster effectively and make it useful for yourself and your team.

Chapter 7, Advanced Impala Concepts, teaches you more about Impala; however, this information is more advance in nature to help you excel in data processing your project through Impala. I have described how Impala works side by side with MapReduce, without using it in the same cluster. I have also explained why Impala has an edge over Hive, even when using Hive as a key component, on which Impala is dependent. Finally, we cover details on using HBase with Impala and processing various Big Data input files on Hadoop with Impala.

Appendix, Technology Behind Impala and Integration with Third-party Applications, covers the detailed technology behind Impala and real-time query concepts with Impala. I have also described a few third-party data visualization applications, from Tableau, Zoomdata, and Microsoft Excel to Microstrategy, which connect with Impala to provide effective data visualization.

What you need for this book

You must have a Hadoop cluster (single-node experimental or multinode production) up and running to install Impala on it or already have Impala installed on it. Cloudera CDH 4.3 or above is preferred to install Impala. If you decide to install Cloudera Impala in your Hadoop Cluster, you can download it from the following link:

<https://www.cloudera.com/content/support/en/downloads/download-components.html>

If you do not have an active Hadoop cluster and still want to learn and try Impala, you have the option of downloading a Cloudera QuickStart Virtual Machine including everything from Cloudera, at the following link:

<https://www.cloudera.com/content/support/en/downloads.html>

Who this book is for

The book, is for those who really want to take full advantage of their Hadoop cluster by processing extremely large amounts of raw data in Hadoop at real-time speed. You may be using Hadoop as your raw data storage medium or using Hive to process your data. You will learn everything you need to start using Impala, to make the best use of your Hadoop cluster, and leverage any Business Intelligence tools you have in order to gain insight from your data using Impala.

Conventions

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

Code words in text, database table names, folder names, filenames, file extensions, pathnames, dummy URLs, user input, and Twitter handles are shown as follows:

"Copy `hdfs-site.xml` and `core-site.xml` from Hadoop cluster to each Impala node into the Impala configuration folder, `/etc/impala/conf`."

Keywords in the text are shown as follows: "Impala statements support data manipulation statements similar to **DML (Data Manipulation Language)**."

Impala shell commands or Impala SQL statements are written as follows:

```
CREATE TABLE table_name (def data_type)
PARTITIONED BY (partiton_name partition_type);
ALTER TABLE table_name ADD PARTITION
(partition_type='definition');
```

When an Impala command or Impala SQL statement is used to show an example, either console output or query output is also displayed for complete understanding. In this scenario, either command or query is shown in bold as follows:

```
[Hadoop.testdomain:21000] > select count(distinct(make)) from
automobiles;
Query finished, fetching results ...
+-----+
| count(distinct make) |
+-----+
| 10                    |
+-----+
Returned 1 row(s) in 0.48s
```

Another example is as follows:

```
[cloudera@localhost ~]$ hdfs dfs -ls /user/cloudera/automobiles/
Found 1 items
-rw-r--r--    3 cloudera cloudera      985 2013-10-15 19:17
  /user/cloudera/automobiles/automobiles.txt
```

Reader feedback

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

To send us general feedback, simply send an e-mail to feedback@packtpub.com, and mention the book title through the subject of your message.

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

Customer support

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

Errata

Although we have taken every care to ensure the accuracy of our content, mistakes do happen. If you find a mistake in one of our books – maybe a mistake in the text or the code – we would be grateful if you would report this to us. By doing so, you can save other readers from frustration and help us improve subsequent versions of this book. If you find any errata, please report them by visiting <http://www.packtpub.com/support>, 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.

Piracy

Piracy of copyright material on the Internet is an ongoing problem across all media. At Packt Publishing, 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

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

1

Getting Started with Impala

This chapter covers the information on Impala, its core components, and its inner workings in detail. We will cover Impala architecture including Impala daemon, statestore, and execution model, and how they interact together along with other components. Impala metadata and metastore are also discussed here, to understand how Impala maintains its information. Finally, we will study various ways to interface Impala.

The objective of this chapter is to provide enough information for you to kick-start Impala on a single node experimental or multimode production cluster. This chapter covers the Impala essentials within the following broad categories:

- System requirement
- Installation
- Configuration
- Upgradation
- Security
- Impala architecture and execution

Impala is for a new breed of data wranglers who want to process the data at lightening-fast speed using traditional SQL knowledge. Impala provides data analysts or scientists a way to access data, which is stored on Hadoop at lightening speed by directly using SQL or other Business Intelligence tools. Impala uses the Hadoop data processing layer, also called HDFS, to process the data so there is no need to migrate data from Hadoop to any other middleware, specialized system, or data warehouse. Impala provides data wranglers a **Massively Parallel Processing (MPP)** query engine, which runs natively on Hadoop.

Native on Hadoop means the engine runs on Hadoop and uses the Hadoop core component, HDFS, along with other additional components, such as Hive and HBase. To process data, Impala has its own execution component, which runs on each DataNode where the data is stored in blocks. There is a list of third-party applications that can directly process data stored on Hadoop through Impala. The biggest advantage of Impala is that data transformation or data movement is not required for data stored on Hadoop. No data movement means all the processing is happening where the data resides in the cluster. In other distributed systems, data is transferred over the network before it is processed; however, with Impala the processing happens at the place where data is stored, which is one of the premier reasons why Impala is very fast in comparison to other large data processing systems.

Before we learn more about Impala, let's see what the key Impala features are:

- First and foremost, Impala is 100% open source under the Apache license
- Impala is a native MPP engine, running on the Cloudera Hadoop distribution
- Impala supports in-memory processing for data through SQL-like queries
- Impala uses **Hadoop Distributed File System (HDFS)** and HBase
- Impala supports integration with leading Business Intelligence tools, such as Tableau, Pentaho, Microstrategy, Zoomdata, and so on
- Impala supports a wide variety of input file formats, that is, regular text files, files in CSV/TSV or other delimited format, sequence files, Avro, RCFile, LZO, and Parquet types
- For third-party application connectivity, Impala supports ODBC drive, SQL-like syntax, and Beeswax GUI (in Apache Hue) from Apache Hive
- Impala uses Kerberos authentication and role-based authorization with Sentry

The key benefits of using Impala are:

- Impala uses Hive to read a table's metadata; however, using its own distributed execution engine it makes data processing very fast. So the very first benefit of using Impala is the super fast access of data from HDFS.
- Impala uses a SQL-like syntax to interact with data, so you can leverage the existing BI tools to interact with data stored on Hadoop. The engineers with SQL expertise can benefit from Impala as they do not need to learn new languages and skills. Additionally, Impala offers higher performance and execution speed.
- While running on Hadoop, Impala leverages the Hadoop file and data format, metadata, resource management, and security, all available on Hadoop.

- As Impala interacts with the stored data in Hadoop, it preserves full fidelity of data while analyzing the data, due to aggregations or conformance of fixed schemas.
- Impala performs interactive analysis directly on the data stored on Hadoop DataNodes without requiring data movement, which results in lightening-fast query results, because there are no network bottlenecks and the time available to move data is zero.
- Impala provides a single repository and metadata store from source to analysis, which enables more users to interact with a large amount of data. The presence of a single repository also reduces data movement, which helps in performing interactive analysis directly on full fidelity data.

Impala requirements

Impala is supported on 64-bit Linux-based operating systems. At the time of writing this book, Impala was supported on the following operating systems:

- Red Hat Enterprise Linux 5.7/6.2/6.4
- CentOS 5.7/6.2/6.4
- SLES 11 with SP 1 or newer
- Ubuntu 10.04/12.04
- Debian 6.03

As Impala runs on Hadoop, it is also important to discuss the supported Hadoop version. At the time of writing this book, Impala was supported on the following Hadoop distributions:

- Impala 1.1 and 1.1.1
 - Cloudera Hadoop CDH 4.1 or later
- Impala 1.0
 - ClouderaHadoopCDH 4.1 or later
- Impala 0.7 and older
 - Cloudera Hadoop CDH 4.1 only

Besides CDH, Impala can run on other Hadoop distributions by compiling the source code and then configuring it correctly as required.



Depending on the latest version of Impala, requirements might change, so please visit the Cloudera Impala website for updated information.

Dependency on Hive for Impala

Even though the common perception is that Impala needs Hive to function, it is not completely true. The fact is that only the Hive metastore is required for Impala to function and Hive can be installed on some other client machine. Hive doesn't require being installed on the same DataNode where Impala is installed, because as long as Impala can access the Hive metastore, it will function as expected. In brief, the Hive metastore stores tables and partitions' specific information, which is also called metadata.

As Hive uses PostgreSQL or MySQL for the Hive metastore, we can also consider that either PostgreSQL or MySQL is required for Impala.

Dependency on Java for Impala

For those who don't know, Impala is written in C++. However, Impala uses Java to communicate with various Hadoop components. In Impala, the `impala-dependencies.jar` file located at `/usr/lib/impala/lib` includes all the required Java dependencies. Oracle JVM is the officially supported JVM for Impala and other JVMs might cause problems while running Impala.

Hardware dependency

The source datasets processed by Impala, along with join operations, could be very large, and because processing is done in the memory, as an Impala user you must make sure that you have sufficient memory to process the join operations. The memory requirement is based on your source dataset requirement, which you are going to process through Impala. You also know that Impala cannot run queries that have a working set greater than the maximum available RAM. In a case when memory is not sufficient, Impala will not be able to process the query and the query will be canceled.

For best performance with Impala, it is suggested to have DataNodes with multiple storage disks because disk I/O speed is often considered the bottleneck for Impala performance. The total amount of physical storage requirement is based on the source data, which you would want to process with Impala.

As Impala uses the SSE4.2 CPU instructions set, which is mostly found in the latest processors, the latest processors are often suggested for better performance with Impala.

Networking requirements

Impala daemons running in DataNodes can process data stored in local nodes as well as in remote nodes. To achieve the highest performance, it is advised that Impala attempts to complete data processing on the local data instead of remote data using a network connection. To achieve local data processing, Impala matches the hostname provided to each Impala daemon with the IP address of each DataNode by resolving the hostname flag to an IP address. For Impala to work with the local data stored in a DataNode, you must use a single IP interface for the DataNode and an Impala daemon on each machine. Since there is a single IP address, make sure that the Impala daemon hostname flags resolve the IP address of the DataNode.

User account requirements

When Impala is installed, a user name `impala` and group name `impala` is created, and Impala uses this username and group name during its life after installation. You must ensure that no one changes the `impala` group and user settings, and also no other application or system activity obstructs the functionality of the `impala` user and group. To achieve the highest performance, Impala uses direct reads and, because a root user cannot do direct reads, Impala is not executed as root. To achieve full performance with Impala, the user must make sure that Impala is not running as a root user.

Installing Impala

As Impala is designed and developed to run on the Cloudera Hadoop distribution, there are two different ways Impala can be installed on supported Cloudera Hadoop distributions. Both installation methods are described in a nutshell, as follows.

Installing Impala with Cloudera Manager

Cloudera Manager is only available for the Cloudera Hadoop distribution. The biggest advantage of installing Impala using Cloudera Manager is that most of the complex configuration is taken care of by Cloudera Manager, and applies to all depending applications, if applicable. Cloudera Manager has various versions available; however, to support specific Impala versions, the user must have a proper Cloudera Manager for successful installation.

Once previously described requirements are met, using Cloudera Manager can help you install Impala. Depending on the Cloudera Manager version, you can install specific Impala versions. For example, to install Impala version 1.1.1 you would need Cloudera Manager 4.7 or a higher version, which supports all the features and the auditing feature introduced in Impala 1.1.1. Just use the Cloudera Manager UI to install Impala from the list and follow the instructions as they appear. As shown in the following Cloudera Manager UI screenshot, I have Impala 1.1.1 installed; however, I can upgrade to Impala 1.2.1 just using Cloudera Manager.



To learn more about the installation of Cloudera Manager, please visit the Cloudera documentation site at the following link, which will give you the updated information:

<http://www.cloudera.com/content/cloudera-content/cloudera-docs/Impala/latest/Cloudera-Impala-Release-Notes/Cloudera-Impala-Release-Notes.html>

The screenshot shows the Cloudera Manager interface. At the top, there's a navigation bar with 'Services', 'Hosts', 'Activities', 'Diagnose', 'Audits', 'Charts', and 'Administration'. Below this, the 'All Hosts' page is displayed. Under the 'Parcels' tab, there's a 'Downloadable' section with three parcels: SOLR 0.9.3-1.cdh4.3.0.p0.366, IMPALA 1.2.1-1.p0.8 50, and CDH 4.5.0-1.cdh4.5.0.p0.30. Each parcel has a 'Download' button and a status indicator (1, 2, 3). Below the 'Downloadable' section, there's a 'Cluster 1 - CDH4' section with two activated parcels: IMPALA 1.1.1-1.p0.1 7 and CDH 4.4.0-1.cdh4.4.0.p0.39. Each activated parcel has an 'Actions' button.

Installing Impala without Cloudera Manager

If you decide to install Impala on your own in your Cloudera Hadoop cluster, you must make sure that basic Impala requirements are met and necessary components are already installed. First you must have the correct version of the Cloudera Hadoop cluster ready depending on your Impala version, and have the Hive metastore installed either using MySQL or PostgreSQL.

Once you have made sure that the Hive metastore is available in your Cloudera Hadoop cluster, you can start the Impala installation to all DataNodes as follows:

- Make sure that you have Cloudera public repo set in your OS, so Impala specific packages can be downloaded and installed on your machine. If you do not have the Cloudera specific public repo set, please visit the Cloudera website to get your OS specific information.
- After that, you will need to install the following three packages on your machine:
 - Impala
 - Impala-server
 - Impala-state-store
- Then, copy `hive-site.xml`, `core-site.xml`, and `hdfs-site.xml` Hadoop configuration files to the `/etc/impala/conf` folder, which is the Impala configuration folder.
- As per Cloudera advice, it is not a good choice to install Impala in Namenode, so please do not do so, because any problem caused by Impala may bring your Hadoop cluster down.
- Finally, install Impala shell to a single DataNode or a network-connected external machine on which you have decided to run queries.



Impala is also compiled and tested to run on the MapR Hadoop distribution, so if you are interested in running Impala on MapR, please visit the following link:

<http://doc.mapr.com/display/MapR/Impala>

Configuring Impala after installation

After Impala is installed, you must perform a few mandatory and recommended configuration settings for smooth Impala operations. Cloudera Manager does some of the configurations automatically; however, a few of them need to be completed after any kind of installation. The following is a list of post-installation configurations:

- On Cloudera Hadoop CDH 4.2 or newer distribution, the user must enable short-circuit reads on each DataNode, after each type of installation. To enable short-circuit reads, here are the steps to follow on your Cloudera Hadoop cluster:
 1. First configure `hdfs-site.xml` in each DataNode as follows:

```
<property>
  <name>dfs.client.read.shortcircuit</name>
  <value>true</value>
</property>
<property>
  <name>dfs.domain.socket.path</name>
  <value>/var/run/hadoop-hdfs/dn._PORT</value>
</property>
<property>
  <name>dfs.client.file-block-storage-
    locations.timeout</name>
  <value>3000</value>
</property>
```
 2. If `/var/run/Hadoop-hdfs/` is group writable, make sure its group is the root.
 3. Copy `core-site.xml` and `hdfs-site.xml` from the Hadoop configuration folder to the Impala configuration folder at `/etc/impala/conf`.
 4. Restart all DataNodes.

- Cloudera Manager enables "block location tracking" and "native checksumming" for optimum performance; however, for independent installation both of these have to be enabled. Enabling block location metadata allows Impala to know on which disk data blocks are located, allowing better utilization of the underlying disks. Both "block location tracking" and "native checksumming" are described in later chapters for better understanding. Here is what you can do to enable block location tracking:
 1. `hdfs-site.xml` on each DataNode must have the following setting:

```
<property>
  <name>dfs.datanode.hdfs-blocks-metadata.enabled</name>
  <value>true</value>
</property>
```
 2. Make sure the updated `hdfs-site.xml` file is placed in the Impala configuration folder at `/etc/impala/conf`.
 3. Restart all DataNodes.
- Enabling native checksumming causes Impala to use an optimized native library for computing checksums if that library is available. If Impala is installed using Cloudera Manager, "native checksumming" is automatically configured and no action is needed. However, if you need to enable native checksumming on your self installed Impala cluster, you must build and install the `libhadoop.so` Hadoop Native Library. If this library is not available, you might receive the **Unable to load native-hadoop library for your platform... using built-in-java classes where applicable** message in Impala logs, indicating that native checksumming is not enabled.

Starting Impala

If you have used Cloudera Manager to install Impala, then you can use the Cloudera Manager UI to start/shutdown Impala. However, those who installed Impala directly need to start at least one instance of Impala-state-store and Impala on all DataNodes where it is installed. In this scenario, you can either use init scripts or you can start the statestore and Impala directly. Impala uses Impala-state-store to run in the distributed mode. Impala-state-store helps Impala to achieve the best performance; however, if the state store becomes unavailable, Impala continues to function.

To start the Impala-state-store, use the following command:

```
$ sudo service impala-state-store start
```

To start Impala on each DataNode, use the following command:

```
$ sudo service impala-server start
```

Impala-state-store and Impala server-specific init scripts are located at `/etc/default/impala`, which can be edited if necessary when you want to automate or start these services depending on certain conditions.

Stopping Impala

To stop Impala services in all nodes where it is installed, use the following command:

```
$sudo service impala-server stop
```

To stop any instances of Impala-state-store in the Hadoop Cluster, use the following command:

```
$sudo service impala-state-store stop
```

Restarting Impala

To restart Impala services in all nodes where it is installed, use the following command:

```
$sudo service impala-server restart
```

To restart any instances of Impala-state-store in the Hadoop Cluster, use the following command:

```
$sudo service impala-state-store restart
```

Upgrading Impala

Upgrading Impala from an older to a newer version is similar to other application upgrades on Linux machines. Upgrading Impala requires stopping the currently running Impala services. Upgrade Impala, and then add extra configurations if needed, and finally restart Impala services. Here we will learn how we can upgrade Impala services depending on our initial installation method.

Upgrading Impala using parcels with Cloudera Manager

This method is used only when Impala is installed using Cloudera Packages, but now you are using Parcels from Cloudera Manager by accessing Cloudera Manager UI. The steps to be followed are:

1. First remove all the Impala-related packages.
2. Connect to the Cloudera Manager Admin Console.
3. Navigate to the **Hosts | Parcels** tab. You should see a parcel with a newer version of Impala that you can upgrade to.
4. Click on **Download**.
5. Click on **Distribute**.
6. Click on **Activate**.
7. Once activation is completed, a **Restart** button will appear.
8. Click on the **Restart** button to restart the Impala service.

Upgrading Impala using packages with Cloudera Manager

The steps to be followed are as follows:

1. Connect to the Cloudera Manager Admin Console.
2. In the **Services** tab, click on the **Impala** service.
3. Click on **Actions**.
4. Click on **Stop**.
5. Update the Impala server on each Impala node in your cluster.
6. Make sure to update `hadoop-1zo-cdh4` depending on whether it is installed already or not.
7. Update Impala shell on each node on which it is installed.
8. Connect to the Cloudera Manager Admin console.
9. In the **Services** tab, click on the **Impala** service.
10. Click on **Actions** and then on **Start**.

Upgrading Impala without Cloudera Manager

The steps to be followed are as follows:

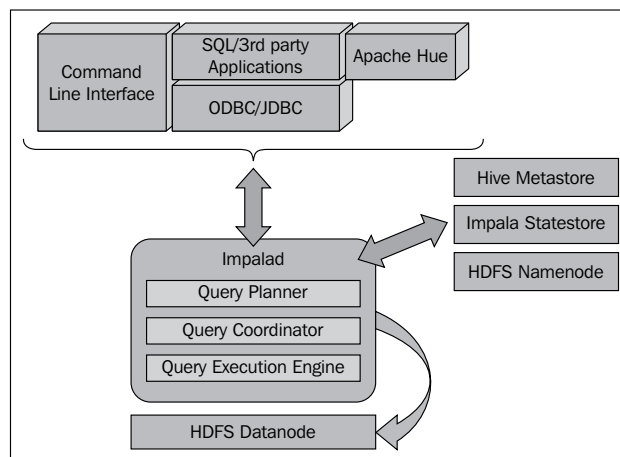
1. Stop Impala services and Impala-state-store in all nodes where it is installed.
2. Validate if any update-specific configuration is needed and, if so, please apply that configuration.
3. Update the Impala-server and Impala shell using appropriate update commands on your Linux OS. Depending on your Linux OS and Impala package types, you might be using these commands, for example, "yum" on RedHat/CentOS Linux and "apt-get" on the Ubuntu/Debian Linux OS.
4. Restart Impala services.

Impala core components

In this section we will first learn about various important components of Impala and then discuss the intricate details on Impala inner workings. Here, we will discuss the following important components:

- Impala daemon
- Impala statestore
- Impala metadata and metastore

Putting together the above components with Hadoop and an application or command line interface, we can conceptualize them as seen in the following figure:



Let's start discussing the core Impala components in detail now.

Impala daemon

At the core of Impala, there exists the Impala daemon, which runs on each DataNode where Impala is installed. The Impala daemon is represented by an actual process named *impalad*. This Impala daemon process *impalad* is responsible for processing the queries, which are submitted through Impala shell, API, and other third-party applications connected through ODBC/JDBC connectors or Hue.

A query can be submitted to any *impalad* running on any node, and that particular node serves as a "coordinator node" for that query. Multiple queries are served by *impalad* running on other nodes as well. After accepting the query, *impalad* reads and writes to data files and parallelizes the queries by distributing the work to other Impala nodes in the Impala cluster. When queries are processing on various *impalad* instances, all *impalad* instances return the result to the central coordinator node. Depending on your requirement, queries can be submitted to a dedicated *impalad* or in a load balanced manner to another *impalad* in your cluster.

Impala statestore

Impala has another important component called Impala statestore, which is responsible for checking the health of each *impalad*, and then relaying each *impala* daemon health to other daemons frequently. Impala statestore is a single running process and can run on the same node where the Impala server or any other node within the cluster is running. The name of the Impala statestore daemon process is *statestored*. Every Impala daemon process interacts with the Impala statestore process providing its latest health status and this information is relayed within the cluster to each and every Impala daemon so they can make correct decisions before distributing the queries to a specific *impalad*. In the event of a node failure due to any reason, *statestored* updates all other nodes about this failure, and once such a notification is available to other *impalad* no other Impala daemon assigns any further queries to the affected node.

One important thing to note here is that even when the Impala statestore component provides a critical update on the node in trouble, the process itself is not critical to the Impala execution. In an event where the Impala statestore becomes unavailable, the rest of the node continues working as usual. When statestore is offline, the cluster becomes less robust, and when statestore is back online it restarts communicating with each node and resumes its natural process.

Impala metadata and metastore

Another important component of Impala is its metadata and metastore. Impala uses traditional MySQL or PostgreSQL databases to store table definitions. While other databases can also be used to configure the Hive metastore, either MySQL or PostgreSQL is recommended. The important details, such as table and column information and table definitions are stored in a centralized database known as a metastore. Apache Hive also shares the same databases for its metastore, because of which Impala can access the table created or loaded by Hive if all the table columns use the supported data types, data format, and data compression types.

Besides that, Impala also maintains information about the data files stored on HDFS. Impala tracks information about file metadata, that is, the physical location of the blocks about data files in HDFS. Each Impala node caches all of the metadata locally, which can expedite the process of gathering metadata for a large amount of data, distributed across multiple DataNodes. When dealing with an extremely large amount of data and/or many partitions, getting table specific metadata could take a significant amount of time. So a locally stored metadata cache helps in providing such information instantly.

When a table definition or table data is updated, other Impala daemons must update their metadata cache by retrieving the latest metadata before issuing a new query against the table in question. Impala uses `REFRESH` when new data files are added to an existing table. Another statement, `INVALIDATE METADATA`, is also used when a new table is included, or an existing table is dropped. The same `INVALIDATE METADATA` statement is also used when data files are removed from HDFS or a DFS rebalanced operation is initiated to balance data blocks in HDFS.

The Impala programming interface

Impala provides the following ways to submit queries to the Impala daemon:

- Command-line interface through Impala shell
- Web interface through Apache Hue
- Third-party application interface through ODBC/JDBC

The Impala daemon process is configured to listen to incoming requests from the previously described interfaces via several ports. Both the command-line interface and web-based interface share the same port; however, JDBC and ODBC use different ports to listen for the incoming requests. The use of ODBC- and JDBC-based connectivity adds extensibility to Impala running on the Linux environment. Using ODBC and JDBC third-party applications running on Windows or other Linux platforms can submit queries directly to Impala. Most of the third-party Business Intelligence applications use JDBC and ODBC to submit queries to the Impala cluster and the *impalad* processes running on various nodes listen to these requests and process them as requested.

The Impala execution architecture

Previously we discussed the Impala daemon, statestore, and metastore in detail to understand how they work together. Essentially, Impala daemons receive queries from a variety of sources and distribute the query load to Impala daemons running on other nodes. While doing so, it interacts with the statestore for node-specific updates and accesses the metastore, either stored in the centralized database or in the local cache. Now to complete the Impala execution, we will discuss how Impala interacts with other components, that is, Hive, HDFS, and HBase.

Working with Apache Hive

We have already discussed earlier the Impala metastore using the centralized database as a metastore, and Hive also uses the same MySQL or PostgreSQL database for the same kind of data. Impala provides the same SQL-like query interface used in Apache Hive. Since both Impala and Hive share the same database as a metastore, Impala can access Hive-specific table definitions if the Hive table definition uses the same file format, compression codecs, and Impala-supported data types for their column values.

Apache Hive provides various kinds of file-type processing support to Impala. When using formats other than a text file, that is, RCFile, Avro, and SequenceFile, the data must be loaded through Hive first and then Impala can query the data from these file formats. Impala can perform a read operation on more types of data using the `SELECT` statement and then perform a write operation using the `INSERT` statement. The `ANALYZE TABLE` statement in Hive generates useful table and column statistics and Impala uses these valuable statistics to optimize the queries.

Working with HDFS

Impala table data are actually regular data files stored in HDFS and Impala uses HDFS as its primary data storage medium. As soon as a data file or a collection of files is available in a specific folder of a new table, Impala reads all of the files regardless of their names, and new data is included in files with the name controlled by Impala. HDFS provides data redundancy through the replication factor and relies on such redundancy to access data on other DataNodes in case it is not available on a specific DataNode. We have already learned earlier that Impala also maintains the information on the physical location of the blocks about data files in HDFS, which helps data access in case of node failure.

Working with HBase

HBase is a distributed, scalable, big data storage system that provides random, real-time read and write access to data stored on HDFS. HBase, a database storage system, sits on top of HDFS; however, like other traditional database storage systems, HBase does not provide built-in SQL support. Third-party applications can provide such functionality.

To use HBase, first the user defines tables in Impala and then maps them to the equivalent HBase tables. Once a table relationship is established, users can submit queries into the HBase table through Impala. Join operations can also be formed including HBase and Impala tables.



To learn more about using HBase with Impala, please visit the Cloudera website at the following link, for extensive documentation:
http://www.cloudera.com/content/cloudera-content/cloudera-docs/Impala/latest/Installing-and-Using-Impala/ciiu_impala_hbase.html

Impala security

Impala is designed and developed to run on top of Hadoop. So you must understand the Hadoop security model as well as the security provided in the OS where Hadoop is running. If Hadoop is running on Linux, then a Linux administrator and Hadoop administrator user can tighten the security, which definitely can be taken into account with the security provided by Impala. Impala 1.1 or higher uses Sentry Open Source Project to provide a detailed authorization framework for Hadoop. Impala 1.1.1 supports auditing capabilities in a cluster by creating auditing data, which can be collected from all nodes and then processed for further analysis and insight.

Here, in this chapter, we will talk about the security features provided by Impala. To start with Impala security, we can consider the following types of security features.

Authorization

Authorization means "who can access the data resources" and "what kind of action is approved for which user." Impala uses the Linux OS user ID of the user who started the Impala shell process or another client application. This user ID is associated with other privileges to be used with Impala. With Impala 1.1, the Open Source Sentry project is used for authorization. so users can learn more by accessing relevant information in this regard.

Impala uses the same authorization privilege model that is used with other database systems, that is, MySQL and Hive. In Impala, privilege is granted to various kinds of objects in schema. Any privilege that can be granted is associated with a level in the object hierarchy. For example, if a container object is given privilege, the child object automatically inherits it.

Currently only Server Name, URI, Databases, and Tables can be used to restrict privileges; however, partition- or column-level restriction is not supported.

Following this we will learn how a restricted set of privileges determines what you can do with each object.

The SELECT privilege

The `SELECT` privilege allows the user to read the data from a table. If users use `SHOW DATABASES` and `SHOW TABLES` statements, only objects for which a user has this privilege will be shown in the output and the same goes with the `REFRESH` and `INVALIDATE METADATA` statements. These statements will only access metadata for tables for which the user has this privilege.

The INSERT privilege

The `INSERT` privilege applies only to the `INSERT` and `LOAD DATA` statements, and allows the user to write data into a table.

The ALL privilege

With the `ALL` privilege users can create or modify any object. This access privilege is needed to execute DDL statements, that is, `CREATE TABLE`, `ALTER TABLE`, or `DROP TABLE` for a table, `CREATE DATABASE` or `DROP DATABASE` for a database, or `CREATE VIEW`, `ALTER VIEW`, or `DROP VIEW` for a view.

Here are a few examples of how you can set the described privileges:

```
GRANT SELECT on TABLE table_name TO USER user_name
GRANT ALL on TABLE table_name TO GROUP group_name
```

Authentication through Kerberos

Authentication means verifying the credentials and confirming the identity of the user before processing the request. Impala uses Kerberos security subsystems to authenticate the user and his or her identity.

In the Cloudera Hadoop distribution, the Kerberos security can be enabled through Cloudera Manager. Running Impala in a managed environment, Cloudera Manager automatically completes the Kerberos configuration. At the time of writing this book, Impala does not support application data wire encryption. Once your Hadoop distribution has Kerberos security enabled, you can enable Kerberos security in Impala.



To learn more about enabling Kerberos security features with Impala, please visit the Cloudera Impala documentation website, where you can find the latest information.

Auditing

Auditing means keeping account of each and every operation executed in the system and maintaining a record of whether they succeed or failed. Using auditing features, users can look back to check what operation was executed and what part of the data has been accessed by which user. The auditing feature helps track down such activities in the system, so respective professionals can take proper measurements. In Impala, the auditing feature produces audit data, which is collected and presented in user-friendly details by Cloudera Manager.

Auditing features are introduced with Impala 1.1.1 and the key features are as follows:

- Enable auditing directory with the *impalad* startup option using `audit_event_log_dir`.
- By default, Impala starts a new audit logfile after every 5,000 queries. To change this count, use the `-max_audit_event_log_file_size` option with the *impalad* startup option.
- Optionally, the Cloudera Navigator application is used to collect and consolidate audit logs from all nodes in the cluster.
- Optionally, Cloud Manager is used to filter, visualize, and produce the audit reports.

Here are the types of SQL queries that are logged with audit logs:

- Blocked SQL queries that could not be authorized
- SQL queries that are authorized to execute are logged after analysis is done and before the actual execution

Query information is logged into the audit log in JSON format, using a single line per SQL query. Each logged query can be accessed through SQL syntax by providing any combination of session ID, user name, and client network address.

Impala security guidelines for a higher level of protection

Now let's take a look at the security guidelines for Impala, which could improve the security against malicious intruders, unauthorized access, accidents, and common mistakes. Here is the comprehensive list, which definitely can harden a cluster running Impala:

- Impala specific guidelines
 - Make sure that the Hadoop ownership and permissions for Impala data files are restricted
 - Make sure that the Hadoop ownership and permissions for Impala audit logs files are restricted
 - Make sure that the Impala web UI is password protected
 - Enable authorization by executing `impalad` daemons with `-server_name` and `-authorization_policy_file` options on all nodes
 - When creating databases, tables, and views, using tables and other databases structures allow policy rules to specify simple and consistent rules
- System specific guidelines
 - Create a policy file that specifies which Impala privileges are available to users in particular Hadoop groups
 - Make sure that the Kerberos authentication is enabled and working with Impala
 - Tighten the HDFS file ownership and permission mechanism

- Keeping a long list of sudoers is definitely a big red flag. Keep the list of sudoers to a bare minimum to stop unauthorized and unwanted access
- Secure the Hive metastore from unwanted and unauthorized access

Summary

In this chapter we covered basic information on Impala, core components, and how various components work together to process the data with lightening speed. We have learned about Impala installation, configuration, upgradating, and security in detail, and in the next chapter we will learn about Impala shell and commands, which can be used to manage Impala components in a cluster.

2

The Impala Shell Commands and Interface

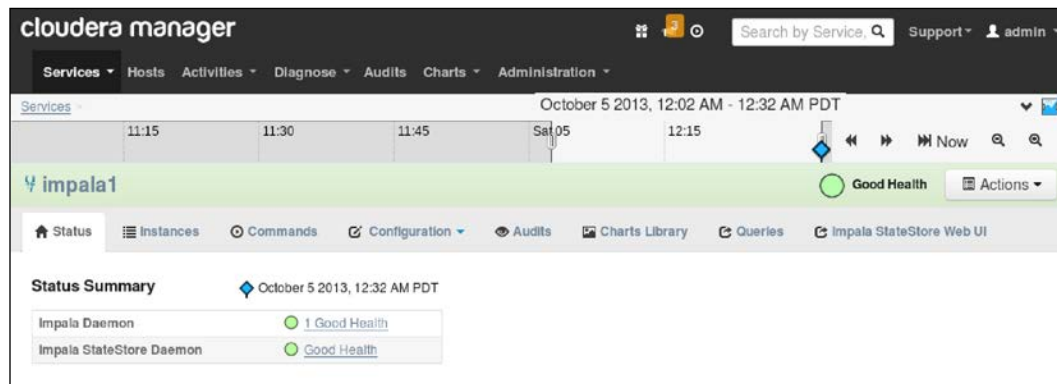
Once `impala` is installed, configured, and ready to start, the next step is to know how to interact with Impala in different ways for various reasons. This chapter explains the various command options to interact with Impala, mainly using command-line references. In the previous chapter, we also discussed various ways to install Impala.

In the previous chapter, we understood that **`impalad`** is the Impala daemon, which runs on every node in the cluster and receives queries submitted through various interfaces such as third-party applications using the ODBC or JDBC connectivity, Web interface, or API, and finally the Impala shell. In general, the `impala-shell` is a process that runs in a node and works as a gateway to connect to *`impalad`* through commands. The Impala shell is used to submit various commands that can set up databases and tables, insert data into tables, and finally submit queries on stored data.

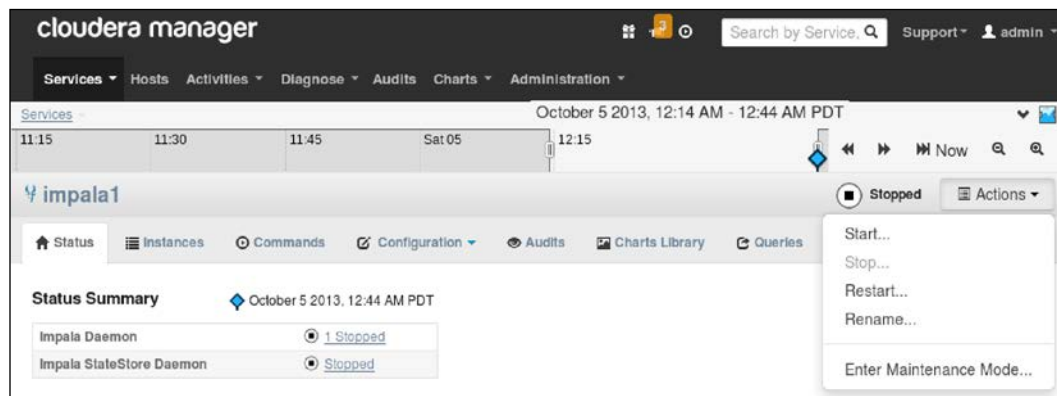
Using Cloudera Manager for Impala

Before we jump into Impala shell, let's first try using Cloudera Manager to check the status of Impala. By default, Cloudera Manager configures to run on port 7180. In your cluster where you have installed Impala using Cloudera Manager, open Cloudera Manager in your favorite web browser and browse through all services to check the status of Impala.

If Impala daemon (*impalad*) and Impala statestore (*statestored*) are installed correctly and running, you will see them listed as shown in the following screenshot:



As shown in the previous screenshot, both **Impala Daemon** and **Impala StateStore Daemon** are running successfully. However, in a situation where Impala shows the **Stopped** status, you can start both the daemons at once just by using the **Actions** button on Cloudera Manager, as shown in the following screenshot:



If we list running processes specific to Impala using the Linux `ps` command, we can find both the *impalad* and *statestored* processes listed as follows:

Impala Daemon (impalad)

```
impala/usr/lib/impala/sbin-retail/impalad --flagfile=/var/run/cloudera-scm-agent/process/21-impala-IMPALAD/impala-conf/impalad_flags
```

Impala Statestore Daemon (statstored)

```
/usr/lib/impala/sbin-retail/statstored --flagfile=/var/run/cloudera-scm-agent/process/20-impala-STATESTORE/impala-conf/state_store_flags
```

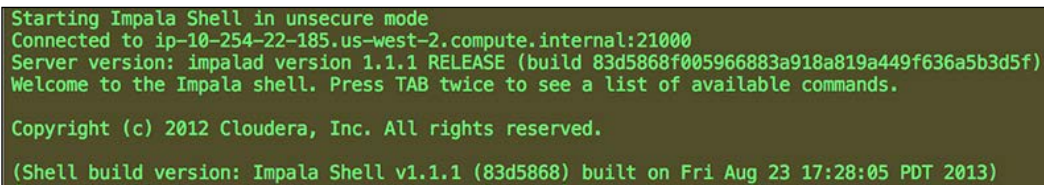
We can see that each daemon starts with its flag file, and on opening the flagfile we can learn more about the base settings for both *impalad* and *statstored*.

Launching Impala shell

Once ready, you can launch the Impala shell just by typing `impala-shell` in the console window. It can run on any machine in your cluster as long as it has connection availability to Impala daemon. Impala-shell can also run on the same machine where *impalad* is running or on a DataNode as well. You can launch Impala shell by just executing the following command in command prompt:

```
$impala-shell
```

After execution, Impala shell's command window will look like the following screenshot:



```
Starting Impala Shell in unsecure mode
Connected to ip-10-254-22-185.us-west-2.compute.internal:21000
Server version: impalad version 1.1.1 RELEASE (build 83d5868f005966883a918a819a449f636a5b3d5f)
Welcome to the Impala shell. Press TAB twice to see a list of available commands.

Copyright (c) 2012 Cloudera, Inc. All rights reserved.

(Shell build version: Impala Shell v1.1.1 (83d5868) built on Fri Aug 23 17:28:05 PDT 2013)
```

The previous `impala-shell` is connected to `impalad`, running on the same machine at `localhost` and port `21000`, where `impala-shell` is also running. Connecting the `impala-shell` using the hostname as `localhost` when `impalad` is also running on the same node is also connected.

If the Impala shell cannot connect to the default server, you may see an **Error connecting: <class 'thrift.transport.TTransport.TTransportException'>, Could not connect to <hostname>:<port>** error message.

To resolve this problem, please make sure that the Impala server is running and you can connect to that server from the machine where you are trying to run `impala-shell`. When the `impala-shell` session is started, queries can be submitted only when `impala-shell` is connected to an instance of `impala` daemon.

Connecting impala-shell to the remotely located impalad daemon

In some cases, impala-shell is installed manually on other machines that are not managed through Cloudera Manager. In such cases, you can still launch impala-shell and submit queries from those external machines to a DataNode where impalad is running. In such a specific scenario, impala-shell is started and connected to remote hosts by passing an appropriate hostname and port (if not the default, 21000).

To use Impala shell to connect to Impala daemons running on other DataNode machines, you just need to have a DataNode hostname and a port number where impalad is configured, to receive queries and pass both hostname and port with the `connect <hostname:port>` command, as shown in the following code:

```
[Not connected] > connect datanode-hostname
[datanode-hostname:21000] >
```

As you can see, I did not include a port number with the `connect` command, and the connection to Impala daemon was made onto the default port 21000. If you have configured Impala daemon to connect from another port, other than port 21000, then you would have to include the correct port number with the `connect` command.

Impala-shell command-line options with brief explanations

Impala-shell can be launched with other optional parameters to either perform a specific action or to provide more information about the action. These command line options are used along with the `impala-shell` command as a parameter. Some of these options are created to provide assistance with impala-shell usage, while others are designed to perform a specific action. For example, the `-q` or `-query` option can run a query directly outside the shell and show the output directly on console windows, as shown in the following screenshot:

```
$ impala-shell -q 'describe studentlist;'
Starting Impala Shell in unsecure mode
Connected to ip-10-254-22-185.us-west-2.compute.internal:21000
Server version: impalad version 1.1.1 RELEASE (build 83d5868f005966883a918a819a449f636a5b3d5f)
Query: describe studentlist
Query finished, fetching results ...
+----+-----+-----+
| name | type | comment |
+----+-----+-----+
| id   | int  |          |
| name | string |          |
+----+-----+-----+
Returned 2 row(s) in 0.04s
```

Another example is to use the `-d` option with a specific database name, as shown in the following screenshot. Once `impala-shell` starts, it connects to `impalad` at `localhost` through the default port `21000` and uses the selected database. In the shell we can call any command specific to the database used, which is also shown in the following screenshot:

```
$ impala-shell -d students
Starting Impala Shell in unsecure mode
Connected to ip-10-254-22-185.us-west-2.compute.internal:21000
Server version: impalad version 1.1.1 RELEASE (build 83d5868f005966883a918a819a449f636a5b3d5f)
Welcome to the Impala shell. Press TAB twice to see a list of available commands.

Copyright (c) 2012 Cloudera, Inc. All rights reserved.

(Shell build version: Impala Shell v1.1.1 (83d5868) built on Fri Aug 23 17:28:05 PDT 2013)
Query: use students
[ip-10-254-22-185.us-west-2.compute.internal:21000] > show tables;
Query: show tables
Query finished, fetching results ...
+-----+
| name |
+-----+
| studentlist |
+-----+
Returned 1 row(s) in 0.01s
```

The syntax to use the command-line option is shown in the following two examples:

```
$ impala-shell -q 'select * from mytable;'
$ impala-shell -d students
```

In the next few sections, we will learn more about these command-line options within their specific category.

General command-line options

In this section, we will learn about a few command options, which can be used to get help about `Impala-shell` or to get the version along with other general details. A list of general commands are listed in the following table:

Command option	Description
<code>-h</code> or <code>--help</code>	This option prints the <code>impala shell</code> help details as console output. You can use this option as follows: \$ <code>impala-shell -h</code> or <code>\$impala-shell --help</code>

Command option	Description
-v or --version	These options print the current running impala-shell version and you can use this option as follows: <code>\$impala-shell -v or \$impala-shell --version</code> An example of using the <code>-v</code> option is as follows: <code>\$ impala-shell -v</code> Impala Shell v1.1.1 (83d5868) built on Fri Aug 23 17:28:05 PDT 2013
-V or --verbose	This option enables the verbose output mode, so you will see lot more information in your output.
--quiet	This option disables the verbose output mode.
-c	This option continues command execution even when query failure occurs. This option is useful when multiple queries are submitted via shell scripts or other means, and processing does not stop if one or more queries return failure.

Connection-specific options

These command options can be use to connect specific impalad located remotely or to connect a specific database after starting impala-shell. A list of connection-specific command options is in the following table:

Command option	Description
-I hostname or -impalad=hostname	This option connects impala-shell to impalad, using a specific host-passed hostname. By default, connection is made on port 21000; however, a different port number can be passed with the hostname as <code>hostname:port</code> . This command is useful when impala-shell is connected to such an impala daemon, which is running on a machine other than the current machine running impala-shell. You can use this command as follows: <code>\$impala-shell -I myhostname:21005</code>
-r or -refresh_after_ connect	This option refreshes Impala metadata after successful connection. This option is the same as applying the <code>REFRESH</code> statement after connecting to the Impala server. You can use this command to refresh metadata just after connecting to impalad as follows: <code>\$impalad -r</code>

Command option	Description
-d database_name or -database database_name	By default, when <code>impala-shell</code> starts and connects to the Impala server, it connects to a database named <code>default</code> . Using this option, you can pass a desired database name to be used as the default after a successful connection. If you have a database named <code>logs</code> and want to use it with <code>impala-shell</code> , you will achieve your objective by using the following command: <code>\$impala-shell -d logs</code>

Query-specific options

These commands are specific to passing queries with the `impala-shell` command. Once you use such a command option, the query result can be saved to a file outside `impala-shell` or can be printed on the console, depending on your choice. A list of query-specific command options is shown in the following table:

Command option	Description
-q query or -query=query	This option allows you to issue a single query directly from a command line. This option is most useful when you have a need to run <code>impala-shell</code> inside a programming language such as Python or Perl, or inside a shell script. If launched from the shell script, the query output will be displayed on the console or can be collected through the programming interface, if used. An example of using this option is as follows: <code>\$ impala-shell -query 'select * from table_name;'</code>
-o filename or --output_file filename	This option allows you to save the query output to a file.
-B or --delimited	This option is use to produce comma-separated, tab-separated, or other delimited text files as output. The default delimiter is the TAB character (<code>\t</code>); however, it can be changed using the <code>-output_delimiter</code> option and passing the desired delimiter character.
-f query_file_name or -query_file=query_file_name	Using this option you can pass a SQL query file that includes all of the queries you want to pass through <code>impala-shell</code> . An example of using this option is as follows: <code>\$impala-shell -f myqueryfile.sql</code>
-p or --show_profiles	This option is similar to the SQL <code>EXPLAIN</code> statement that provides a query-execution plan including detailed query execution steps for every query executed by the <code>impala-shell</code>

Secure connectivity-specific options

If configured correctly, you can pass the appropriate command-line options shown in the following table, to use these security options while connecting impalad from impala-shell.

Command options	Description
-k or --kerberos	When using this option, the impala-shell connects to impalad using the Kerberos authentication. Make sure to have the Kerberos authentication enabled and working with impalad, otherwise a connection cannot be established and errors will be displayed.
-s Kerberos_service_name or --kerberos_service_name=Kerberos_service_name	This option instructs impala-shell to connect to impalad using a particular impalad service principal, as passed in the option. If <code>Kerberos_service_name</code> is not specific, Impala is used by default with Kerberos.

Impala-shell command reference

After you launch Impala shell, you can use Impala shell commands to perform specific tasks. You can also pass these commands to Impala shell using the `-q` option as an argument. Some of these commands are referenced from SQL-like syntax because they are passed to an impala daemon inside impala-shell as SQL-like statements. Inside Impala shell, all the commands must end with a semicolon, as a semicolon is used as a command termination sequence. In this section, we will cover most of these commands, which you can run directly on Impala shell as shown in following screenshot:

```
$ impala-shell
Starting Impala Shell in unsecure mode
Connected to ip-10-254-22-185.us-west-2.compute.internal:21000
Server version: impalad version 1.1.1 RELEASE (build 83d5868f005966883a918a819a449f636a5b3d5f)
Welcome to the Impala shell. Press TAB twice to see a list of available commands.

Copyright (c) 2012 Cloudera, Inc. All rights reserved.

(Shell build version: Impala Shell v1.1.1 (83d5868) built on Fri Aug 23 17:28:05 PDT 2013)
[ip-10-254-22-185.us-west-2.compute.internal:21000] > show databases;
Query: show databases
Query finished, fetching results ...
+-----+
| name   |
+-----+
| default|
| students|
+-----+
Returned 2 row(s) in 0.15s
```

In the previous screenshot, as soon as Impala shell is ready, the `show databases` command is used to list all the databases. These shell commands are described next, within their specific section to provide better context around the commands, and to explain their specific usage.

General commands

You can use the following general commands inside `impala-shell` to perform the various actions described:

Command option	Description
help	<p>You can get more information about the available shell commands using the <code>help</code> command. Alternatively, using <code>help <command_name></code> provides command-specific help. Here is an example:</p> <pre>> help profile; Prints the runtime profile of the last INSERT or SELECT query executed.</pre>
version	<p>Using this command you can get <code>impala-shell</code> and Impala daemon version information. Following is the output of the <code>version</code> command:</p> <pre>Shell version: Impala Shell v1.1.1 (83d5868) built on Fri Aug 23 17:28:05 PDT 2013 Server version: impalad version 1.1.1 RELEASE (build 83d5868f005966883a918a819a449f636a5b3d5f)</pre>
history	<p>Gives a list of all commands passed into <code>impala-shell</code>. The history list is stored in file name <code>~/.impalahistory</code>. You can call this command inside <code>impala-shell</code> as follows:</p> <pre>> history;</pre>
shell or !	<p>Using this command you can run a certain file system command inside <code>impala-shell</code>. For example if you want to see the current working directory or file listing in the folder where the Impala shell is running, you can use the commands as follows:</p> <pre>> !pwd; > ! ls -l;</pre>
refresh	<p>Use this command to refresh impala metadata. This command is the same as the <code>REFRESH</code> command.</p>
connect	<p>Use this command to connect Impala shell with <code>impalad</code> on the remote host and the default port, 21000. If the port is not 21000, then pass the port number along with hostname as <code>hostname:port</code>.</p>
exit/quit	<p>You can use this command to exit from Impala shell.</p>

Query-specific commands

Your primary objective is to use the Impala shell to perform query operations, and in this section we can see a few of those query commands with their examples:

Command option	Description
set/unset	<p>Use this command to manage query options for the current Impala shell session; use <code>set</code> to include query options and <code>unset</code> to remove them. Mostly, the <code>set</code> command is used with specific options for fine tuning or troubleshooting the queries.</p> <p>To set <code>DEFAULT_ORDER_BY_LIMIT</code> to 10, you can use the <code>set</code> command as follows:</p> <pre>> set default_order_by_limit=10;</pre> <p>Now, running the <code>set</code> command in the Impala shell returns the current set query option, as shown in the following commands:</p> <pre>> set;</pre> <p>Default query options:</p> <pre>NUM_SCANNER_THREADS: 0 ABORT_ON_DEFAULT_LIMIT_EXCEEDED: 0 MAX_IO_BUFFERS: 0 DEFAULT_ORDER_BY_LIMIT: -1 BATCH_SIZE: 0 NUM_NODES: 0 DISABLE_CODEGEN: 0 MAX_ERRORS: 0 ABORT_ON_ERROR: 0 MAX_SCAN_RANGE_LENGTH: 0 ALLOW_UNSUPPORTED_FORMATS: 0 SUPPORT_START_OVER: false DEBUG_ACTION: MEM_LIMIT: 0</pre> <p>Query options currently set:</p> <pre>DEFAULT_ORDER_BY_LIMIT: 10</pre> <p>To unset the value, just use the <code>unset</code> command as follows:</p> <pre>> unset default_order_by_limit;</pre>

Command option	Description
profile	Using this command, you can get low-level information on the last processed query. You can use this command to fine-tune the query performance by understanding the output of this command. The same command can also be used for troubleshooting certain issues with the query. The command is as follows: <pre>> profile;</pre>
explain	This command is similar to the EXPLAIN query command to provide an execution plan of a query. An example of using this command is as follows: <pre>> explain select * from studentlist; Query: explain select * from studentlist Query finished, fetching results ...</pre> <pre>+-----+ Explain String +-----+ PLAN FRAGMENT 0 PARTITION: UNPARTITIONED 1:EXCHANGE tuple ids: 0 PLAN FRAGMENT 1 PARTITION: RANDOM STREAM DATA SINK EXCHANGE ID: 1 UNPARTITIONED 0:SCAN HDFS table=default.studentlist #partitions=0 size=0B tuple ids: 0 +-----+</pre> <p>Returned 16 row(s) in 2.36s</p>

Table- and database-specific commands

Now let's take a look at table- and database-specific commands (shown in the following table), which you can use inside the Impala shell to play with your databases and tables. I have explained these commands in great detail in the following chapters, so these commands are listed here only for reference.

Command option	Description
alter	Use this command to change the table structure or table settings.
describe	Use this command to see information about columns, column data types, and column comments for a specific table.
insert	Use this command to store query results into a table.
drop	Use this command to remove a schema object associated with a table or database.
select	Use this command to select a dataset that will process the action, the same as the <code>SELECT</code> statement.
use	Use this command to select a database that will be used to process a group of queries.

Summary

In this chapter, we have covered the Impala command-line interface, explaining various ways Impala shell can connect to Impala daemon. Once a connection between Impala shell and `impalad` is established, we discussed various commands that can be used in `impala-shell` to process *impalad*. You may have seen that various commands perform specific tasks, and by using specific command line options, you can either change the behavior of commands or you can perform the same shell commands directly outside Impala shell.

In the next chapter, we will continue focusing on Impala Query Language and Impala built-in functions.

3

The Impala Query Language and Built-in Functions

By now you will have a good idea of how Impala works and how different components interact with each other to support query requests. In the previous chapter we have learned how the Impala shell interacts with an Impala daemon running either on the same node or some other DataNode, and how users can use certain shell commands to perform their tasks through the Impala shell. Now in this chapter, we will learn how to make great use of the Impala shell to interact with data by using the Impala Query Language. Along with the Impala Query Language, we will also learn various Impala built-in functions along with great examples.

Once again, we would like to repeat that each statement must end with a semicolon in the Impala shell, and you can use multiple statements using copy and paste in the Impala shell. The Impala interpreter parses each statement by recognizing the semicolon and considering it as the end of the current statement, and this way it can parse multiple statements.

In this chapter you will find detailed descriptions of the Impala Query Language, and to extend the knowledge gained from this chapter, you can refer to another chapter that explains most of the Impala query sentences with an example.

Chapter 4, Impala Walkthrough with an Example complements this chapter.

Now let's start learning the Impala Query Language.

The Impala Query Language is based on SQL while providing a great degree of compatibility with HiveQL. Hive statements are based on SQL statements, and because Impala statements are based on SQL, several statements in both Hive and Impala are identical; however, some of the statements do show differences. Let's check the key points regarding how Impala statements are based on SQLs:

- Hive as well as Impala uses the **Data Definition Language (DDL)**.

- To store table structures and their properties, Hive uses metastore, and Impala uses the same metastore to record the information. For example, Impala can access the tables created using Hive statements or directly by using the `CREATE TABLE` command.
- Data types, that is `int`, `tinyint`, `smallint`, `bigint`, `float`, `double`, `boolean`, `string`, and `timestamp`, in Impala share the same name and semantics as Hive for the supported data types.

Now, let's take a quick look at how the Impala Query Language supports HiveQL; note the following key points:

- Impala statements and clauses are similar to those of HiveQL, such as `JOIN`, `UNION ALL`, `ORDERBY`, `LIMIT`, `DISTINCT`, and `AGGREGATE`
- Impala statements support data manipulation statements similar to the **Data Manipulation Language (DML)**
- `SELECT` and `INSERT` statements in Impala function the same as in HiveQL
- Impala also supports `INSERT INTO` and `INSERT OVERWRITE` statements
- Several built-in functions in various categories such as mathematical, conditional, or string are the same in Impala and HiveQL and use the same name and parameter types

We must understand that not every SQL statement is supported in Impala. To make it simple, a list of unsupported SQL statements is at the end of this chapter. Now, let's get to know the most useful SQL language statements, which are commonly used in Impala.

Impala SQL language statements

With Impala, users can work on various types of data through databases, tables, and views. Impala uses these SQL statements to process data stored in databases and tables, and in the next several sections we will study Impala statements using some examples. Databases and the table metadata is modified differently in both Hive and Impala. In Hive, you can use `ALTER`, `CREATE`, `DROP`, or `INSERT` operations to modify the data; however, in Impala, you will have to use `CREATE TABLE`, `ALTER TABLE`, and `INSERT` operations to achieve the same objective. Let's start with Database-specific SQL statements:

Database-specific statements

Let's first understand what a database is:

- A database is a logical entity to group related tables into one single namespace.
- With Impala running on a DataNode, physically, a database in Impala is represented as a directory on **HDFS**. All internal tables, partitions, and data files are saved inside the parent directory.
- An Impala database is created inside the Impala directory on HDFS with the name `database_name.db`.
- The database directory on HDFS is the same as any other directory on HDFS and supports all directory operations like any other directory.

Now let's learn a few database-specific statements in Impala.

The CREATE DATABASE statement

When there is a need to create a database, you can use the `CREATE DATABASE` statement to create a new database as follows:

```
CREATE (DATABASE|SCHEMA)
      [IF NOT EXISTS] database_name [COMMENT 'database_comment']
      [LOCATION hdfs_path];
```

The DROP DATABASE statement

To remove a database from Impala, you can use the `DROP DATABASE` statement. Once the `DROP DATABASE` statement is executed, the corresponding `database_name.db` directory from HDFS is removed. The syntax of the `DROP DATABASE` statement is as follows:

```
DROP (DATABASE|SCHEMA) [IF EXISTS] database_name;
```

There is no protection to the `DROP DATABASE` statement. If the `DROP DATABASE` statement is executed on a database that has tables and data files, all the content will be removed from the HDFS, and the database directory will be deleted. So, it is recommended that you must empty the database yourself before calling the `DROP` statement.

The SHOW DATABASES statement

To get the list of databases available for Impala, use the `SHOW DATABASE` statement with the following syntax:

```
SHOW DATABASES;
```

Using database-specific query sentence in an example

Now, let's see an example of database-specific statements to understand everything together. If you have just installed Impala, started it, and run `SHOW DATABASE`, you will see only one database name, `default`, as shown in the following example:

```
[Hadoop.testdomain:21000] > show databases;
Query: show databases
Query finished, fetching results ...
+-----+
| name   |
+-----+
| default |
+-----+
Returned 1 row(s) in 0.12s
[Hadoop.testdomain:21000] > create database items;
Query: create database items
[Hadoop.testdomain:21000] > use items;
Query: use items
[Hadoop.testdomain:21000] > create table list (item string, total
int);
Query: create table list (item string, total int)
[Hadoop.testdomain:21000] > show databases;
Query: show databases
Query finished, fetching results ...
+-----+
| name   |
+-----+
| default |
| items   |
+-----+
Returned 2 row(s) in 0.13s
[Hadoop.testdomain:21000] > drop database items;
Query: drop database items
ERROR: AnalysisException: Cannot drop current default database: items
```

```
[Hadoop.testdomain:21000] > use default;
Query: use default
[Hadoop.testdomain:21000] > drop database items;
Query: drop database items
```



The error with DROP is seen because we have items databases in use, so we need to use some other database to free items database from use so that we can remove it.

Table-specific statements

After looking at database-specific commands, we will now dig deeper to understand some table-specific commands. Some of these commands also apply to the partitions in the table, and when applicable I have used partitions and tables together in my description and the examples in the following sections.

The CREATE TABLE statement

Because files are stored in a DataNode on the HDFS, tables in Impala work a little differently. When you use the CREATE TABLE statement, Impala creates an internal table where Impala manages the underlying data file for the table. When DROP TABLE is called, the underlying file is physically deleted. The full create table statement is as follows:

```
CREATE [EXTERNAL] TABLE [IF NOT EXISTS] [db_name.]table_name
[(col_namedata_type [COMMENT 'col_comment'], ...)]
[COMMENT 'table_comment']
[PARTITIONED BY
(col_namedata_type [COMMENT 'col_comment'], ...)]
[ [ROW FORMAT row_format] [STORED AS file_format] ]
[LOCATION 'hdfs_path']
data_type
: primitive_type
primitive_type
: TINYINT | SMALLINT | INT | BIGINT | BOOLEAN
| FLOAT | DOUBLE | STRING | TIMESTAMP
row_format
: DELIMITED [FIELDS TERMINATED BY 'char' [ESCAPED BY 'char']]
[LINES TERMINATED BY 'char']
file_format:
: PARQUETFILE | SEQUENCEFILE | TEXTFILE | RCFILE
```


When a new table is created based on some other table, the `LIKE` clause is used. There are many places on the internet to learn about the `CREATE TABLE` SQL statement.

The `CREATE EXTERNAL TABLE` statement

While using the `CREATE EXTERNAL TABLE` syntax, the newly created table points to an existing data file on HDFS. Using this statement, the file data is not imported from the existing data file on HDFS to the new table; instead the new table points to the data file on HDFS while the new table is empty. When external is not defined, the data is copied to the new table so the table itself has the data. The process to query the data does not change and still remains the same.

When `EXTERNAL` is used to create a table, Impala treats the table as an external mean. The data files are produced outside Impala, and when `DROP TABLE` is called on that table, the table is removed; however, the underlying data is kept as is.

Because the data file is read directly from HDFS, if there are any changes to the data file, you must use the `REFRESH` statement `impala-shell` so that Impala can recognize the changes and can use the updated data file. The statement to create a table with the `EXTERNAL` clause is the same as `CREATE TABLE`, and the only difference is that it starts with `CREATE EXTERNAL TABLE` as shown in the following line of code:

```
CREATE EXTERNAL TABLE table_name;
```

The `ALTER TABLE` statement

Sometimes, you may need to modify the structure or properties of the table, and to achieve this objective the `ALTER TABLE` statement is used. As Impala shares the table metastore with Hive, the table metadata is updated using `ALTER TABLE`, which is available to any other application using the same metadata. It is important to know that `ALTER TABLE` does not actually perform any operation on the actual data; instead, the alteration is done on metadata. So, to achieve full transformation of the data, you will need to make those necessary modifications in the data stored in HDFS.

Here is the statement using `ALTER TABLE` to rename a table:

```
ALTER TABLE old_table_name RENAME new_table_name;
```

To change the physical location of the directory in HDFS where Impala looks for table-specific data files, use the following line of code:

```
ALTER TABLE table_name SET LOCATION 'directory_name_on_HDFS';
```

You can use the following syntax to change the table data file format to meet Impala file format requirements:

```
ALTER TABLE table_name
SET FILEFORMAT { PARQUETFILE | RCFILE | SEQUENCEFILE | TEXTFILE }
```



Chapter 7, Advanced Impala Concepts, has detailed information on various file formats supported in Impala.

Creating an empty table along with the partitioning scheme definition and altering the table partition can be done using the following code:

```
CREATE TABLE table_name (def data_type)
PARTITIONED BY (partiton_name partition_type);
ALTER TABLE table_name ADD PARTITION
(partition_type='definition');
```

The DROP TABLE statement

When there is a need to remove the table from the database, you can use the DROP TABLE command. The DROP command deletes the table and associated files underneath in the HDFS directory, unless the table was created with the EXTERNAL clause. Because the table and data cannot be recovered after deletion, it is suggested that you ensure you have the correct database in use before issuing the DROP statement. The syntax for using DROP TABLE is as follows:

```
DROP TABLE [IF EXISTS] table_name;
```

The SHOW TABLES statement

When you want to see a list of all tables in a database, use the SHOW TABLE statement with the following syntax:

```
SHOW TABLES;
SHOW TABLES [IN database_name];
```

The DESCRIBE statement

Using the DESCRIBE statement, you can learn more about the table metadata with the following syntax:

```
DESCRIBE table_name;
```

Alternatively, you can use the `FORMATTED` clause with the `DESCRIBE` statement, which will provide various other information about the table, as follows:

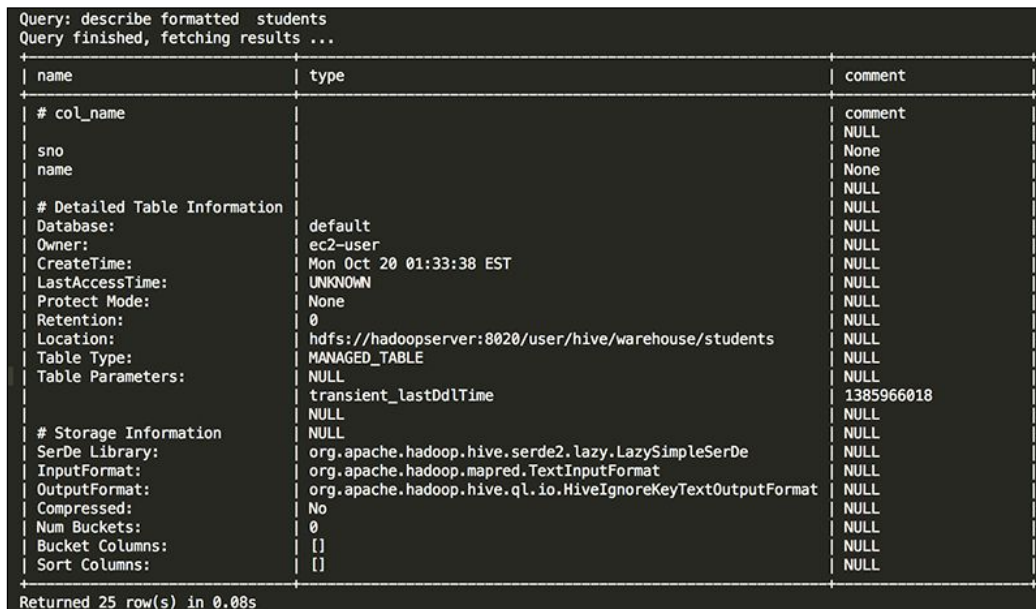
```
DESCRIBE [FORMATTED] table_name;
```

Here is an example of using `DESCRIBE` and `DESCRIBE FORMATTED` to help you understand the difference between the two.

The following code shows how `DESCRIBE` displays the output:

```
[Hadoop.testdomain:21000] > create table students (sno int, name
string);
Query: create table students (sno int, name string)
[Hadoop.testdomain:21000] > describe students;
Query: describe students
Query finished, fetching results ...
+-----+-----+-----+
| name | type | comment |
+-----+-----+-----+
| sno | int | |
| name | string | |
+-----+-----+-----+
Returned 2 row(s) in 1.66s
```

The following screenshot shows how `DESCRIBE FORMATTED` displays the results to show the difference:



```
Query: describe formatted students
Query finished, fetching results ...
+-----+-----+-----+
| name | type | comment |
+-----+-----+-----+
| # col_name | | comment |
| sno | | None |
| name | | None |
| # Detailed Table Information | | |
| Database: | default | NULL |
| Owner: | ec2-user | NULL |
| CreateTime: | Mon Oct 20 01:33:38 EST | NULL |
| LastAccessTime: | UNKNOWN | NULL |
| Protect Mode: | None | NULL |
| Retention: | 0 | NULL |
| Location: | hdfs://hadoopserver:8020/user/hive/warehouse/students | NULL |
| Table Type: | MANAGED_TABLE | NULL |
| Table Parameters: | NULL | NULL |
| | transient_lastDdlTime | 1385966018 |
| | NULL | NULL |
| # Storage Information | | |
| SerDe Library: | org.apache.hadoop.hive.serde2.lazy.LazySimpleSerDe | NULL |
| InputFormat: | org.apache.hadoop.mapred.TextInputFormat | NULL |
| OutputFormat: | org.apache.hadoop.hive.ql.io.HiveIgnoreKeyTextOutputFormat | NULL |
| Compressed: | No | NULL |
| Num Buckets: | 0 | NULL |
| Bucket Columns: | [] | NULL |
| Sort Columns: | [] | NULL |
+-----+-----+-----+
Returned 25 row(s) in 0.08s
```

The INSERT statement

The `INSERT` statement is used to insert data into tables and partitions, which are already created using the `CREATE TABLE` statement. Impala can also use the `INSERT` statement with the tables that are created in Hive as they both share the same metastore. The `INSERT` statement can use various clauses, such as `INTO` or `OVERWRITE`, which change the `INSERT` statement behavior; so, let's learn more about the key features of the `INSERT` statement:

- `INSERT INTO` **statement**: This appends data into table
- `INSERT OVERWRITE` **statement**: This replaces the data in the table with the new data

To insert data into a table, you will use the `INSERT` statement with the `SELECT` statement, as shown in the following line of code:

```
INSERT [INTO | OVERWRITE] TABLE table_name SELECT ...
```

Each `INSERT` statement creates new data files in HDFS with unique names, and this way multiple `INSERT INTO` statements can be executed simultaneously. It is possible that `INSERT` commands were executed on a different Impala daemon (*impalad*); then, using the `REFRESH table_name` command on other nodes will help in syncing the data into a single table effectively. In general, the `INSERT` statement is very detailed, and to learn the various functions that come with it, my suggestion would be to look at the SQL statements documentation for `INSERT`.

The SELECT statement

The `SELECT` statement is used to select data from a table, which is part of the database currently in use. To use the database, you start the `USE` statement first and then use the `SELECT` statement. Here are some features of the `SELECT` clause in Impala:

- The `DISTINCT` clause can also be used but it is applied per query
- The `SELECT` clause also uses the `WHERE`, `GROUP BY`, and `HAVING` clauses
- You can also use `LIMIT` while using `ORDER BY` with `SELECT`

I have written several examples by using `SELECT` with other clauses in this chapter, so here is the syntax of using the `SELECT` clause in SQL statements for reference purposes:

```
SELECT column_name, column_name FROM table_name;
SELECT * FROM table_name;
```

Internal and external tables

It is good that we can have a little discussion on internal and external tables while learning about table-specific statements. When using `CREATE TABLE`, the newly created table is considered as the internal table, whereas while using the `CREATE EXTERNAL TABLE` statement the tables created are considered as external tables.

The properties of internal tables in Impala are as follows:

- With an internal table, a directory in HDFS is created to store data files
- While using the `INSERT` statement, the data is stored into files into the directory at HDFS
- The `LOAD DATA` statement reads data from the files from HDFS
- `DROP TABLE` removes the directory along with files in HDFS

Data types

A data type is an attribute that specifies the type of data that the object can hold, such as integer data, character data, monetary data, date and time data, and binary strings. In Impala, the data in each and every table belongs to some kind of data type; so, it is very important to understand various data types supported in Impala. Each data type has its range and the value belonging to the data type stays within the range. Impala does not convert any data type to another data type by itself, and if a conversion is needed the `CAST` operator can be used to transform some of the data types to others. I have included the use of `CAST` operator where it is applicable.

Let's start learning some of most common data types in Impala as follows:

- **BOOLEAN:** While creating a table, you can define a field or column in the tables as a `BOOLEAN` data type, and the Boolean value represents either true or false choices. These values can be written in uppercase, lowercase, or mixed case. However, when these values are queried from a table, these values are returned in lowercase as `true` or `false`.

Here is an example of using the boolean operator:

```
[Hadoop.testdomain:21000] > create table list (item string, listed
boolean);
Query: create table list (item string, listed boolean)
```

It can be noted that the `CAST()` operator can be used to convert numeric values to Boolean values. A value `0` represents `false` and any non-zero value is converted to `true`. String values cannot be converted to `BOOLEAN` values; however, `BOOLEAN` values can be converted to strings, returning `1` for `true` and `0` for `false`.

- **INT:** `INT` in Impala represents a 4-byte integer type of data when used with `CREATE TABLE` or `ALTER TABLE`. The value of `INT` in Impala ranges between -2147483648 and 2147483647, and there is no `UNSIGNED` subtype with `INT`.

Here is how Casting works with the `INT` data type:

Impala automatically converts the `INT` data type to a larger integer data type (`BIGINT`) or a floating-point data type (`FLOAT` or `DOUBLE`).

You can use the `CAST ()` operator to convert `INT` to `TINYINT`, `SMALLINT`, `STRING`, or `TIMESTAMP`.

You can also cast an `INT` value to `TIMESTAMP`. While using `Cast ()` with `INT N` to `TIMESTAMP`, the resulting value shows `N` seconds past the start of the epoch date (January 1, 1970).

- **BIGINT:** In Impala, `BIGINT` represents an 8-byte integer type of data when used with the `CREATE TABLE` or `ALTER TABLE` statement. The value of `BIGINT` in Impala ranges between -9223372036854775808 and 9223372036854775807, and there is no `UNSIGNED` subtype with `BIGINT`.

Here is how Casting works with the `BIGINT` data type:

Impala automatically converts `BIGINT` to a floating-point type (`FLOAT` or `DOUBLE`).

Using the `CAST ()` operator, `BIGINT` is converted to `TINYINT`, `SMALLINT`, `INT`, `STRING`, or `TIMESTAMP`.

Conversion of `BIGINT` to `TIMESTAMP` also works as `INT`. While using `Cast ()` with `BIGINT N` to `TIMESTAMP`, the resulting value shows `N` seconds past the start of the epoch date (January 1, 1970).

- **SMALLINT:** In Impala, `SMALLINT` represents a 2-byte integer type of data when used with the `CREATE TABLE` or `ALTER TABLE` statements. The value of `SMALLINT` in Impala ranges between -32768 and 32767, and there is no `UNSIGNED` subtype with `SMALLINT`.

Here is how Casting works with the `SMALLINT` data type:

Impala automatically converts `SMALLINT` to a larger integer data type (`INT` or `BIGINT`) or a floating-point type (`FLOAT` or `DOUBLE`).

Using the `CAST ()` operator, `SMALLINT` is converted to `TINYINT`, `STRING`, or `TIMESTAMP`.

`SMALLINT` conversion to `TIMESTAMP` also works with `INT`. While using `Cast ()` with `SMALLINT N` to `TIMESTAMP`, the resulting value shows `N` seconds past the start of the epoch date (January 1, 1970).

- **TINYINT:** In Impala, TINYINT represents a 1-byte integer type of data when used with the CREATE TABLE or ALTER TABLE statement. The value of TINYINT in Impala ranges between -128 and 127, and there is no UNSIGNED subtype with SMALLINT.

Here is how Casting works with the TINYINT data type:

Impala automatically converts TINYINT to a larger integer data type (SMALLINT, INT, or BIGINT) or a floating-point type (FLOAT or DOUBLE).

Using the CAST () operator, SMALLINT is converted to only STRING and TIMESTAMP.

TINYINT conversion to TIMESTAMP also works as INT. While using Cast () with BIGINT N to TIMESTAMP, the resulting value shows N seconds past the start of the epoch date (January 1, 1970).

More information on the INT data type is available in mathematical functions also.

- **DOUBLE:** A DOUBLE data type in Impala represents an 8-byte double, precision, floating-point data type when used with the CREATE TABLE or ALTER TABLE statement. The value of DOUBLE in Impala ranges between 4.94065645841246544e-324d and 1.79769313486231570e+308, and this value can either be positive or negative.

Casting with DOUBLE works, as follows, in Impala:

Impala does not automatically convert DOUBLE to any other type.

The CAST () function can be used to convert DOUBLE values to FLOAT, TINYINT, SMALLINT, INT, BIGINT, STRING, TIMESTAMP, or BOOLEAN.

Exponential notation in DOUBLE literals can be used when casting from STRING.

As an example, value 1.0e6 represents one million.

- **FLOAT:** A FLOAT data type in Impala represents a 4-byte single precision floating-point data type when used with the CREATE TABLE or ALTER TABLE statement. The value of DOUBLE in Impala ranges between 1.40129846432481707e-45 and 3.40282346638528860e+38, and this value can either be positive or negative.

Casting with the FLOAT data type works as follows in Impala:

Impala automatically converts FLOAT to a more precise DOUBLE value but not from DOUBLE to FLOAT.

For converting `FLOAT` to other data types, you can use the `CAST()` function and the result will be any one of the `TINYINT`, `SMALLINT`, `INT`, `BIGINT`, `STRING`, `TIMESTAMP`, or `BOOLEAN` data types.

Exponential notations can be used in `FLOAT` when needed. Exponential literals can also be used when casting from `STRING`; as an example `1.0e6` represents one million.

- **STRING:** A `STRING` data type holds a maximum of 32767 bytes of data in it when used with the `CREATE TABLE` or `ALTER TABLE` statements. Here are the key features of using the `STRING` data type:
 - It is suggested that you limit the string values to the ASCII character set for full support.
 - Multibyte characters can also be used; however, their application will be limited to query operations. String manipulation, comparison operators, and the `ORDER BY` clause may not function correctly with multibyte characters in the `STRING` data type.
 - Impala does not include the metadata definition for ISO-8859-1 or ISO-8859-2-encoded national language aspect type of data, so you would need to implement the application-size logic if you want to support such a requirement.

Here is how casting works with the `STRING` data type:

Impala does not automatically convert `STRING` to any other type.

The `CAST()` function can be used for converting `STRING` to other data types, such as `TINYINT`, `SMALLINT`, `INT`, `BIGINT`, `FLOAT`, `DOUBLE`, or `TIMESTAMP`.

Casting `STRING` values to `BOOLEAN` is not permitted; however, `BOOLEAN` values `1` and `0` can return as `true` and `false`, respectively.

- **SUM:** `SUM` is an aggregate function, which returns the sum of the set of numbers in the table. Here are some key properties of `SUM` functions:
 - `SUM` can be used with numeric columns in a table
 - You can also use `SUM` with the numeric result of a function or expression on a column value
 - When `SUM` is applied, rows with `NULL` values are ignored
 - If a table is empty, `SUM` will return `NULL` as the result
 - If all the values supplied to `MIN` are also `NULL`, `SUM` will return `NULL` as the result

The SUM function can be used in various ways; here are a few syntaxes using SUM:

```
SELECT SUM(column_name) FROM table_name;  
SELECT SUM(distinct(column_name)) FROM table_name;  
SELECT SUM(length(column_name)) FROM table_name;
```

- **TIMESTAMP:** A `TIMESTAMP` in Impala represents a point in time when the `TIMESTAMP` data type is used with the `CREATE TABLE` or `ALTER TABLE` statement. The resolution of the time portion of a `TIMESTAMP` value is in nanoseconds.

As `TIMESTAMP` values are time-specific values, they are not stored using the local time zone; instead, all timestamps in Impala are stored relative to GMT.

Impala automatically converts `STRING` values of the correct format into `TIMESTAMP` values if they are written in any supported time format string.

For example, 1980-12-21 or 2001-01-01 05:05:50 can be converted to `TIMESTAMP` values.

Casting an integer or floating-point value `N` to `TIMESTAMP` produces a value, which is `N` seconds past the start of the epoch date (January 1, 1970).

Now, we are going to learn about how to use operators with Impala SQL statements.

Operators

An operator is a symbol specifying an action that is performed on one or more expressions. Operators are used with multiple statements in SQL. There are various kinds of operators in SQL. They are as follows:

- Arithmetic operators
- Logical operators
- The assignment operator
- Scope-resolution operators
- Bitwise operators
- Set operators
- Comparison operators
- The string-concatenation operator
- Compound operators
- Unary operators

Explaining each kind of operator is out of our scope here, so we will discuss a few commonly used operators in Impala as follows.

- **BETWEEN:** The `BETWEEN` operator is used with the `WHERE` clause, and this operator compares the values between the lower and upper bound given with the `WHERE` clause. The comparison is successful if the expression is greater than or equal to the lower bound and less than or equal to the upper bound. If the bound values are switched, the lower bound is greater than the upper bound; the `BETWEEN` operator does not match any values as a result. The syntax to use the `BETWEEN` operator with the `WHERE` clause is as follows:

```
expression BETWEEN lower_bound AND upper_bound
```

Example:

```
SELECT column_name FROM table_name WHERE value_in_column  
    BETWEEN lower_bound AND upper_bound;
```

Here are a few key things to remember when using the `BETWEEN` operator in Impala:

The `BETWEEN` operator works with any kind of data type; however, it is not practical to use it with `BOOLEAN`

Sometimes it is possible that the values provided are in the lower or upper bound, so you can apply the `Cast()` operator to those values to convert them into compatible types

It is also important to note that the `BETWEEN` operator is typically used with numeric data types, so other functions can also be used to extract numeric values, if possible.

While using `BETWEEN` with a string, it is advisable to use the `upper()`, `lower()`, `substr()`, or `trim()` function to operate on the string instead of having them operated with the `BETWEEN` operator, because variation in the string length could change the outcome of the comparison operator in `BETWEEN`.

- **DISTINCT:** The SQL `DISTINCT` operator is used with the `SELECT` statement to retrieve only unique data entries, depending on the column list selected in the table. Here are the key features of the `DISTINCT` operator when used with Impala:
 - The `DISTINCT` operator returns the unique values from the column
 - `NULL` is also included as a value in the column and as part of the result
 - The `DISTINCT` operator can also return unique combinations of values from multiple columns within the same `SELECT` statement

- The `DISTINCT` operator can be combined with other functions, such as `COUNT`, to aggregate total unique values found in the reference column
- Impala does not support using `DISTINCT` in more than one aggregate function in the same query

Here is the syntax and an example of using the `DISTINCT` operator in Impala:

```
SELECT DISTINCT column_name FROM table_name;
SELECT DISTINCT column_name1, column_name2 FROM table_name;
SELECT COUNT(DISTINCT column_name) FROM table_name;
SELECT SUM(DISTINCT column_name) FROM table_name;
SELECT SUM(DISTINCT column_name) FROM table_name WHERE
(column_name CONDITION value);
```

- **LIKE:** Sometimes performing wildcard-based comparisons with `STRING` data is required, and for this requirement the `LIKE` operator is used. In Impala, the `LIKE` operator is used as a comparison operator to match the `STRING` type of data. For a single character match, `LIKE` uses `_` (underscore) and for multiple characters, the `%` (percentage) sign is used. Using the `%` wildcard at the end of the string to get efficient results is suggested.

The following is the syntax of using the `LIKE` operator in Impala:

```
SELECT column_name FROM table_name WHERE column_name LIKE '%';
SELECT column_name from table_name WHERE column_name LIKE '_';
SELECT column_name from table_name WHERE column_name LIKE
 '_' OR column_name LIKE '%';
```

Now, let's see a few examples of using the `LIKE` operator in Impala. The following statement will return all the state names, which are only two characters and start with the letter `C`:

```
SELECT DISTINCT(state_names) from us_state_list WHERE
state_name LIKE 'C_';
```

The following statement will return all the names that start with `Jo` and can be two or more characters long. In this Impala statement, the result will include John, Joe, Jo, and Johnson.

```
SELECT DISTINCT(names) from names_list WHERE names LIKE 'Jo%';
```

Functions

SQL has lots of built-in functions to perform calculations on the data stored in various columns in tables. There are two different kinds of functions and Impala uses both kinds:

- **Scalar functions:** These functions return a single value, which is based on the input value. Scalar functions are mostly used with the `STRING` and `TIMESTAMP` columns. The most common scalar functions are `UCASE`, `LCASE`, `MID`, `LEN`, `ROUND`, `NOW`, and `FORMAT`. Short descriptions of these SQL Scalar functions follow:
 - `FORMAT()`: This function formats the column value to the display per user preference
 - `NOW()`: This function returns the current system time and date as a value
 - `LEN()`: This function returns the length of the text value in the column
 - `ROUND()`: This function rounds the numeric column value to the specified decimal number
 - `MID()`: This function extracts specific characters from text values in a column
 - `UCASE()`: This function converts the column text value to all upper case
 - `LCASE()`: This function converts the column text value to all lower case
- **Aggregation Functions:** These functions also return a single value after the calculation is done on column values. Aggregation functions are operated mostly on numeric column values. The most-used aggregation functions are `AVG`, `COUNT`, `MAX`, `MIN`, `FIRST`, `LAST`, and `SUM`. We will learn a few of them in *Chapter 4, Impala Walkthrough with an Example*.

Let's go through the details of a few aggregation functions as follows:

- **AVG:** `AVG` is the aggregation function that returns the average value from a group of numbers in the specified column, as supplied in the SQL statement. The `AVG` function uses a single argument, which can be numeric or a numeric result of other functions or expressions applied to the column value. If there is a row with a `NULL` value, the `AVG` function ignores it. For an empty table, `AVG` returns `NULL`, or if all column values are null, `AVG` will return `NULL`. The result data type of an `AVG` function is `DOUBLE`.

Here is the syntax of the AVG function; however, it can be used in various ways:

```
SELECT AVG(column_name) FROM table_name;
SELECT AVG(LENGTH(column_name)) FROM table_name;
SELECT AVG(ISNULL(value1, value2) ) FROM table_name;
SELECT AVG(column_name) FROM table_name WHERE column_value
    CONDITION;
SELECT column_name1, AVG(column_name2) FROM table_name
    GROUP BY column_name1;
```

- **COUNT:** COUNT is another aggregation function in Impala, which returns the number of rows or the number of nonnull rows provided in the statement. When using COUNT(*), the result includes everything including NULL values. However, when using COUNT(column_name), the nonnull values from the specific column_name are calculated. To eliminate duplicate values in COUNT, you can use the DISTINCT operator with column_name first to pass unique items into the COUNT function. The result of the COUNT function is of a BIGINT data type. Here are a few examples and syntaxes of using the COUNT function in Impala:

```
SELECT COUNT(*) FROM table_name;
SELECT COUNT(column_name) FROM table_name;
SELECT COUNT(DISTINCT column_name) FROM table_name;
SELECT column_name1, COUNT(column_name2) FROM table_name;
SELECT COUNT(*) FROM table_name WHERE column_value
    CONDITION;
SELECT column_name1, COUNT(column_name2) FROM table_name
    GROUP BY column_name1, column_name2;
```

- **MAX and MIN:** MAX is another aggregate function, which returns the maximum value from a set of numbers, while MIN returns the minimum value from a set of numbers. Both MAX and MIN support single numeric arguments or numeric results of a function or any expression applied to a column value. Both MAX and MIN ignore NULL values, and for empty tables both return NULL results. The output data type of MAX and MIN is the same as its input argument data type.

When MAX or MIN is used with the GROUP BY clause, both return one value for each combination of grouping values. The following is the syntax and a few examples of using MAX functions:

```
SELECT MAX(column_name) FROM table_name;
```

Examples:

```
SELECT MAX(days) FROM full_year_data WHERE month='October'
    and year = '2013';
SELECT MAX(DISTINCT A) FROM table_name;
SELECT column_name1, column_name2, MAX(column_name3) FROM
    table_name GROUP BY column_name1, column_name2;
```

And here is the syntax and a few examples of using MIN functions:

```
SELECT MIN(column_name) FROM table_name;
```

Examples:

```
SELECT MIN(LENGTH(s)) FROM table_name;
SELECT column_name1, MIN(column_name2) FROM table_name ORDER BY
column_name1 ;
SELECT MIN(Price) AS Order_Price FROM Items;
```

Clauses

SQL statements require clauses to fulfill the statement condition or make it complete. For example, the `SELECT` statement will not be able to fulfill its action unless we provide what kind of `SELECT` action is actually needed. Most of these clauses are used with `SELECT`; however, some of them may find other uses as well. Now, let's learn the most common clauses in SQL, which are typically used in Impala. They are as follows:

- **FROM:** The `SELECT` statement cannot be completed without the `FROM` clause. The `FROM` clause specifies one or more tables containing the data that the query retrieve from. The common syntax for the `FROM` clause along with a few examples is as follows:

```
FROM [table_name,...]
    WHERE... [Condition];
```

Examples:

```
SELECT name, age, class, city, state, country FROM
    studentslist;
SELECT name, age, class, city, state, country FROM
    studentslist WHERE age < 18;
```

- **WHERE:** The next clause in our list is WHERE. The WHERE clause is used in Impala to extract only those records that fulfill the defined criteria in SQL statements. WHERE is very popular and is one of the most-used clauses along with SELECT and FROM. The syntax for the WHERE clause with an example is as follows:

```
SELECT column_name, column_name FROM table_name
      WHERE column_name operator value;
```

Example:

```
SELECT name, age, class, city, state, country FROM
      studentslist WHERE city = 'San Francisco';
```

Let's study the condition operators in the following table, which are used with WHERE:

Operator	Description
=	Equal
<>	Not equal to
>	Greater than
<	Less than
>=	Greater than or equal to
<=	Less than or equal to
BETWEEN	Between an inclusive range
LIKE	Search for a pattern
IN	To specify multiple possible values for a column

- **WITH:** Sometimes SQL statements can become very complex while dealing with multiple tables and associated conditions. To make the SQL statement easier to understand and process, the WITH clause is used before the SELECT statement to define aliases for the expression that are referenced multiple times within SELECT statements. Using the WITH clause, you can apply a single command to the existing SELECT statements without modifying these statements. The syntax of the WITH clause is shown in the following example:

```
WITH [common_table_statements];
```

Example:

```
WITH myWithEx1 as (SELECT 1), myWithEx2 as (SELECT 2) INSERT into
SELECT * FROM myWithEx1 UNION ALL SELECT * from myWithEx2;
```

In the previous example, we have defined two aliases, myWithEx1 and myWithEx2, which are referenced by the SELECT query as defined.

- **GROUP BY:** The `GROUP BY` statement is used along with aggregate functions such as `COUNT()`, `AVG()`, `SUM()`, `MIN()`, or `MAX()`, to group the results set by one or more columns defined in the SQL statement.

The syntax of the `GROUP BY` statement is as follows:

```
SELECT column_name, aggregate_function(column_name)
FROM table_name
WHERE column_name operator value
GROUP BY column_name;
```

In the following example, we are counting items on `ID` from `table1` and then grouping them by `ID` from `table2`:

```
SELECT table1.Name, COUNT(table2.ID) AS TotalOrders FROM table2
LEFT JOIN table1
ON table2.ID=table1.ID
GROUP BY Name;
```

- **ORDER BY:** The `ORDER BY` clause is used with SQL statements to sort the result data by one or more columns. The sorting is done in **ascending** order by default, and to change sort order you can use `DESC` at the end of the statement. The syntax of the `ORDER BY` clause is shown in the following example:

```
SELECT column_name, column_name
FROM table_name
ORDER BY column_name, column_name ASC|DESC;
```

Example:

```
SELECT * FROM citizens ORDER BY state;
SELECT * FROM items ORDER BY price DESC;
```

- **HAVING:** The `HAVING` clause is also used with aggregate functions such as `COUNT()`, `AVG()`, `SUM()`, `MIN()`, or `MAX()`, when a filter operation is conducted on a `SELECT` query. This clause also works with `GROUP BY` in some cases. The syntax of a `HAVING` clause is as follows:

```
SELECT column_name, aggregate_function(column_name) FROM
table_name
WHERE column_name operator value GROUP BY column_name
HAVING aggregate_function(column_name) operator value;
```

- **LIMIT:** Sometimes, when you want to limit the results of a `SELECT` query, you can use the `LIMIT` clause to set the maximum number of rows in the result set. The syntax of the `LIMIT` clause is as follows:

```
SELECT column_name(s) FROM table_name LIMIT number;
```


Query-specific SQL statements in Impala

Now, we will spend some time in understanding the query-specific SQL statements used in Impala. Most of these statements are exactly the same as they are defined in SQL, so to learn more, I would suggest you to look at any SQL reference documentation. Here, I am covering some key information for reference purposes:

- **EXPLAIN:** Using the `EXPLAIN` clause, we can learn the execution plan of a SQL statement by understanding low-level mechanisms that Impala will use to read and process the data in the whole cluster, and then finally show the results. You can use the `EXPLAIN` clause ahead of a `SELECT` statement as shown in the following example:

```
[Hadoop.testdomain:21000] > EXPLAIN SELECT * FROM list;
Explain query: select * from list
PLAN FRAGMENT 0
  PARTITION: UNPARTITIONED

  1:EXCHANGE
    tuple ids: 0

PLAN FRAGMENT 1
  PARTITION: RANDOM

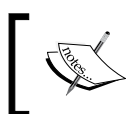
  STREAM DATA SINK
    EXCHANGE ID: 1
    UNPARTITIONED

  0:SCAN HDFS
    table=default.list #partitions=0 size=0B
    tuple ids: 0
  REFRESH table_name;
```

- **REFRESH:** In a multimode environment, the data files reside on multiple DataNodes while the Impala shell is interacting with the Impala daemon (which acts as the data coordinator for all other nodes). Data files can be updated on other nodes without any update event or information to the coordinator. In this situation, using the `REFRESH` clause with the table name loads the latest metadata and block location of the data files for a particular table.

Please refer to *Chapter 2, The Impala Shell Commands and Interface*, to understand more on how `REFRESH` works and why it is so important to use.

- **JOIN:** The `JOIN` clause is used in SQL statements to select data from two or more tables and then return the result set containing items from some of all of those tables, depending on the conditions applied. The `JOIN` query result set is filtered by including the corresponding join column names in the `ON` clause or by comparison operators referencing columns from both tables in the `WHERE` clause. To improve `JOIN` performance, here are some suggestions:
 - It is advisable to perform the `JOIN` operation on the biggest table first and then smaller tables
 - Join subsequent tables depending on which table has the most selective filter



The `JOIN` clause itself is very detailed, so I have introduced it here only for reference; however, I would suggest you study some SQL documentation on `JOIN` to learn more about it.

Defining VIEWS in Impala

A `VIEW` in SQL is the result set of a stored query defined with a `SELECT` statement or the `SELECT` portion of an `INSERT` statement. A `VIEW` can represent a subset of data contained in a table by limiting the data set depending on the query statement. In Impala, you can use `VIEW` to your advantage in the following ways:

- You can use `VIEW` in place of lengthy subqueries and repeating the same subquery in many queries
- Reduce maintenance by using `VIEW` in place of complicated queries across multiple applications
- You can issue complex queries with a compact and simple syntax
- You can set up granular security in a table by limiting data access to only a few columns
- You can set up aliases for tables, columns, and `JOIN` results with a more informative name

Here are the common `VIEW`-specific statements in Impala:

- `CREATE VIEW`
- `ALTER VIEW`
- `DROP VIEW`

Loading data from HDFS using the LOAD DATA statement

As you know, data is stored in HDFS and Impala processes this data. So, when you need to perform some **Extract Transform Load (ETL)** activity to load the data from HDFS to Impala tables, you can use `LOAD DATA` statements. The key properties of `LOAD DATA` statements are as follows:

- The loaded data files are moved from HDFS to the Impala data directory
- You can either give a file name from HDFS or a directory name to load all the files into an Impala table; however, a wild card pattern is not supported with the HDFS path

The `LOAD DATA` statement and examples are as follows:

```
LOAD DATA INPATH 'hdfs_file_or_directory_path' [OVERWRITE]
            INTO TABLE tablename
            [PARTITION (partcol1=val1, partcol2=val2 ...)]
```

Examples:

```
CREATE TABLE students (id int, name string);
LOAD DATA INPATH '/user/avkash/students.txt' INTO TABLE students;
```

In the previous example, you have to make sure that the `students.txt` file is located at HDFS in folder `/user/avkash`.

Comments in Impala SQL statements

You can write multiple queries into a file and then let Impala run your queries directly by passing the query file name. When lots of queries are written in a file, you might require adding comments to specify what each query or a group of queries are, and adding comments to the query file will help you achieve your objective. Impala supports the following two kinds of comments with SQL statements:

- `--` Sequence (two dashes): All the text after `--` Sequence is considered as a comment and ignored for processing. This comment is used mostly in commenting a complete line in multiline SQL statements.
- `/* this is a comment */`: The text inside `/*` and `*/` will be considered as a comment and will not be processed. This comment is used mostly within the line but it can be stretched into multiple lines as well.

Built-in function support in Impala

Impala supports lots of built-in functions in various categories, and these functions are used to perform several types of data transformation operations, such as mathematical calculations, string manipulations, and data calculations. You can use built-in functions with SQL queries to avoid post-processing of data; by using these functions you can get fully-formatted, calculated, and type changed data as results. Aggregate functions ignore NULL values rather than returning a NULL result.

These built-in functions can be used directly with a SELECT statement, as shown in the following example:

```
SELECT ABS(-1);
SELECT CONCAT('NFL ', 'American Football');
SELECT POWER(3,3);
SELECT CONCAT('State = ',state_name) FROM states WHERE population
    > 10000000;
SELECT SIN(null);
SELECT POWER(2,null);
SELECT MAX(wheels), AVG(windows) FROM automobiles WHERE year
    <1950;
```

Impala supports the following categories of built-in functions:

- Mathematical functions
- String functions
- Conditional functions
- Date and time functions
- Type conversation functions
- Aggregate functions (which we have already discussed in previous sections)

Now, we will take a look at a few functions from each category and their usage patterns. Let's starts with mathematical functions, described in the following table:

Function name	Usage	Return type
ABS(DOUBLE a)	To return the absolute value of the argument	DOUBLE
COS(DOUBLE a)	To return the cosine of the argument	DOUBLE
BIN(BIGINT a)	To return the binary representation of the integer value	DOUBLE
FLOOR(DOUBLE a)	To return the largest, least, or equal to the argument	INT

Function name	Usage	Return type
PI()	To return the value of constant Pi	DOUBLE
RAND(INT seed)	To return a random value between 0 and 1	DOUBLE

Besides these, there are several other mathematical functions in Impala such as `SIGN`, `SIN`, `SQRT`, `TAN`, `ROUND`, `POW`, `NEGATIVE`, `HEX`, `DEGREES`, and `ASIN`. To study most of the functions, please visit the Impala documentation at Cloudera's website. Now, let's study a few of the common string built-in functions in the following table:

Function name	Usage	Return type
ASCII(STRING str)	To return the numeric ASCII code of the first character	INT
CONCAT(STRING a, STRING b..)	To return a single string representing all the argument values joined together	STRING
LENGTH(STRING s)	To return the length of characters in an argument	INT
REVERSE(STRING a)	To return the reverse string of an argument	STRING

Besides these, there are many other string functions such as `FIND_IN_SET`, `INSTR`, `LOCATE`, `LOWER`, `UPPER`, `LTRIM`, `REPEAT`, `RTRIM`, `SUBSTR`, `TRANSLATE`, `TRIM`, and `UPPER`. Now, we can learn a few conditional functions, as described in the following table:

Function name	Usage	Return type
CASE	An expression to get one or more possible values	Argument
COALESCE	To return the first specified non NULL argument	Argument
IF	To test an expression and then produce the result	Argument
ISNULL	To test if an expression is NULL or not	Argument

Besides these, the other conditional functions are `NVL` and `CASE`. Now it is time to learn a few of the date/time-specific functions, as in the following table:

Function name	Usage	Return type
NOW()	To return the current date and time in UTC	TIMESTAMP
TO_DATE(STRING date)	To convert a date string to a TIMESTAMP value	TIMESTAMP
YEAR(STRING date)	To return the year value from a string date type	INT
DATEDIFF(date1, date2)	To return the number of days between two dates passed as arguments	INT

Other date and time functions not described previously are `DATE_ADD`, `DATE_SUB`, `DAY`, `DAYNAME`, `DAYOFWEEK`, `FROM_UNIXTIME`, `FROM_UTC_TIMESTAMP`, `hour`, `minute`, `month`, `second`, `TO_UTC_TIMESTAMP`, `UNIX_TIMESTAMP`, and `WEEKOFYEAR`.

The type conversion function

For type conversion, Impala uses the `CAST()` function within strict rules regarding data types for functional parameters. The `CAST()` function is mostly used in conjunction with other SQL statements with other functions to explicitly pass the desired data types. The syntax for using the `CAST` operator is shown in the following example:

```
CAST (expression as TYPE)
SELECT CONCAT ('Today is ', 28 , 'October.');
```

The previous SQL statement will generate an error as 28 is used as a numeric value, and the `CONCAT` function only accepts string values. So we can use the `CAST` operator to convert numeric to `STRING` values as follows:

```
SELECT CONCAT ('Today is ', CAST(28 as STRING) , 'October.');
```

I have previously suggested that most of the functions are described in the SQL documentation, so you can look at specific SQL documentation to learn more about Impala built-in functions.

Unsupported SQL statements in Impala

At the time of writing this book, Impala's latest version, 1.1.x, does not support the following SQL features, which are available in HiveQL:

- Nonscalar data types such as `maps`, `array`, and `struct`
- **XML** and **JSON** functions
- The `LOAD DATA` statement to load raw files
- Custom UDF, **User Defined Aggregation Functions (UDAF)**, and **User Defined Table Generating Functions (UDTF)** up to Cloudera Impala 1.1.x. Please check *Appendix, Technology Behind Impala and Integration with Third-party Applications*, on support for these functions in Impala 1.2.0 Beta.
- Sampling, Lateral Views, Roles, Custom SerDes, and Multiple `DISTINCT`

Here is a list of **HiveQL** statements that are not supported in Impala:

- ANALYZE TABLE
- DESCRIBE COLUMN
- DESCRIBE DATABASE
- EXPORT TABLE
- IMPORT TABLE
- SHOW PARTITIONS
- SHOW TABLE EXTENDED
- SHOW TBLPROPERTIES
- SHOW FUNCTIONS
- SHOW INDEXES
- SHOW COLUMNS
- SHOW CREATE TABLE

Summary

By end of this chapter we have covered various SQL statements written using Impala Query Language, which can be used either with the Impala shell or directly from a web interface. When applicable, we have provided small examples of SQL statements, because in the next chapter we will take the learning from this and the previous chapters into a full-scale example, and apply these SQL statements to load data from HDFS and then run queries on it. By the time this book was ready in print, Impala 1.2.0 Beta was released with new features. It is possible that some of the features written in this chapter either work differently or require more attention. Please check the Cloudera documentation for more information regarding this.

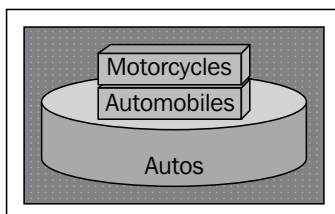
4

Impala Walkthrough with an Example

In this chapter, we will go over a use case to see Impala concepts in action. This way you can experience a real-world scenario using Impala, and understand how and where to use Impala statements in real-world applications. In this chapter, I will be using a scenario as described in the following sections.

Creating an example scenario

We are going to deal with information related to automobiles. We have two data files that contain information about automobiles and motorcycles in two separate text files. The following conceptual image shows that within the **Autos** database, there are two tables named **Motorcycles** and **Automobiles**.



So far, it is imprinted on your mind that Impala is running on DataNode, and the files in our project are stored on HDFS. First we will load these files from HDFS to Impala and then we will use SQL statements to process this information through multiple queries.

Example dataset one – automobiles (automobiles.txt)

Let's take a look at this example dataset, which has a list of automobile names and their properties as defined in the schema. The following is the first text file, which has automobile-specific data:

- **File:** `automobiles.txt`
- **Schema:** `make, model, year, fuel-type, numofdoors, design, type, cylinders, horsepower, city_hwy_mpg, price`

Here is the data in the `automobiles.txt` file:

```
Audi,A4,2011,gas,4,edan,casual,6,476,22-30,45000
Jeep,Compass,2007,gas,3,suv,sport,6,170,24-32,22000
Dodge,Challenger,2013,gas,4,coupe,casual,6,210,20-30,28000
Chevrolet,Volt,2014,electric,4,edan,casual,0,180,35-40,35000
Toyota,Prius,2013,hybrid,4,edan,casual,4,134,51-48,32000
BMW,M3,2010,gas,2,coupe,sport,6,300,18-28,41000
BMW,X5,2005,gas,4,suv,sport,6,265,19-26,55000
Toyota,Camry,2009,gas,4,edan,casual,6,178,25-35,26000
Toyota,Camry,2014,hybrid,4,edan,sport,6,200,43-39,30000
Honda,Civic,2013,gas,2,coupe,sport,4,140,28-36,18000
Nissan,Leaf,2014,electric,4,edan,casual,0,107,129-102,29000
Audi,Q7,2013,gas,4,suv,sport,6,333,16-22,60000
Audi,A7 TDI,2014,diesel,4,edan,sport,6,240,30-25,58000
Mercedes,CLA,2013,gas,4,edan,casual,4,208,22-28,29000
Fisker,Karma,2014,electric,2,coupe,sport,0,260,85-90,100000
```

Example dataset two – motorcycles (motorcycles.txt)

Now let's take a look at another example dataset, which has a list of motorcycle names and their properties as defined in the schema. The following is the second text file, which includes motorcycles-specific data:

- **File:** `motorcycles.txt`
- **Schema:** `make, model, year, fuelType, wheels, body, style, cc_rpm, highSpeed, auto, price`

Here is the data in the `motorcycles.txt` file:

```
Honda,CBR600,1990,gas,2,casual,sport,599,165,false,12000
BMW,R1200RT,1990,gas,2,casual,sport,1170,135,false,20000
Honda,CB900,1995,gas,2,casual,sport,919,135,false,10000
```

```

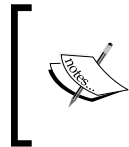
Honda,VFR400,1998,gas,2,casual,sport,399,130,false,8000
KTM,Super Duke,2005,gas,2,casual,sport,999,151,false,25000
Triumph,Speed Triple,2001,gas,2,deluxe,sport,1050,150,false,23000
Suzuki,RGV250,2000,gas,2,deluxe,sport,249,127,false,7500
Triumph,Daytona,gas,2005,2,deluxe,sport,675,156,false,16000
Triumph,Street Triple,2010,gas,2,casual,sport,123,141,true,12000
Ducati,1098,2010,gas,2,deluxe,sport,1099,180,false,30000
Harley,Tri Glide,2012,gas,3,deluxe,luxury,1600,180,false,35000
Harley,Iron,2012,gas,casual,luxury,1500,200,false,14000
Bramo,Icon,2012,electric,2,casual,sport,4500,80,false,20000
Zero,Police,2013,electric,2,casual,sport,4300,95,false,25000
Can-am,spider,2014,gas,3,deluxe,luxury,998,120,false,22000

```

Data and schema considerations

You cannot use a dash (-) with the schema name; instead use an underscore. In the preceding example's dataset schema, you can see some of the schema definition are using underscores while others are not.

In your data, you must not have spaces between a separator; otherwise, numeric values cannot be processed correctly or you will get NULL instead.



Before we go further, I would like to make sure it is understood that the preceding data is created by me, specifically for reference purpose to demonstrate Impala query processing. This data is completely fabricated and does not reflect any correct information.

Commands for loading data into Impala tables

This section covers activity to load data into Impala tables in two steps. First step moves the file, which we have created previously, from the local filesystem to HDFS. In the second step, the data is transferred into Impala tables, from the source file located on HDFS.

HDFS specific commands

Now we will make sure that the preceding files are located in our Linux files system as follows:

```

[cloudera@localhost ~]$ ls -l *.txt
-rw-rw-r-- 1 cloudera cloudera 985 Oct 15 18:48 automobiles.txt
-rw-rw-r-- 1 cloudera cloudera 932 Oct 15 18:49 motorcycles.txt

```

Our next step is to move these files from the local filesystem to HDFS in separate folders. Depending on your Hadoop version, you can use either `Hadoop fs` or `hdfs dfs`; however, I have used `hdfs dfs` as follows:

```
[cloudera@localhost ~]$ hdfs dfs -mkdir /user/cloudera/automobiles
[cloudera@localhost ~]$ hdfs dfs -mkdir /user/cloudera/motorcycles
[cloudera@localhost ~]$ hdfs dfs -ls /user/cloudera/
Found 2 items
drwxr-xr-x - cloudera cloudera          0 2013-10-15 19:16 /user/
cloudera/automobiles
drwxr-xr-x - cloudera cloudera          0 2013-10-15 19:16 /user/
cloudera/motorcycles
[cloudera@localhost ~]$ hdfs dfs -moveFromLocal automobiles.txt /user/
cloudera/automobiles/automobiles.txt
[cloudera@localhost ~]$ hdfs dfs -moveFromLocal motorcycles.txt /user/
cloudera/motorcycles/motorcycles.txt
[cloudera@localhost ~]$ hdfs dfs -ls /user/cloudera/motorcycles/
Found 1 items
-rw-r--r--  3 cloudera cloudera          932 2013-10-15 19:19 /user/
cloudera/motorcycles/motorcycles.txt
[cloudera@localhost ~]$ hdfs dfs -ls /user/cloudera/automobiles/
Found 1 items
-rw-r--r--  3 cloudera cloudera          985 2013-10-15 19:17 /user/
cloudera/automobiles/automobiles.txt
```

Now, we will load the preceding data into two separate tables in two different steps, to learn various ways of loading data. The tables we are using here are external tables instead of internal. For automobile data, I will load them directly from a script into the `automobiles` table; and then I will load motorcycle data in the `motorcycles` table inside the Impala shell. In the script, I will add another empty table, `automakers`. Later, we will join a list of automakers from both tables. All of this processing will be done in a database named `autos`.

Loading data into the Impala table from HDFS

Here is the SQL script to create a database `autos` first, create the `automobiles` table, and then load the whole dataset from HDFS. I am also creating an empty table `automakers` in the `autos_script.sql` script as follows:

```
USE default;
DROP DATABASE IF EXISTS autos;
CREATE DATABASE autos;
USE autos;
DROP TABLE IF EXISTS automobiles;
CREATE EXTERNAL TABLE automobiles
```

```
(
  make STRING,
  model STRING,
  year INTEGER,
  fuelType STRING,
  numOfDoors INTEGER,
  design STRING,
  type STRING,
  cylinders INTEGER,
  horsepower INTEGER,
  city_hwy_mpg STRING,
  price FLOAT
)
ROW FORMAT DELIMITED FIELD TERMINATED BY ',' STORED AS TEXTFILE
LOCATION '/user/cloudera/automobiles/automobiles.txt';
CREATE EXTERNAL TABLE IF NOT EXISTS automakers
(
  autoMaker STRING
)
ROW FORMAT DELIMITED FIELD TERMINATED BY ',';
```

Let's understand the preceding script. In the first line, we are setting the default database and then removing the `autos` database if it exists. Next, we are creating a totally new `autos` database and then using the `auto` database. After that, we are dropping the `automobiles` table if it exists. Then, we are creating a new `automobiles` table using the schema as defined earlier. Then, we are passing the data source text files so the table can be populated with appropriate content. At last, we will also create another table named `automaker` and use the `IF NOT EXISTS` syntax. This syntax means that this particular table is created only if it does not exist.

Now, we are going to execute the preceding SQL script with the Impala shell using the following command syntax:

```
$impala-shell -i Impala-Server-Name:PORT -f SQL_Script_Name.sql
```

Here is the execution of the preceding SQL script:

```
[cloudera@localhost ~]$ impala-shell -i localhost.localdomain:21000 -f
autos_setup.sql
Connected to localhost.localdomain:21000
Server version: impalad version 1.0.1 RELEASE (build
df844fb967cec8740f08dfb8b21962bc053527ef)
Query: use default
Query: drop DATABASE IF EXISTS autos
Query: create DATABASE autos
Query: use autos
```

```
Query: drop TABLE IF EXISTS automobiles
Query: create EXTERNAL TABLE automobiles ( make STRING, model STRING,
autoyear INTEGER, fuelType STRING, numOfDoors INTEGER, design STRING,
autoType STRING, cylinders INTEGER, horsePower INTEGER, city_hwy_mpg
STRING, price FLOAT) ROW FORMAT DELIMITED FIELDS TERMINATED BY ','
STORED AS TEXTFILE LOCATION '/user/cloudera/automobiles'
Query: drop TABLE IF EXISTS automakers
Query: create EXTERNAL TABLE automakers (autoMaker STRING)
```

Launching the Impala shell

Now, we will launch the Impala shell and verify if the `autos` database is created and both `automobiles` and `automakers` are created or not. Let's launch the Impala shell first as follows:

```
$impala-shell
```

Once the Impala shell is started, I have used the `autos` database first and verified our steps from the preceding script.

Database and table specific commands

Now, we will use some database and table commands to verify the execution of the script in the previous sections. Once we see that the database is created and data is loaded, we can push queries from the Impala shell against our example database as follows:

```
[Hadoop.testdomain:21000] > USE autos;
[Hadoop.testdomain:21000] > SHOW tables;
Query finished, fetching results ...
+-----+
| name      |
+-----+
| automakers |
| automobiles |
+-----+
Returned 2 row(s) in 0.11s
```

In the preceding code, we are first using the `autos` databases and then listing all the tables within this selected database.

In the next step, we will understand each table's schema using the `describe` command as shown in the following two examples. The first example is as follows:

```
[Hadoop.testdomain:21000] > describe automakers;
Query finished, fetching results ...
+-----+-----+-----+
| name      | type   | comment |
+-----+-----+-----+
| automaker | string |          |
+-----+-----+-----+
Returned 2 row(s) in 0.46s
```

The following is the second example:

```
[Hadoop.testdomain:21000] > describe automobiles;
Query finished, fetching results ...
+-----+-----+-----+
| name      | type   | comment |
+-----+-----+-----+
| make      | string |          |
| model     | string |          |
| autoyear  | int    |          |
| fueltype  | string |          |
| numofdoors | int    |          |
| design    | string |          |
| autotype  | string |          |
| cylinders | int    |          |
| horsepower | int    |          |
| city_hwy_mpg | string |          |
| price     | float  |          |
+-----+-----+-----+
Returned 11 row(s) in 0.54s
```

In the preceding code snippets, you can see how the `describe` command shows each field's name and field-type information.

Now, I will create another table named `motorcycles` and load data from the `/user/cloudera/motorcycles/motorcycles.txt` file located in HDFS into the table as follows:

```
[Hadoop.testdomain:21000] > CREATE EXTERNAL TABLE IF NOT EXIST
motorcycles ( make STRING, model STRING, year INTEGER, fuelType
STRING, wheels INTEGER, body STRING, style STRING, cc_rpm INTEGER,
highSpeed INTEGER, automatic BOOLEAN, price FLOAT ) ROW FORMAT
DELIMITED FIELDS TERMINATED BY ',' STORED AS TEXTFILE LOCATION '/user/
cloudera/motorcycles' ;
```

```
Query: create EXTERNAL TABLE motorcycles ( make STRING, model STRING,
year INTEGER, fuelType STRING, wheels INTEGER, body STRING, style
STRING, cc_rpm INTEGER, highSpeed INTEGER, automatic BOOLEAN, price
FLOAT ) ROW FORMAT DELIMITED FIELDS TERMINATED BY ',' STORED AS
TEXTFILE LOCATION '/user/cloudera/motorcycles'
```

In the preceding command, we are creating an external table only if it does not exist. And then we will pass the `motorcycles` text content from the text file to populate the table.

Now, we can check the schema of the newly created `motorcycles` table using the `describe` command as follows:

```
[Hadoop.testdomain:21000] > describe motorcycles;
Query finished, fetching results ...
+-----+-----+-----+
| name      | type      | comment |
+-----+-----+-----+
| make      | string    |         |
| model     | string    |         |
| year      | int       |         |
| fueltype  | string    |         |
| wheels    | int       |         |
| body      | string    |         |
| style     | string    |         |
| cc_rpm    | int       |         |
| highspeed | int       |         |
| automatic | boolean   |         |
| price     | float     |         |
+-----+-----+-----+
Returned 11 row(s) in 1.22s
```

SQL queries against the example database

Now, let's list all the items from both the `automobiles` and `motorcycles` table. Because we have a long list of items, to save page space we will limit the output to only the top five items, as follows:

```
[Hadoop.testdomain:21000] > select * from automobiles limit 5;
Query finished, fetching results ...
```

```

+-----+-----+-----+-----+-----+-----+
+-----+-----+-----+-----+-----+-----+
| make      | model      | autoyear | fueltype | numofdoors | design |
| autotype  | cylinders  | horsepower | city_hwy_mpg | price  |
+-----+-----+-----+-----+-----+-----+
+-----+-----+-----+-----+-----+-----+
| Audi      | A4         | 2011     | gas      | 4          | sedan  |
casual    | 6          | 476      | 22-30    | 45000      |
| Jeep      | Compass    | 2007     | gas      | 3          | suv    |
sport     | 6          | 170      | 24-32    | 22000      |
| Dodge     | Challenger | 2013     | gas      | 4          | coupe  |
casual    | 6          | 210      | 20-30    | 28000      |
| Chevrolet | Volt       | 2014     | electric | 4          | sedan  |
casual    | 0          | 180      | 35-40    | 35000      |
| Toyota    | Prius      | 2013     | hybrid   | 4          | sedan  |
casual    | 4          | 134      | 51-48    | 32000      |
+-----+-----+-----+-----+-----+-----+
+-----+-----+-----+-----+-----+-----+
Returned 5 row(s) in 1.77s

```

In the next step, we will use the same select statement with a variation to list only those motorcycles that have autoyear above 2010, as shown in the following code snippet:

```

[Hadoop.testdomain:21000] > select * from motorcycles where year >
2010;
Query finished, fetching results ...
+-----+-----+-----+-----+-----+-----+-----+
+-----+-----+-----+-----+-----+-----+
| make      | model      | year | fueltype | wheels | body   | style
| cc_rpm    | highspeed  | automatic | price |
+-----+-----+-----+-----+-----+-----+
+-----+-----+-----+-----+-----+-----+
| Harley    | Tri Glide  | 2012 | gas      | 3      | deluxe | luxury
| 1600      | 180        | false | 35000    |
| Harley    | Iron       | 2012 | gas      | NULL   | luxury | 1500
| 200       | NULL       | NULL  | NULL     |
| Bramo     | Icon       | 2012 | electric | 2      | casual | sport
| 4500      | 80         | false | 20000    |
| Zero      | Police     | 2013 | electric | 2      | casual | sport
| 4300      | 95         | false | 25000    |
| Can-am    | spider     | 2014 | gas      | 3      | deluxe | luxury
| 998       | 120        | false | 22000    |
+-----+-----+-----+-----+-----+-----+
+-----+-----+-----+-----+-----+-----+
Returned 5 row(s) in 0.61s

```


Now, we will try to get a list of unique automakers from both the `automobiles` and `motorcycles` tables by using the `distinct` command as follows:

```
[Hadoop.testdomain:21000] > select distinct(make) from automobiles;  
Query finished, fetching results ...
```

```
+-----+  
| make  |  
+-----+  
| Mercedes |  
| Audi    |  
| Nissan  |  
| Dodge   |  
| BMW     |  
| Toyota  |  
| Fisker  |  
| Honda   |  
| Chevrolet |  
| Jeep    |  
+-----+
```

```
Returned 10 row(s) in 0.68s
```

And for the `motorcycles` table, we will use the `distinct` command as follows:

```
[Hadoop.testdomain:21000] > select distinct(make) from motorcycles;  
Query finished, fetching results ...
```

```
+-----+  
| make  |  
+-----+  
| Can-am |  
| Suzuki |  
| BMW    |  
| Zero   |  
| KTM    |  
| Brama  |  
| Honda  |  
| Triumph |  
| Ducati |  
| Harley |  
+-----+
```

```
Returned 10 row(s) in 0.48s
```

SQL join operation with the example database

Now, we will try to check which maker is common between both tables and their models by using the `join` SQL command as follows:

```
[Hadoop.testdomain:21000] > select automobiles.make, automobiles.
model, motorcycles.make,motorcycles.model from automobiles JOIN
motorcycles USING (make);
Query finished, fetching results ...
+-----+-----+-----+-----+
| make  | model | make  | model |
+-----+-----+-----+-----+
| BMW   | M3    | BMW   | R1200RT |
| BMW   | X5    | BMW   | R1200RT |
| Honda | Civic | Honda | VFR400  |
| Honda | Civic | Honda | CB900   |
| Honda | Civic | Honda | CBR600  |
+-----+-----+-----+-----+
Returned 5 row(s) in 0.40s
```

Now, we will insert automakers from both the `automobiles` and `motorcycles` tables into the **automakers** table by using the `Insert Overwrite` and `Insert Into` SQL statements as follows:

```
[Hadoop.testdomain:21000] > INSERT OVERWRITE TABLE automakers SELECT
distinct(make) from automobiles;
Inserted 10 rows in 2.00s
[Hadoop.testdomain:21000] > INSERT INTO TABLE automakers SELECT
distinct(make) from motorcycles;
Inserted 10 rows in 0.71s
```

Using various types of SQL statements

In the `automobiles` table, I have included a field named `city_hwy_mpg`. It includes miles per gallon in the city and highway, separated by a dash (-). Here, I want to show you how to use the `STRING` manipulation to get both highway and city miles per gallon values along with the `LIMIT` and `WHERE` clauses:

```
[Hadooptestdomain:21000] > select city_hwy_mpg, substr(city_hwy_
mpg,1,2), substr(city_hwy_mpg, instr(city_hwy_mpg, "-")+1 , 5) from
automobiles WHERE price > 15000 LIMIT 2;
Query finished, fetching results ...
```

```
+-----+-----+-----+
| city_hwy_mpg | substr(city_hwy_mpg, 1, 2) | substr(city_hwy_mpg,
instr(city_hwy_mpg, '-') + 1, 5) |
+-----+-----+-----+
| 22-30        | 22                          | 30
|
| 24-32        | 24                          | 32
|
+-----+-----+-----+
Returned 2 row(s) in 0.35s
```

In the preceding code snippet, the `substr` SQL command is used to select the first two letters that represent the `city_mpg` value. After this, `instr` is used to find the location of `-` and then the same `substr` is used to get the `hwy_mpg` part of the value.

Now, let's use the `COUNT` and `DISTINCT` clause to collect unique automakers as follows:

```
[Hadoop.testdomain:21000] > select count(distinct(make)) from
automobiles;
Query finished, fetching results ...
+-----+
| count(distinct make) |
+-----+
| 10                   |
+-----+
Returned 1 row(s) in 0.48s
```

The preceding `distinct` SQL command removes the duplicate `make` values to make them unique and after that the `count` SQL command counts all these items to return the result.

Now, you will see how to use the `EXPLAIN` clause to understand a query execution as follows:

```
[Hadoop.testdomain:21000] > explain select * from motorcycles where
price > 10000 and price < 20000;
PLAN FRAGMENT 0
  PARTITION: UNPARTITIONED
  1:EXCHANGE
    tuple ids: 0
PLAN FRAGMENT 1
  PARTITION: RANDOM
  STREAM DATA SINK
```

```
EXCHANGE ID: 1
UNPARTITIONED
0:SCAN HDFS
  table=autos.motorcycles #partitions=1 size=889B
  predicates: price > 10000.0, price < 20000.0
  tuple ids: 0
```

The `explain` command takes the remaining part of the query and then shows how it is going to handle the query. If you pass the `select` command with `explain`, you will get results about the query execution plan from the optimizer. Depending upon your query, `explain` will show you if a partition is used, where the data is stored, how the results are fetched from the table, and if any index is used or not. You can use the result from the `explain SQL` command to do an interactive analysis of your query and then modify it for faster execution as and if needed. You can also use the results from `explain` to troubleshoot specific issues.

Summary

In this chapter, you have learned how to use various SQL statements in Impala. The example covers various aspects of data transformation and data processing. I hope by using the preceding examples, you could learn the SQL statements and functions described in detail in *Chapter 3, The Impala Query Language and Built-in Functions*. You can continue working on this `autos` database and the `automobiles` and `motorcycles` table to learn other SQL clauses and built-in functions as well. The main reason you should use Impala instead of Hive is the great increase in query-processing speed. The execution time for the first query in both Apache Hive and Impala will be nearly the same. However, for the subsequent queries, you will see a tremendous increase in the speed of query execution. This results in new real-time performance to justify your use of Impala.

In *Chapter 5, Impala Administration and Performance Improvements*, we are going to learn how to administer and manage Impala to improve performance and keep it running in the high-availability mode.

5

Impala Administration and Performance Improvements

After going through all the examples in the previous chapter, I am sure you are able to process data through Impala queries. Now you will have questions about how to improve query performance, and this is one of the two key objectives of this chapter. The other objective is to show effective management of our Impala cluster that will keep it up and running.

In this chapter, we will cover two important topics: Impala administration and performance improvements. Within the *Impala administration* section, I will show you how you can administer Impala using Cloudera Manager. After that, using debug web server, I will teach you to verify Impala-specific information for its correctness. We will see Impala logs and daemons using the statestore UI. The next part of the Impala admin is about Impala High Availability. We will learn key traits of how to keep Impala going in the event of a problem.

In the *Improving performance* section, we will cover various ways to improve and tune query performance. We will learn to test Impala queries to understand if they are performing well or not and, if not, what you can do to improve their performance—either fine-tune the cluster or modify the query statement or its execution. Finally, let's start with Impala administration.

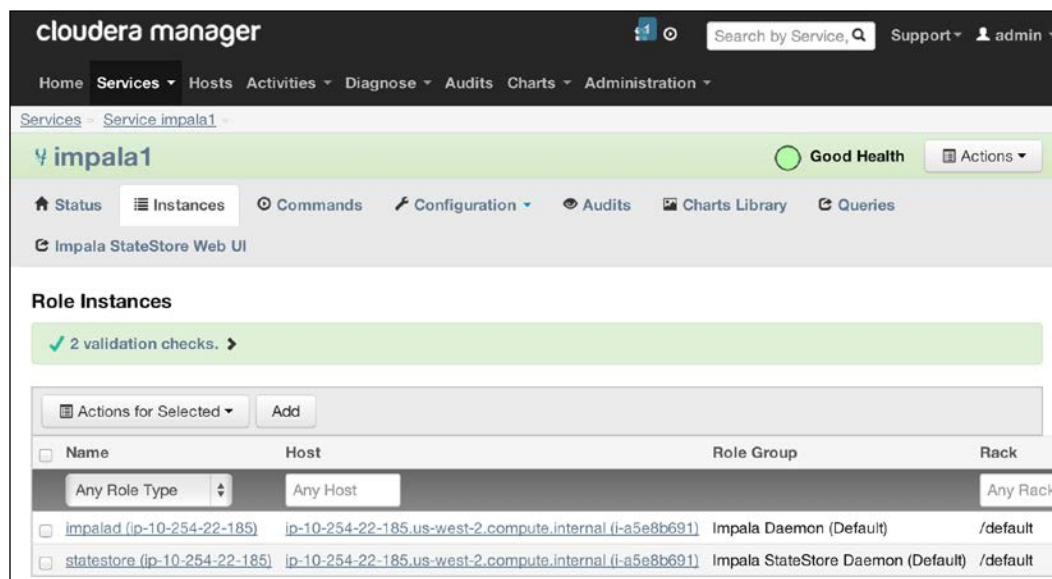
Impala administration

We have already discussed in previous chapters that you can install and run Impala with or without Cloudera Manager; however, for simplicity, it is good to have Cloudera Manager manage your Impala cluster. This will help you spend your crucial time working with data transformation rather than cluster administration. In this chapter, I will assume that you are managing your Impala cluster using Cloudera Manager and provide more information based on that assumption.

Administration with Cloudera Manager

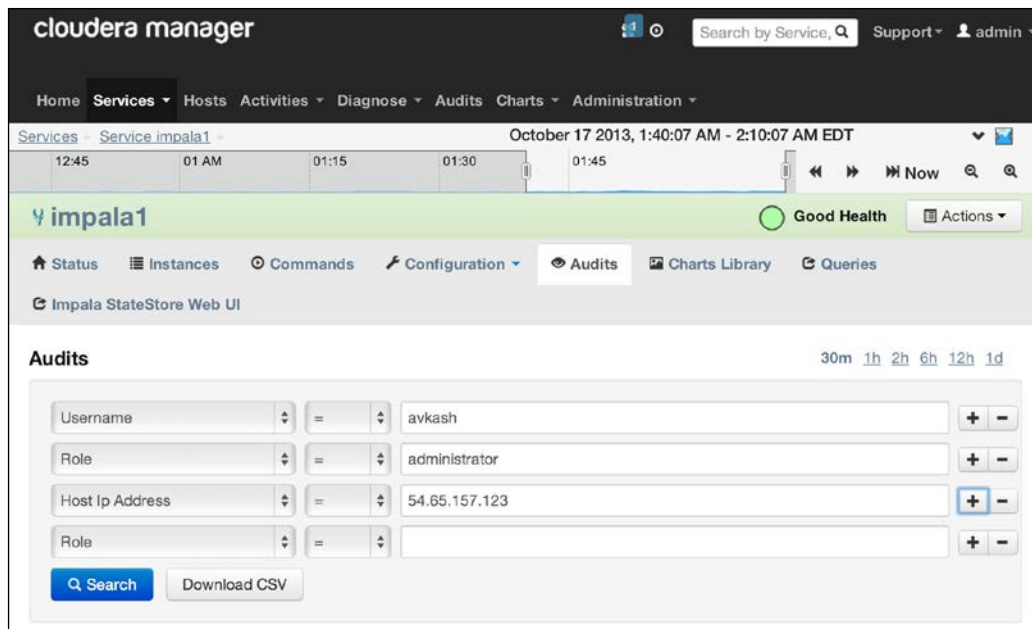
While describing Cloudera Manager in detail is beyond the scope of this book, I will try to provide some guidance to you so you can use Cloudera Manager to administer Impala. Once the Cloudera Manager web-based user interface is in front of you, just select **Service impala1** from the **Services** list, and then you have multiple ways to start, stop, and restart both the Impala daemon(s) and statestore service directly from there. You can also change the Impala configuration, view log files, manage Impala nodes, and troubleshoot some of the problems just by opening the Impala debugging interface.

In the next few screenshots, let's see how you can use the Cloudera Manager web-based user interface to manage Impala:

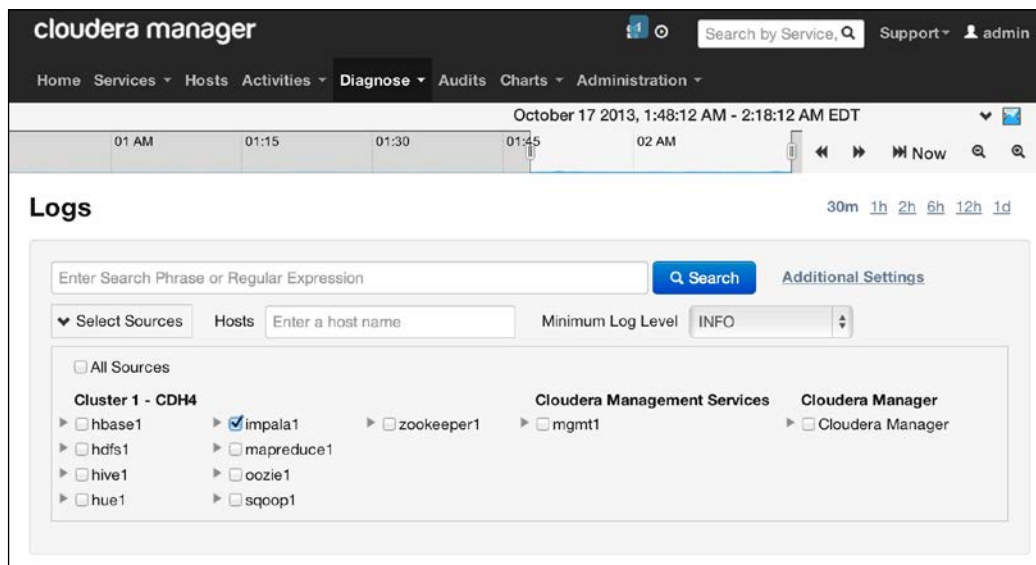


In the preceding screenshot, you can see the list of Impala daemons and statestore services running that can be managed. In the **Queries** tab, you can search the SQL statement directly from the Impala web interface and look at various graphs and charts to understand query performance.

In the following screenshot, you can learn configuring Impala auditing features with Impala 1.1.x and above. This configuration helps you to input an auditing scheme based on **Username**, **Role**, and **Host Ip Address** and, based on that, you can analyze the logs directly on the web or download them for further processing.

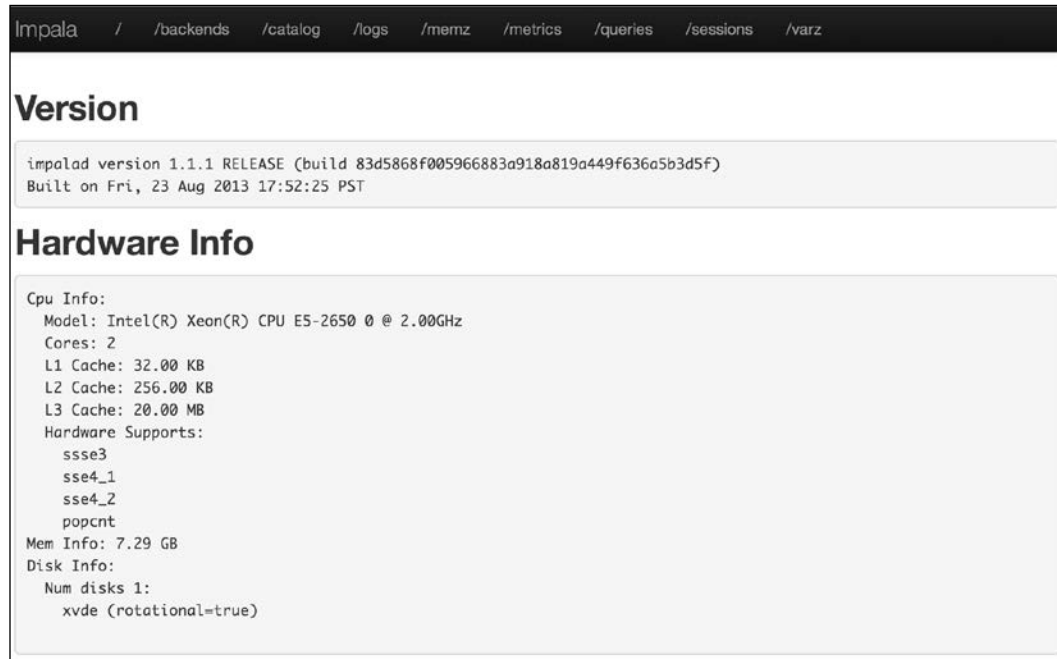


I have detailed Impala logging in *Chapter 6, Troubleshooting Impala*. However, because in this chapter we are talking about Impala administration, it is appropriate to inform you that you can use Cloudera Manager to configure Impala logging based on Impala services on various nodes and then read those logs at your convenience, as shown in the following screenshot:



The Impala statestore UI

When the Impala server is running with Cloudera Manager, you can open the Impala debugging web interface at port 25000, as shown in the following screenshot. You can also verify the Impala configuration on the **/varz** page and logs on the **/logs** page, query metrics on **/metrics**, and do many other things.



Impala High Availability

Impala runs on DataNodes and takes advantage of any **High Availability (HA)** configuration available to DataNodes. Impala uses data stored in HDFS, which is the distributed data storage layer in Hadoop, shared between NameNode and DataNodes. Hadoop does provide the NameNode High Availability configuration; if you would like to learn more about it, I would recommend looking at the Hadoop documentation.

To make Impala High Availability, the best option is to take advantage of the HDFS HA feature. As an Impala cluster administrator, you can upgrade a Hive metastore to use HDFS HA features. Because Impala depends on Hive metastore, in the event the primary metastore is not available, it will instantly be available on the other HDFS HA node without interrupting any significant downtime.

Single point of failure in Impala

The best way to start this section is that there is no single point of failure in Impala, meaning every and all Impala daemons are capable of executing incoming queries. A specific node failure will impact only those query segments that were distributed on the affected machine because one single query is distributed across multiple nodes. In this situation, re-execution of the same query will allow the system to recover from the problem. For Hadoop cluster stability, it is suggested to run various Impala components on DataNode. Running Impala on NameNode is not suggested because in an unfortunate event, Impala on NameNode could cause overall NameNode failure, which ultimately could impact Hadoop cluster stability. Running Impala on DataNode means as long as the Hadoop cluster is up and running smoothly, the Impala cluster will function well, even if there is an issue with failure of a single or a few DataNodes. Also, if NameNode is highly available, the Impala cluster will be highly available as well.

One thing to remember on the same account is that Impala has dependency on statestore, which runs only on a single machine. If statestore is not available, it will not bring Impala to a complete shutdown; however, it does impact its operation and query distribution.

Improving performance

In this section, we will learn a few helpful pointers to improve performance by modifying Impala daemon execution and the underlying platform where Impala performs user actions.

Enabling block location tracking

When queries are executed in Impala, data is read from HDFS that is distributed across multiple DataNodes in the form of data blocks. If Impala knows more information about these data blocks on HDFS, the data can be read faster and queries can achieve faster execution. To enable block location tracking for Impala, you just need to perform the following steps:

1. Modify the HDFS configuration `hdfs-site.xml` as follows:

```
<property>
  <name>dfs.datanode.hdfs-blocks-metadata.enabled</name>
  <value>true</value>
</property>
```
2. Copy `hdfs-site.xml` and `core-site.xml` from the Hadoop cluster to each Impala node into the Impala configuration folder, `/etc/impala/conf`.
3. Restart all DataNodes in your cluster.

Enabling native checksumming

Computing data checksum for very large amounts of data could add a significant amount of time. So having a native library to perform checksum helps improve the performance. You can use the following information to enable native checksumming in Impala:

- If Impala is installed using Cloudera Manager, **native checksumming** is configured automatically and no action is needed.
- To enable native checksumming on your self-installed Impala, you must build and install the Hadoop native library, `libhadoop.so`. If this library is not available, you might receive the following message in Impala logs, indicating native checksumming is not enabled:

```
"Unable to load native-hadoop library for your platform... using built-in-java classes where applicable"
```

Enabling Impala to perform short-circuit read on DataNode

Short-circuit read means reading data locally from the filesystem instead of communicating first with DataNode, and it definitely improves performance. You must have Cloudera CDH 4.2 or higher to achieve faster and compatible short-circuit reading. The following guideline is provided based on the assumption that you have Cloudera CDH 4.2 or higher installed:

1. Modify `hdfs-site.xml` on each Impala node as follows:

```
<property>
  <name>dfs.client.read.shortcircuit</name>
  <value>true</value>
</property>
<property>
  <name>dfs.domain.socket.path</name>
  <value>/var/run/hadoop-hdfs/dn._PORT</value>
</property>
<property>
  <name>dfs.client.file-block-storage-locations.timeout</name>
  <value>3000</value>
</property>
```
2. Make sure that `/var/run/hadoop-hdfs/` is group writable for root users.
3. Copy `hdfs-site.xml` and `core-site.xml` from the Hadoop configuration to each Impala node configuration at `/etc/impala/conf`.
4. Restart all DataNodes.

Adding more Impala nodes to achieve higher performance

It is a fact that Impala performance improves if more nodes are added to the cluster. In the same way, Hadoop performance improves by adding more DataNodes and TaskTrackers. Having more nodes in the Hadoop cluster will distribute the data to more clusters, and queries will have more distribution, which ultimately will return higher performance.

Optimizing memory usage during query execution

You can improve query performance by restricting the amount of memory consumed by a query during its execution and you can do that by setting the `-mem_limits` flag when starting Impala daemon. This flag will restrict the memory consumed only by a query; however, there is still memory available for starting Impala to cache metadata and perform other startup actions.

Query execution dependency on memory

You might wonder about memory limitation impact on query execution as Impala has a strong dependency on available memory. If dataset size exceeds the available memory in a machine, the query will fail. The memory usages in Impala are not directly based on the input dataset size; instead it varies depending on types of query. An aggregation will require memory equivalent to the number of rows after grouping; however, join queries require memory equivalent to the combined size of remaining tables excluding the biggest table.

Using resource isolation

If you are using Cloudera Manager, you have the ability to implement resource isolation using the cgroups mechanism and it can be achieved by configuring Cloudera Manager. For more information, please read the Cloudera Manager documentation on resource isolation.

Testing query performance

Most user time is spent writing and executing queries in Impala. To understand if your Impala cluster is performing optimally, you usually measure query execution time before and after fine-tuning the Impala cluster or your query. The difference between both measurements explains if you have achieved any positive improvements. Let's learn how to measure query execution time precisely to make proper judgments.

Benchmarking queries

When processing terabytes of data from multiple nodes, a query runs for a long time. If you are printing a query output for a console, the time to render the query output on the console is still part of the query execution. It is suggested that you disable the query output on the console by using the `-B` option with the query. This is because you can get the closest execution time. The other option is to save query results in a file using the `-o` option.

Verifying data locality

We have repeatedly seen that to achieve maximum performance with Impala, the query must be distributed on every node in the cluster. You can design a query to be executed on all the nodes in the cluster; however, how can you check if the query actually ran on all nodes? We are going to find the answer to this question in this section.

To find out if a query is executed on all nodes, you will have to dig inside the Impala logs. Make sure you have Impala logging enabled and, after executing the query, open the logs either on an editor or using Cloudera Manager or Navigator. In the logs, if you find the following line, it means the query is not distributed and it is not running on other nodes:

```
Total remote scan volume = 0
```

You can search for the presence of `remote scan` in the log files and, based on its occurrence, you can troubleshoot this problem on your Impala cluster. More information related to troubleshooting this problem is explained in *Chapter 6, Troubleshooting Impala*.

Choosing an appropriate file format and compression type for better performance

Impala is used to process large amounts of data stored in your Hadoop cluster. There is no limitation in Hadoop about what type of data can be stored; however, to improve data access performance in Hadoop, some file types and compression provide better results than others. Impala can query most of the popular structured and unstructured file formats available in Hadoop along with compression used in a file. Here is a list of the supported file formats and compression types in Impala:

File type	File format	Compression type
Text	Unstructured	LZO
Avro	Structured	GZIP, BZIP2, deflate, Snappy
RCFile	Structured	GZIP, BZIP2, deflate, Snappy
SequenceFile	Structured	GZIP, BZIP2, deflate, Snappy
Parquet	Structured	GZIP, Snappy (Default)

Now let's take a look at how choosing a proper file format can improve performance in Impala:

- Sometimes the original file format in which data is stored does not provide the required performance. The possible solution here is to create a new table with a different file format or compression, and then use the `INSERT` statement to perform a one-time conversion. This new table will provide comparatively better performance if you have chosen a new format or compression carefully.
- Processing data, which is compressed, requires disk I/O and CPU cycles to read and uncompress. However, if data were uncompressed, only the disk I/O would comprise the primary cost during processing. So if the application architecture supports processing, uncompressed data does expedite the performance. With uncompressed data storage, you will end up taking lots of space on the disk compared to compressed data. So, you will need to take storage cost into consideration with performance gain.
- Sometimes, changing the file format or compression does not yield any performance gain; rather it slows down the processing comparatively. In this scenario, just using the original file and compression format is fine. So, the lesson here is to understand the file and compression formats properly and then choose them to derive better performance.



Chapter 7, *Advanced Impala Concepts*, has more information about various file formats and compression types and how to use them in Impala.

Fine-tuning Impala performance

In this section, let's review a few key factors that affect Impala performance.

Partitioning

In this method, data is physically divided from frequently queried fields or columns into different values. This way, when a query is executed, it processes only a specific partition or a portion of the data, achieving significant faster results than the full dataset. In general, data files specific to a single table reside in a single directory. Using partitioning, you can distribute the data in a way such that a fraction of data should be read, depending on the query and its data-limiting clause. Once a partition is applied to a table, the data is physically loaded on a different location on the disk based on the query parameter, which provides faster access to the data when queried by the `SELECT` statement and the partition name.

You can write `TABLE` specific Impala SQL statements as follows to take advantage of the `PARTITIONED BY` method:

- Add a partition when creating a table by adding the `PARTITIONED BY` clause as follows:

```
CREATE TABLE [...] PARTITIONED BY
```

- You can modify an existing table to support partitions by using the `ALTER TABLE` statement followed by `PARTITIONED BY` as follows:

```
ALTER TABLE [...] PARTITIONED BY
```

Join queries

It is a well-known fact that a query operating on multiple tables using the `JOIN` operation will take a long time to finish if it is not written correctly. Here are some techniques you can use in this regard:

- A well-known technique for fine-tuning an Impala `join` query is to specify the tables in an optimal order by first having the table that has the maximum number of records or rows, which are part of the result set. After that, select the next largest table in terms of the number of rows in the results set, and finally the smallest table.

- Another method to optimize a faster JOIN operation is to use the HINT clause with JOIN to select a specific Impala query planner. When a SELECT statement based on JOIN is given to Impala to execute, the Impala query planner works on it to find the best strategy. The Impala query planner first checks metadata and the number of records in the result set for each table in the SELECT statement and then chooses an appropriate JOIN strategy. You can get this information by using the EXPLAIN clause with your query. If you think that changing the JOIN strategy will be helpful, you can use the HINT clause as follows to apply the specific JOIN type. You can learn more about HINT in any detailed SQL documentation:

```
SELECT table1.field1, table2.field1 FROM table1 JOIN  
[BROADCAST | SHUFFLE] table2 ON [condition...]
```

The SQL JOIN operation itself is very large and requires a great deal of understanding to optimize. While the preceding information is good for reference purposes, I would suggest reading more details on it in an external reference document to achieve optimum performance with JOIN queries.

Table and column statistics

In the previous section, we talked about the query planner. Now, we will learn how the query planner decides which strategy is best for it. The query planner uses individual column statistics by getting metadata from the metastore if it is available. All the columns, which are part of the result set in the JOIN query, are calculated for all the records, which helps the query planner to make its decision.

The Impala query planner also uses statistics for all the tables and partitions from the metastore and, based on this information, it makes a decision. These table, column, and partition statistics can be gathered using the ANALYZE TABLE statement by passing the table name. I would like to inform you that the Impala query planner does not create this information; instead it depends on Hive for this and the ANALYZE TABLE statement does work on Hive Shell only for now. The syntax of this SQL statement is as follows:

```
ANALYZE TABLE table_name COMPUTE STATISTICS FOR COLUMNS all_column_  
list;  
ANALYZE TABLE table_name PARTITION (partition_specs) COMPUTE  
STATISTICS FOR COLUMNS column_list;
```


Summary

In this chapter, we have covered Impala administration and performance improvement using various methods including Cloudera Manager. We discussed Impala High Availability, which mainly depends on Hadoop NameNode High Availability. We studied methods such as enabling block location tracking, native checksumming, and short-circuit read, that help us read data quickly in the Hadoop cluster to improve Impala performance. We also discussed how various types of file and compression formats help us to improve performance and, if not chosen wisely, the file format or compression could drag down the data processing performance. We also discussed gaining higher query execution performance by modifying the query in such a way that its processing is expedited. As most of these topics require a great deal of background information, having them here in this book as a reference will definitely help you to understand them and use them to improve your Impala cluster performance.

The next chapter is all about troubleshooting Impala when experiencing problems. We will extend our knowledge by learning how to find the root cause of various problems in the Impala cluster and resolve them quickly.

6

Troubleshooting Impala

In the first part of this chapter, we are going to learn how to troubleshoot various Impala issues in different categories. We will use Impala logging to understand more about Impala execution, query processing, and possible issues. The objective of this chapter is to provide you some critical information about Impala troubleshooting and log analysis, so that you can manage the Impala cluster effectively and make it useful for your team and yourself. Let's start with troubleshooting various problems while managing the Impala cluster.

Troubleshooting various problems

Impala runs on DataNodes in a distributed clustered environment. So when we consider the potential issues with Impala, we also need to think about the problems within the platform itself that can impact Impala. In this section, we will cover most of these issues along with query, connectivity, and HDFS-specific issues.

Impala configuration-related issues

If you find that Impala is not performing as expected, and you want to make sure it is configured correctly, it is best to check the Impala configuration. With Impala installed using Cloudera Manager, you can use the **Impala** debug web server at port 25000 to check the Impala configuration. Here is a small list describing what you could see in the **Impala** debug web server:

- **Impala Configuration Variables List:** `http://impala_server_name:25000/varz`
- **Impala Memory consumption details:** `http://impala_server_name:25000/memz`
- **Impala cluster statistics:** `http://impala_server_name:25000/metrics`
- **All databases and tables:** `http://impala_server_name:25000/catalog`

The block locality issue

In *Chapter 5, Impala Administration and Performance Improvements*, we have learned that enabling "block locality" helps Impala to process queries faster. However, it is possible that "block locality" is not configured properly and you might not be taking advantage of such functionality. You can make sure by checking the logs to verify if you see the following log message:

```
Unknown disk id. This will negatively affect performance. Check your
hdfs settings to enable block location metadata
```

If you see the preceding log message, it means that tracking block locality is not enabled. Therefore, configure it correctly as described in *Chapter 5, Impala Administration and Performance Improvements*.

Native checksumming issues

We have also studied in *Chapter 5, Impala Administration and Performance Improvements*, that having native checksumming improves performance. If you see the following log message, it means native checksumming is not enabled and you need to configure it correctly. This is described in *Chapter 5, Impala Administration and Performance Improvements*.

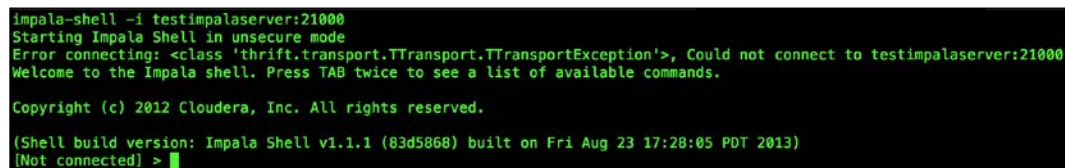
```
Unable to load native-hadoop library for your platform... using
builtin-java classes where applicable
```

Various connectivity issues

In this section, we will cover various connectivity scenarios and learn what could go wrong in each and how to troubleshoot them.

Connectivity between Impala shell and Impala daemon

When you start Impala shell by passing the hostname using the `-i` option or the Impala shell, try connecting to the default Impala daemon that is running on the local machine. The connection can not be established. You will see a connection error as shown in following screenshot:



```
impala-shell -i testimpalaserver:21000
Starting Impala Shell in unsecure mode
Error connecting: <class 'thrift.transport.TTransport.TTransportException'>, Could not connect to testimpalaserver:21000
Welcome to the Impala shell. Press TAB twice to see a list of available commands.

Copyright (c) 2012 Cloudera, Inc. All rights reserved.

(Shell build version: Impala Shell v1.1.1 (83d5868) built on Fri Aug 23 17:28:05 PDT 2013)
[Not connected] > █
```

To troubleshoot the preceding connection problem, you can try the following options:

- Check if the hostname is correct and a connection between both machines is working. You can use `ping` or another similar utility to check the connectivity between machines.
- Make sure that the machine where the Impala shell is running can resolve the Impala hostname, and the default port 21000 (or the other configured port) is open for connectivity.
- Make sure that the firewall configuration is not blocking the connection.
- Check whether the respective process is running on the Impala daemon host machine. You can use Cloudera Manager or the `ps` command to get more information about the Impala process.

ODBC/JDBC-specific connectivity issues

Impala provides connection through the third-party application that uses the ODBC/JDBC driver running on the machine, which is trying to connect to the Impala server. The connection may not work due to various reasons, which are given as follows:

- ODBC use the default 21000 port and JDBC uses the 21050 port in Impala to provide the connectivity; make sure that the incoming to these ports are working. Cloudera ODBC connection 2.0 and 2.5 uses the 21050 port when connecting to Impala.
- If your Impala cluster environment is secured through Kerberos or another security mechanism, use appropriate settings in the ODBC/JDBC configuration. In some cases, you may need to contact the application vendor to receive information about the problem.
- After the connection, you may find that some of the functions are not working over the ODBC/JDBC connection. It is very much possible that not all functions are supported, so you may be trying to use an ODBC/JDBC function that is not supported.
- The JDBC connectivity requires a specific Java Runtime depending on the JDBC version. Because of the Java Runtime compatibility requirement with JDBC, you must make sure that you do have a compatible Java Runtime on a machine that is making the JDBC connection to the Impala server.

Query-specific issues

The very first query-specific issue is a bad query. The Impala query interpreter is smart in various ways to guide you within the Impala shell for a bad query, or while using API to execute the query statement a detailed error in the log file about it will help you. Besides a bad query, you may also experience the following issues:

- You might use an unsupported statement or clause in your query, which will cause a problem in query execution.
- Using an unsupported data type or a bad data transformation is another prime reason for such issues and the resulting error or log will be helpful to troubleshoot what went wrong.
- Sometimes the query is localized. This means that it is not distributed on other nodes. The problem could be that either the current node could not connect to the other nodes due to connectivity issues, or the Impala daemon is not running there. You will have to troubleshoot this issue by using general connectivity troubleshooting methods between two machines. Also, make sure Impala daemons are running with proper configuration.
- Queries could return wrong or limited results. This is possible if metadata is not refreshed in the Impala cluster. Using the `REFRESH` statement, you can sync Hive metadata to solve this problem. Also, make sure that Impala daemons are running on all the nodes.
- If you find that the `JOIN` operations are failing, it is very much possible that you are hitting the memory limitation. While checking Impala logs, you might look for `Out of Memory` errors logged to confirm memory limitation-specific errors. As the `JOIN` operation is performed among multiple tables, which requires comparatively large memory to process the `JOIN` request, so adding more memory could solve this problem.
- Your query performance could be slow. In the previous chapter, we discussed various ways to find the trouble and then expedite the query performance.
- Sometimes, when a query fails in Impala and you could not find a reason, try running the same query in Hive to see if it works there or not. If it works in Hive, it could be an Impala-specific configuration or a limitation issue.

Issues specific to User Access Control (UAC)

During the Impala installation process, the Impala username and group is created. Impala runs under this username and accesses system resources within this group. If you delete this user or group, or modify its access, either Impala will start acting weird or it will show some undeterministic behavior. If you start Impala under the root user, it will also impact the Impala execution by disabling direct reading. So if you suddenly experience such issues, please check Impala user access settings and make sure that Impala is running as configured.

Platform-specific issues

In this section, I will explain a few platform-specific issues so the an event an Impala execution is sporadic or not working at all, you can troubleshoot the problem and find the appropriate resolution.

Impala port mapping issues

Impala has two main services, Impala daemon and statestore, and both these services are configured to use internal and external ports for effective communication. This is described in the following table:

Component	Port	Type	Service description
Impalad	21000	External	Frontend port to communicate with the Impala shell
Impalad	21050	External	Frontend port for ODBC 2.0
Impalad	22000	Internal	Backend port to communicate with each other
Impalad	23000	Internal	Backend port to get update from the statestore
Impalad	25000	External	Impala web interface for monitoring and troubleshooting
Statestored	24000	Internal	Statestore listen for registration/unregistration
Statestored	25010	External	Statestore web interface for monitoring and troubleshooting



Cloudera ODBC Connector 1.x uses Impala port 21000 to create connections; however, the latest Cloudera ODBC Connector 2.0 and 2.5 connects to Impala on the 21050 port.

It is important to remember that if any of the preceding port configuration is wrong or blocked, you will experience various problems and would need to make sure that the preceding port configuration is correct.

HDFS-specific problems

Impala runs on the DataNode that has dependency on NameNode in the Hadoop environment. Various HDFS-specific issues such as permission to read or write data on HDFS, space limitation, memory swapping, or latency could impact the Impala execution. Any of these issues could introduce instability in HDFS or impact the whole cluster, depending on how serious the problem is. In this situation, you would need to work with your Hadoop administrator to resolve these problems and get Impala up and running.

Input file format-specific issues

Impala can load and query various kinds of datafiles stored on Hadoop. Sometimes you may receive an error while reading these datafiles or failed query requests. Most probably it is because either the file format is not supported, or Impala is limited to only queries and cannot process `CREATE` or `INSERT` requests. In the following table, you can see which file formats are supported and whether Impala can read and query those files:

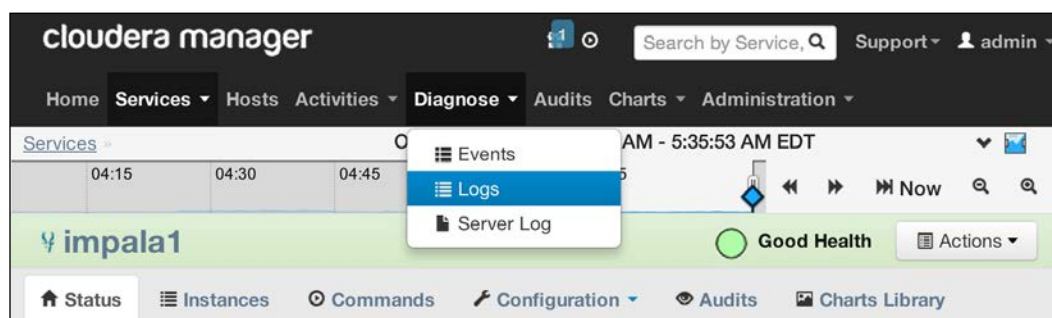
File type	Format type	Compression type	Is CREATE and INSERT supported	Is the query supported?
Text	Unstructured	LZO	Yes	Yes
Avro	Structured	Snappy, GZIP, deflate, BZIP2	No	Query only (use Hive to load file)
RCFile	Structured	Snappy, GZIP, deflate, BZIP2	CREATE: Yes INSERT: No	Query only (use Hive to load file)
SequenceFile	Structured	Snappy, GZIP, deflate, BZIP2	CREATE: Yes INSERT: No	Query only (use Hive to load file)
Parquet	Structured	Snappy, GZIP	Yes	Yes

Using Cloudera Manager to troubleshoot problems

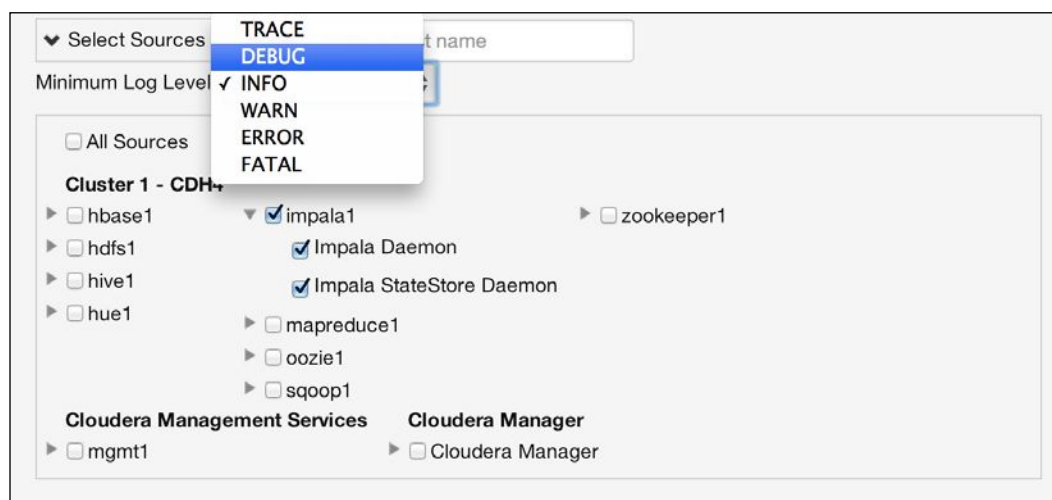
Installing Impala with Cloudera Manager will not only help in installing and upgrading Impala, but it will also be very helpful in Impala management and troubleshooting.

Impala log analysis using Cloudera Manager

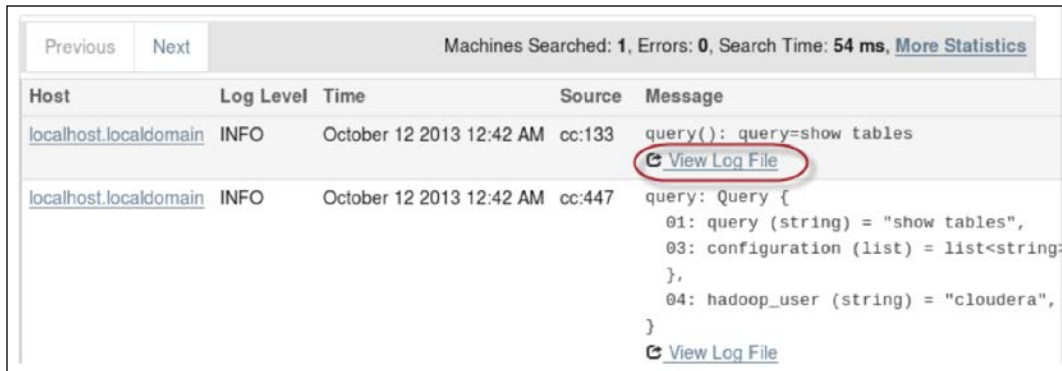
In Cloudera Manager, you can navigate to the **Diagnose** | **Logs** option to select Impala-specific log configurations, as shown in the following screenshot:



In the log configuration page, you can select Impala daemon, statestore, and log types such as INFO, DEBUG, WARN, ERROR, TRACE, or FATAL. This is shown in the following screenshot:



After Impala-specific log configuration is completed, you can see the Impala log files on the same page along with log snippets. You can view the whole log files by just performing one-click operation, as shown in the following screenshot:



Previous	Next	Machines Searched: 1, Errors: 0, Search Time: 54 ms, More Statistics		
Host	Log Level	Time	Source	Message
localhost.localdomain	INFO	October 12 2013 12:42 AM	cc:133	query(): query=show tables View Log File
localhost.localdomain	INFO	October 12 2013 12:42 AM	cc:447	query: Query { 01: query (string) = "show tables", 03: configuration (list) = list<string> }, 04: hadoop_user (string) = "cloudera", } View Log File

Besides this, you can view the Impala log files at console windows by visiting the `/var/log/impala` or `/var/log/impalad` folders. These are the folders where Impala stores logs by default in each node where `impalad` is running. The entire log files are generated at the restart of the `impalad` process with a timestamp. In these directories, the Impala log files are as follows:

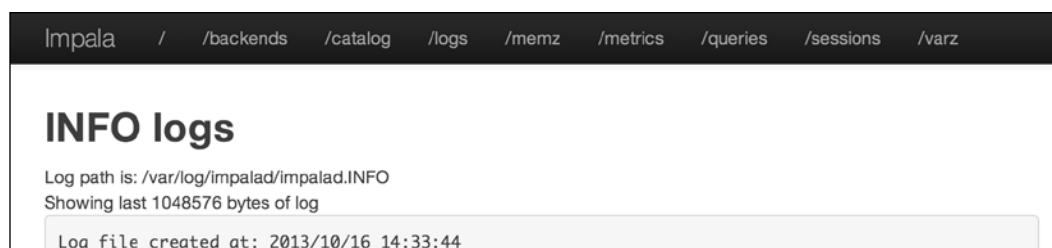
Log file name	The information stored in the log file
<code>impalad.INFO</code>	Impala daemon-specific configuration settings
<code>impalad.WARNING</code>	Information about all kinds of warnings generated by Impala
<code>impalad.ERROR</code>	All kinds of errors and potential problems encountered by Impala
<code>impalad.FATAL</code>	A FATAL error that could stop Impala execution or induce query failure
<code>statestored.ERROR</code>	Statestore-specific errors

Impala logs contain detailed information about the Impala daemon and statestore processes, along with each query processed by Impala. Some of the info, such as process and machine info, is logged once. However, when queries are processed, all query details are logged. This is described in the following table:

One time info in the logs at Impala startup	Each query-specific detail in the Impala log
Machine name	Query composition
Impala version	Degree of data locality
Impala startup flags	Data throughput statistics
CPU and disks information	Query response time

Using the Impala web interface for monitoring and troubleshooting

Impala provides the web interface for the Impala process at the 25000 port, as shown in the following screenshot. This is to check various information, such as configuration, logs, metrics, queries details, and memory:



Using the Impala statestore web interface

Same as the last section, the statestore web interface is available at the 25010 port with the same info as the last section. This is shown in the following screenshot:

The screenshot displays the Impala statestore web interface. At the top, a dark navigation bar contains the text 'Impala' followed by links: '/', '/logs', '/memz', '/metrics', '/subscribers', '/topics', and '/varz'. Below this, the 'Version' section shows 'statestored version 1.1.1 RELEASE (build 83d5868f005966883a918a819a449f636a5b3d5f)' and 'Built on Fri, 23 Aug 2013 17:52:25 PST'. The 'Hardware Info' section provides details on CPU (Intel(R) Xeon(R) CPU E5-2650 0 @ 2.00GHz, 2 cores, 32.00 KB L1 cache, 256.00 KB L2 cache, 20.00 MB L3 cache), hardware supports (ssse3, sse4_1, sse4_2, popcnt), memory (7.29 GB), and disk (1 disk, xvde rotational=true). The 'Status Pages' section lists links: '/', '/logs', '/memz', '/metrics', '/subscribers', '/topics', and '/varz'.

Impala / /logs /memz /metrics /subscribers /topics /varz

Version

statestored version 1.1.1 RELEASE (build 83d5868f005966883a918a819a449f636a5b3d5f)
Built on Fri, 23 Aug 2013 17:52:25 PST

Hardware Info

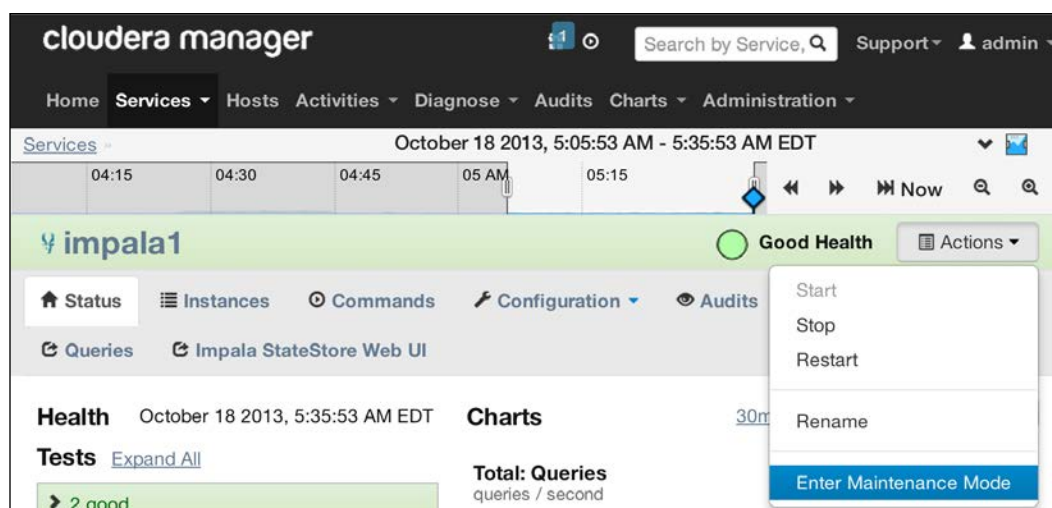
Cpu Info:
Model: Intel(R) Xeon(R) CPU E5-2650 0 @ 2.00GHz
Cores: 2
L1 Cache: 32.00 KB
L2 Cache: 256.00 KB
L3 Cache: 20.00 MB
Hardware Supports:
ssse3
sse4_1
sse4_2
popcnt
Mem Info: 7.29 GB
Disk Info:
Num disks 1:
xvde (rotational=true)

Status Pages

[/](#)
[/logs](#)
[/memz](#)
[/metrics](#)
[/subscribers](#)
[/topics](#)
[/varz](#)

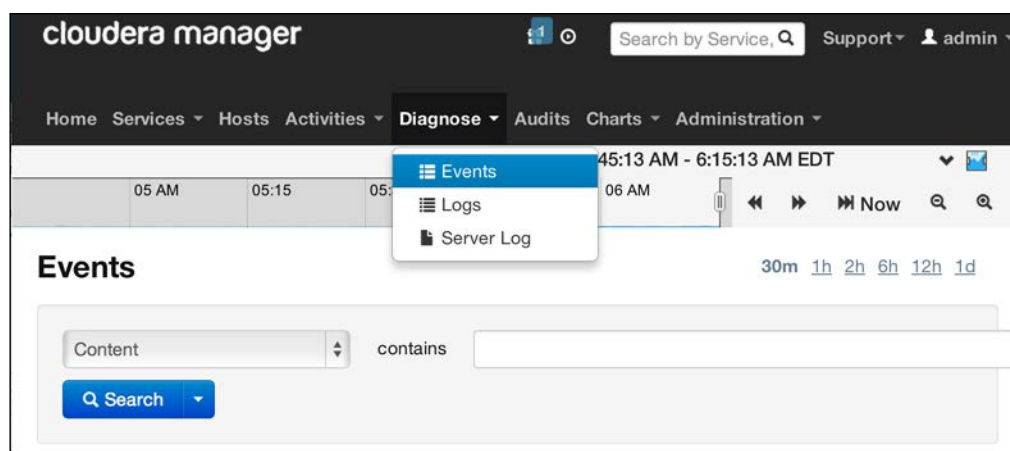
Using the Impala Maintenance Mode

Cloudera Manager provides a configuration named **Maintenance Mode** that can help you monitor the whole Impala cluster effectively. With Cloudera Manager, Maintenance Mode can be set for a service, a role, a host, or a whole cluster. Using Maintenance Mode as an administrator, you can suppress alerts for the host, role, service, or full cluster. While in Maintenance Mode, logging still works as it is. Therefore, even when there are no alerts, all the logs are still saved, so that anyone can check those logs to meet any specific requirement later. When making specific changes to the cluster or any specific component, Maintenance Mode helps to reduce unnecessary change notifications to all users and keeps the noise due to maintenance very low. The following screenshot shows how you can set your Cloudera Hadoop cluster into Maintenance Mode:



Checking Impala events

Using Cloudera Manager, you can track various events that are generated by Impala, events such as activity, error code, file access, user access, exception, command execution, role, and several others listed in the **Diagnose** drop-down list, as shown in the following screenshot:



Summary

In this chapter, we learned various details of Impala troubleshooting through Cloudera Manager: log analysis, checking events, console output, and so on. We have seen how Cloudera Manager can be very useful to troubleshoot various problems in Impala as well as how you can look for potential performance-specific issues in logs. A manual study of the log is very important to learn more about Impala execution and once you understood it very well, you can troubleshoot the problem just by revisiting the log. There are several other factors that can be considered as potential problems that impact Impala performance. Sometimes, the Hadoop cluster itself is very busy performing several MapReduce jobs submitted by other issues. This can consume significant resources from nodes in the Hadoop cluster and ultimately cause problems in Impala execution. Networking issues, such as a congested network, slow performing network cards, and network limitations of any kind could also cause potential performance issues with Impala. In most of these situations, cluster and logs analysis is one of the best options to find the root cause. Then, apply the specific information that you learned in this chapter to solve your problem.

Chapter 7, Advanced Impala Concepts, covers various advanced concepts that will extend Impala to the next level and make it much more useful. In the next chapter, we will cover a few advanced topics such as HBase integration and HDFS file formats to increase your knowledge of Impala.

7

Advanced Impala Concepts

In *Chapter 6, Troubleshooting Impala*, we discussed various concepts about Impala, which have definitely given you enough information to let you take charge of Impala projects and successfully manage them. In this chapter, we are going to learn more about Impala; however, this information is more advanced in nature, to help you excel in data-processing projects using Impala. I describe how Impala works side by side with MapReduce without using it in the same cluster. I also explain why Impala has an edge over Hive even though Hive is a key component on which Impala is dependent. Finally, we will cover some details on using HBase with Impala and processing various Big Data input file formats on Hadoop with Impala.

Impala and MapReduce

The very first thing to note is that Impala does not replace MapReduce or use MapReduce as a processing engine. Impala processes data much, much faster than MapReduce and is considered an alternative data-processing framework on Hadoop. Impala processes data stored at the Hadoop data storage layer using its open source in-memory processing framework, which does not have an overhead as MapReduce does. Impala bypasses MapReduce to have native access to data in HDFS using the distributed query engine designed specially for superfast data processing. As each Impala daemon processes data locally on DataNode, processing is fast due to little or no network latency. You must know the fact that MapReduce is an amazing distributed data-processing framework to process data directly in a distributed clustered environment on DataNodes; however, executing SQL statements through the MapReduce framework exhibits performance inefficiencies mainly due to disk access. Impala overcomes this inefficiency by processing data in memory. Impala runs side by side with MapReduce by using the same Hadoop core components and hardware infrastructure. As mentioned earlier and rephrased here again, Impala is faster because the data is processed in memory; therefore, the memory requirement for Impala-installed Hadoop clusters is comparatively higher.

Impala and Hive

In this book, we have always emphasized that Impala uses the Hive metastore as a catalog only. While Hive uses MapReduce to process its queries, MapReduce takes charge of distributing the queries and then returning results back to Hive. Impala uses its own daemons running on one or many or all DataNodes and performs query process tasks. There are a few key topics where Impala and Hive are very different, and I have noted some of them in the following section.

Key differences between Impala and Hive

- Impala performs **in-memory** query processing while Hive does not
- Hive use **MapReduce** to process queries, while Impala uses its own processing engine
- Hive can be extended using **User Defined Functions (UDF)** or writing a custom **Serializer/Deserializer (SerDes)**; however, Impala does not support extensibility as Hive does for now
- Impala depends on Hive to function, while Hive does not depend on any other application and just needs the core Hadoop platform (HDFS and MapReduce)
- Impala queries are subsets of HiveQL, which means that almost every Impala query (with a few limitation) can run in Hive. But vice-versa is not true because some of the HiveQL features supported in Hive are not supported in Impala

Impala and Extract, Transform, Load (ETL)

Impala provides a complete Big Data solution, which does not require **Extract, Transform, Load (ETL)**. In ETL, you extract and transform the data from the original data store and then load it to another data store, also known as the **data warehouse**. In this model, the business users interact with the data stored at the data warehouse. Mostly, data stored in the data warehouse is partial data compared to the primary data source. Also, users need to perform ETL steps again and again for getting updated data and this step could take time, causing business users significant delay. The following are a few key differentiators that prove Impala's advantage over ETL:

- Impala provides full access to primary data to its users without using a middleman or mid-level processing.

- Impala supports end-to-end data processing and analytics solutions on Hadoop, which helps its users avoid modeling or ETL.
- With Impala, users have direct and full access to data in Hadoop. Impala users do not require any ETL strategy to work on data. Users can take full control of data to process it end-to-end and the results from Impala can be consumed by other application, if needed.
- Impala supports various input file formats that are popular in Big Data, so using a single system for data processing such as Impala negates the need for the user to use ETL for data transformation.

Why Impala is faster than Hive in query processing

We have mentioned many times in this book that Impala is a very fast distributed data-processing framework, so you might want to know how Impala achieves such speed or what is behind Impala that makes it so fast. I would answer this question by providing the following key points:

- While processing SQL-like queries, Impala does not write intermediate results on disk; instead full SQL processing is done in memory, which makes it faster.
- With Impala, the query starts its execution instantly compared to MapReduce, which may take significant time to start processing larger SQL queries and this adds more time in processing.
- **Impala Query Planner** uses smart algorithms to execute queries in multiple stages in parallel nodes to provide results faster, avoiding sorting and shuffle steps, which may be unnecessary in most of the cases.
- Impala has information about each data block in HDFS, so when processing the query, it takes advantage of this knowledge to distribute queries more evenly in all DataNodes.
- Another key reason for fast performance is that Impala first generates assembly-level code for each query. The assembly code executes faster than any other code framework because while Impala queries are running natively in memory, having a framework will add additional delay in the execution due to the framework overhead.

Impala processing strategy

Now let's review how Impala starts processing a query when it is submitted through any of the following ways:

- When a query is submitted, Impala needs two kinds of metadata to start query processing:
 - Catalog information using Hive metadata
 - File metadata using NameNode
- It is strongly recommended to have the Impala daemon running on all DataNodes, which helps Impala run distributed queries directly on the stored data; however, if the Impala daemon is not running on all DataNodes, it still plans to run the query as effectively and as fast as it can.
- At the time of writing this book, Impala only supports in-memory hash aggregations.
- In the case of the `JOIN` operation, all of the tables referenced in the `JOIN` operation must fit in the aggregate memory on the host or hosts where Impala is running.
- If the `JOIN` operation is submitted, Impala will use either broadcast or partitioned join, depending on the query planner's decision, and follow the table order provided in the `SELECT` statement.
- Impala processes all queries in memory, so memory limitation on nodes is definitely a factor. You must have enough memory to support the resultant dataset, which could grow multifold during complex `JOIN` operations.
- If a query starts processing the data and the resultant dataset cannot fit in the available memory, the query will fail.

Impala and HBase

HBase is a very popular nonrelational database on Hadoop that stores data in a column-oriented store model. HBase also uses HDFS as its data storage layer and MapReduce to process data. The key difference between Hive and HBase is that HBase is a complete nonrelational database running on Hadoop, while Hive is a SQL-like database that supports SQL statements to process data. As it is another kind of database, HBase supports the concepts of databases, tables, and columns and uses SQL statements to submit queries while processing the data in tables on HDFS.

Impala does not disappoint us and provides great flexibility to query data in HBase tables. Impala tables process datafiles stored on HDFS—great for bulk loads and full-table-scan queries; however, HBase can perform efficient data processing by performing individual row or range lookups. Impala considers HBase a key-value store in which the key is mapped to one column in the Impala table and value fields are mapped to other columns.



While discussing HBase, internals are out of the scope of this book. If you are working on the HBase table with Impala, I would suggest reading the appropriate HBase documentation or visiting the Apache HBase website for the latest documentation, <http://hbase.apache.org/>.

Here are the steps to work with HBase and Impala together:

1. Use the Hive shell to create a Hive table using `CREATE EXTERNAL TABLE` and specific keywords and map Hive tables with HBase tables. We are using the Hive shell only because certain keywords used in SQL statements are not supported in Impala.
2. Define the column corresponding to the HBase row key as a string with the `#string` keyword or map it to the `STRING` column.
3. Once the preceding steps are done, the Hive metastore will be updated with the required information and Impala can perform queries on these tables.
4. Make sure Impala users have read/write access for HBase tables. Using the `GRANT` command in HBase shell can do this.

Using Impala to query HBase tables

While querying HBase tables, Impala uses the HBase client API to query data stored in HBase. You can create external tables in Hive with or without the string key. Here is an example of creating a table first in HBase and then in Hive for mapping, and finally, querying it in Impala:

1. Create the HBase table in the HBase shell as follows:

```
Create 'hbasetable', 'ints', 'strings'
Enable 'hbasetable'
```

2. Create an external table in the Hive shell with a string row key as follows:

```
CREATE EXTERNAL TABLE hivetableforhbase_userid (
  UserId string,      /* Row Key is set as String */
  UserName string,    UserAge int,
  UserDob timestamp)
```

```
STORED BY
'org.apache.hadoop.hive.hbase.HBaseStorageHandler'
WITH SERDEPROPERTIES (
    "hbase.columns.mapping" =
    ":key,strings:UserID,strings:UserName,ints:UserAge,
    strings:UserBob )
TBLPROPERTIES("hbase.table.name" =
    "hivetableforhbaseuseragg");
```

3. You can also create another table without a string row key for learning purposes as follows:

```
CREATE EXTERNAL TABLE hivetableforhbase (
    UserId int, /* Row Key is not set as String */
    UserName string, UserAge int,
    UserDob timestamp)
STORED BY
'org.apache.hadoop.hive.hbase.HBaseStorageHandler'
WITH SERDEPROPERTIES (
    "hbase.columns.mapping" =
    ":key,strings:UserID, strings:UserName,
    ints:UserAge, strings:UserBob )
TBLPROPERTIES("hbase.table.name" = "hivetableforhbase");
```

4. Now we can issue the following query in the Impala shell:

```
-- When row key is mapped as string column, range predicates are
applied in the scan
SELECT * FROM hivetableforhbase_useragg WHERE UserId = '10';
-- When row key is not transformed into scan parameter (not mapped
as string)
SELECT * FROM hivetableforhbase WHERE id = 10;
```

File formats and compression types supported in Impala


Hadoop is used as a data storage system where all kinds of data is stored in various file formats. To reduce the disk space requirement, the data is stored in a compressed format so various compression types are used with different kinds of file formats. Various file formats and compression types create a collection of file formats and compression combinations for any application to support.

Impala does a great job in supporting most of the popular file formats and compression types, as listed in the following table:

File type	Compression type	CREATE support	INSERT support
Text	LZO	Yes	Yes
Avro	GZIP, BZIP2, deflate, Snappy	No (use Hive)	No
RCFile	GZIP, BZIP2, deflate, Snappy	Yes	No
SequenceFile	GZIP, BZIP2, deflate, Snappy	Yes	No
Parquet	GZIP, Snappy (default)	Yes	Yes

The preceding table also describes if Impala can use the `CREATE` or `INSERT` command with specific file and compression types. For example, with the Parquet file type and the Snappy or GZIP compression type, Impala can create tables as well as insert data into those types of tables. Similarly, with the Avro file type, Impala heavily depends on Hive to provide additional support to process such data formats.

If you have a file format that you know is not supported by Impala, you can first create the table with data in Hive, and then issue `INVALIDATE METADATA` in the Impala shell or through an API. After that, you can query the same Hive table in Impala.


 With the arrival of Impala 1.2, support for various file and compression formats could change, so it is recommended that you please visit Cloudera Impala documentation for additional file type and compression support information.

Processing different file and compression types in Impala

Impala loads files stored in HDFS and these files could be of various types. Some of these files are stored in HDFS directly from their source, or some of the files could be the output of MapReduce or Pig or any other application running on Hadoop.

Impala is limited in terms of supporting various file types on Hadoop; however, it does cover most popular Big Data file formats, which gives Impala a very wide range to cover user input requests. If Impala cannot read an input file type, you can perform the following steps to use a combination of Hive and Impala:

1. Use the `CREATE TABLE` statement in the Hive shell to create the table with input data.
2. Use the Impala shell with the `INVALIDATE METADATA` statement so that it does not generate unsupported file type errors.
3. Now write query statements in the Impala shell to achieve your objective.

A very important point to note here is that Impala performance mostly depends on the input file format and the compression algorithm used to compress input files. Compression is used for two main reasons. First, it requires less disk space to store files and small file reads require less disk I/O and CPU resources to load files in memory. Once the file is loaded in memory, it is decompressed in memory only when the data in the file is required for processing. The following table shows the list of Impala-supported compression types and their usage patterns and properties:

Compression type	Why use it?
Snappy	Very fast; it is the fastest in compression and decompression
GZIP	It is the best option to save disk space
LZO	Use only with text files
BZIP2	Not a top choice but Impala can read input files
Deflate	Not a first or second choice; however, can read input files

The following are a few considerations to keep in mind when choosing an appropriate file format for a table with Impala:

- When `CREATE TABLE` is used with Impala, text files are the default input format. It is easier to read for humans and helps troubleshooting problems; however, it does not provide superfast processing with large amounts of data due to significant disk read activity.
- When performance is your primary consideration, use Snappy, and when disk space saving is your primary consideration, use GZIP. LZO can also be used with text files as an option to expedite things a little.
- If your source files are already in one of Impala's supported type, create a table in Impala using the same file format in most of the cases unless changing the format in the Impala table gives you significant improvement in processing the source data in your file.

- If you want to change the file format sometime in Impala, first use `CREATE TABLE` to create a table with your desired file type format and then use the `INSERT` statement to copy data into the Impala table, which requires a one-time file conversion from source to Impala.
- Data compression does not always mean that you will achieve faster processing time by saving important time in disk I/O. Data compression does require CPU cycles to uncompress before processing so it does add up time somewhere. Sometimes having uncompressed data provides significant speed in processing, that the cost to keep it uncompressed in disk compensates the logic to store uncompressed in disk.
- When using uncompressed text files with Impala, you can just copy them onto HDFS first. After that, use `CREATE TABLE` and then use the `INSERT` statement to copy them into Impala.

Now let's take a look at some of the SQL statements that you can use with various input file types in Impala.

The regular text file format with Impala tables

By default, Impala uses the text file format with the `CREATE TABLE` syntax. When data is inserted into this table using `INSERT`, `Ctrl+A` (Hex 01) is used as a default delimiter. The default syntax is as follows:

```
CREATE TABLE users (userID int, username string);
```

To change the delimiter to, for example, `,`, `\t`, `|`, or `your_choice`, you can use the following syntax:

```
CREATE TABLE users (userID int, username string) STORED AS
textfile FIELDS TERMINATED BY '\t';
```



Please visit the Cloudera Impala documentation for text file format support at the following URL:

http://www.cloudera.com/content/cloudera-content/cloudera-docs/Impala/latest/Installing-and-Using-Impala/ciiu_txtfile.html

The Avro file format with Impala tables

With the Avro file format, you would have to create tables in Hive first, as shown in the following code snippet:

```
CREATE TABLE my_avro_table (userID int, userName 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'
  TBLPROPERTIES (
    'avro.schema.literal'='{
      "type": "record",
      "name": "user_record",
      "fields": [
        {"name": "userID", "type": "int"},
        {"name": "userName", "type": "string"}
      ]
    }');
INSERT OVERWRITE TABLE my_avro_table SELECT *, "avro"
  FROM functional.alltypes;
```

Once the file is created in Hive, you can just use it in Impala as any other file as follows:

```
SELECT * from my_avro_table;
```



Please visit the Cloudera Impala documentation for Avro file format support at the following URL:

http://www.cloudera.com/content/cloudera-content/cloudera-docs/Impala/latest/Installing-and-Using-Impala/ciiu_avro.html

The RCFile file format with Impala tables

When you create a table with the RCFile format, without using any existing data use with the table, the syntax is as follows:

```
CREATE TABLE my_rcfile_table (userID int, userName string)
  STORED AS RCFile;
```

Impala can query RCFile-type tables but cannot write to them, so you would need to use Hive to write data into the file using the `INSERT` statement. With Hive, you don't need to specify the storage file type as Hive takes care of it by default.



Please visit the Cloudera Impala documentation for RCFile file support at the following URL:

http://www.cloudera.com/content/cloudera-content/cloudera-docs/Impala/latest/Installing-and-Using-Impala/ciiu_rcfile.html

The SequenceFile file format with Impala tables

Like RCFile, Impala supports the creation of tables that can store SequenceFile data. To create an empty table to store SequenceFile-type data in Impala, you just need to use the following syntax in the Impala shell:

```
CREATE TABLE my_sequencefile_table (userID int, userName string)
STORED AS SEQUENCEFILE;
```

The rest of the steps require you to use Hive for setting up file compression and then writing data into the table using the appropriate `INSERT` statement.



Please visit the Cloudera Impala documentation for SequenceFile file format support at the following URL:

http://www.cloudera.com/content/cloudera-content/cloudera-docs/Impala/latest/Installing-and-Using-Impala/ciiu_seqfile.html

The Parquet file format with Impala tables

You might be wondering what the Parquet file format is. I would like to provide a little information in this context. The Parquet file format is a column-oriented binary file format that is designed to provide column-specific access to the data. As the data is stored in columns and all columns are stored separately, lookups are happening on columns first. This column-oriented access method makes query processing very fast and efficient, and Impala takes advantage of this file format. Impala provides native support to create, manage, and query tables based on the Parquet file format.

The following is the syntax for creating a table that can store the Parquet file format in Impala:

```
CREATE TABLE my_parquet_table (userID int, userName string)
    STORED AS PARQUETFILE;
```

As Impala supports writing the Parquet file format within Impala, you can use the INSERT statement as shown in the following code snippet to write to your Parquet file type from other files:

```
INSERT OVERWRITE TABLE my_parquet_table
    SELECT * FROM other_table_name;
```



Please visit the Cloudera Impala documentation for Parquet file format support at the following URL:

http://www.cloudera.com/content/cloudera-content/cloudera-docs/Impala/latest/Installing-and-Using-Impala/ciiu_parquet.html

The unsupported features in Impala

Let's take a look at what is not supported in Impala so you can make informed decisions when choosing Impala as your distributed data-processing framework on Hadoop:

- Only HDFS is supported for data storage with Impala, and any other data storage framework or RDBMS is not currently supported.
- Impala does not support dropping or deleting a row in a table. The alternative is to either drop the table or migrate the required data to other tables and then delete the entire original table.
- Transforms and window functions are not supported.
- Performing queries on streaming data is not supported.
- Hive UDF and Hive Index are not supported up to Impala 1.1.x; however, at the time of writing this book, Impala 1.2 Beta was available, which has support for Scalar UDF and **user-defined aggregate (UDA)** functions.
- During query processing, unencrypted data is sometimes transmitted between Impala daemons.
- At the time of writing this book, Hadoop 2.0 achieved the GA milestone; however, Hadoop-2.0-based YARN is not integrated with Impala.
- Custom Hive SerDes classes are not supported and only native file formats are supported using the built-in SerDes.



Please visit this URL to learn about the new feature set available in Impala 1.2.x: http://www.cloudera.com/content/cloudera-content/cloudera-docs/Impala/latest/Cloudera-Impala-Release-Notes/cirn_new_features.html?scroll=new_features_121_unique_1.

Impala resources

Let me point you to some very important information about Impala resources that you can get from the following sources:

- **Impala Source:** <https://github.com/cloudera/impala>
- **Impala Download:** <https://www.cloudera.com/content/support/en/downloads.html>
- **Impala v1.x Latest Documentation:** <http://www.cloudera.com/content/support/en/documentation/cloudera-impala/cloudera-impala-documentation-v1-latest.html>
- **Known Issues in Impala:** http://www.cloudera.com/content/cloudera-content/cloudera-docs/Impala/latest/Cloudera-Impala-Release-Notes/cirn_known_issues.html?scroll=known_issues

Summary

In this chapter, we learned a few advanced concepts for Impala, such as how Impala relates to MapReduce, Hive, and other frameworks in the Hadoop ecosystem. We also discussed what makes Impala so fast in processing data and as an Impala user what you can do to improve processing. We discussed the supported file formats in Hadoop to combine with Impala depending on the input data type. By now, you will be prepared to take advantage of in-memory data processing with Impala to process your various types of source data, stored in Hadoop.

Technology Behind Impala and Integration with Third-party Applications

In the last seven chapters, I described the various traits of Impala, and I believe that you have learned those details as well. Now it is time to finish the book by adding a few more details, which will help you understand the true potential of Impala.

Technology behind Impala

The technology behind Impala is revolutionary and inspired by a Google research project named **Dremel**. Dremel is a scalable ad hoc query-based analysis system for read-only nested data. Dremel-based implementations can run aggregation queries over trillions of rows in seconds by combining multilevel executing trees and columnar data layout. It does not use MapReduce as the core; instead it complements MapReduce. Impala is considered to be a native Massive Parallel Processing query engine running on Apache Hadoop. Depending on the type of query and configuration, Impala excels in data processing performance over traditional database applications on Hadoop, such as Hive, and processing frameworks, such as MapReduce, due to the following key reasons:

- Distributed, scalable aggregation algorithms.
- Specialized hardware configuration, such as reducing CPU load, which increases aggregate I/O bandwidth.
- Using the columnar binary storage format on Hadoop, which adds speed to query processing. This is done by taking advantage of Parquet file types as an input source.

- Impala extends its reach beyond Dremel and provides support for various other popular file formats, making its availability and reach beyond Parquet to multifold users.
- Impala uses the available memory on a machine as a table cache, which mean queries always process the data that is available in the cache, making processing super fast by speeding their execution up to 90 times faster than conventional processing when data is read from a disk.

You can learn more on Google Dremel by referring to a research paper at the following URL:

<http://research.google.com/pubs/pub36632.html>

Data visualization using Impala

Visualizing data is as important as processing it. The human brain perceives pictures faster than reading data in tables, and because of this, data visualization provides super fast understanding of large amount of data in split seconds. Reports, charts, interactive dashboards, and any form of infographics are all part of data visualization and provide deeper understanding of results.

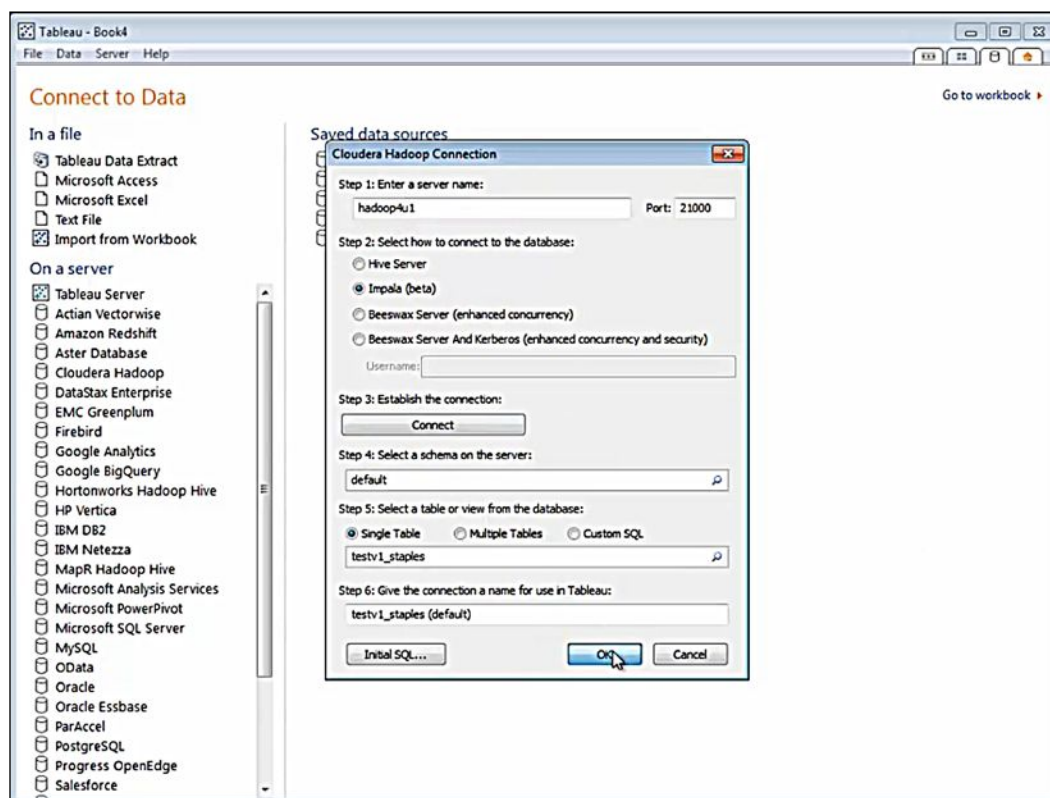
To connect with third-party applications, Cloudera provides ODBC and JDBC connectors. These connectors are installed on machines where third-party applications are running, and by configuring the correct Impala server and port details on those connectors, third-party applications connect with Impala, submit those queries, and then take results back to the application. The result is then displayed on third-party applications, where it is rendered on a graphics device for visualization, displayed in a table format, or further processed depending on the application requirement. In this section, we will cover a few notable third-party applications, which can take advantage of Impala's super fast query processing and then display amazing graphical results.

Tableau and Impala

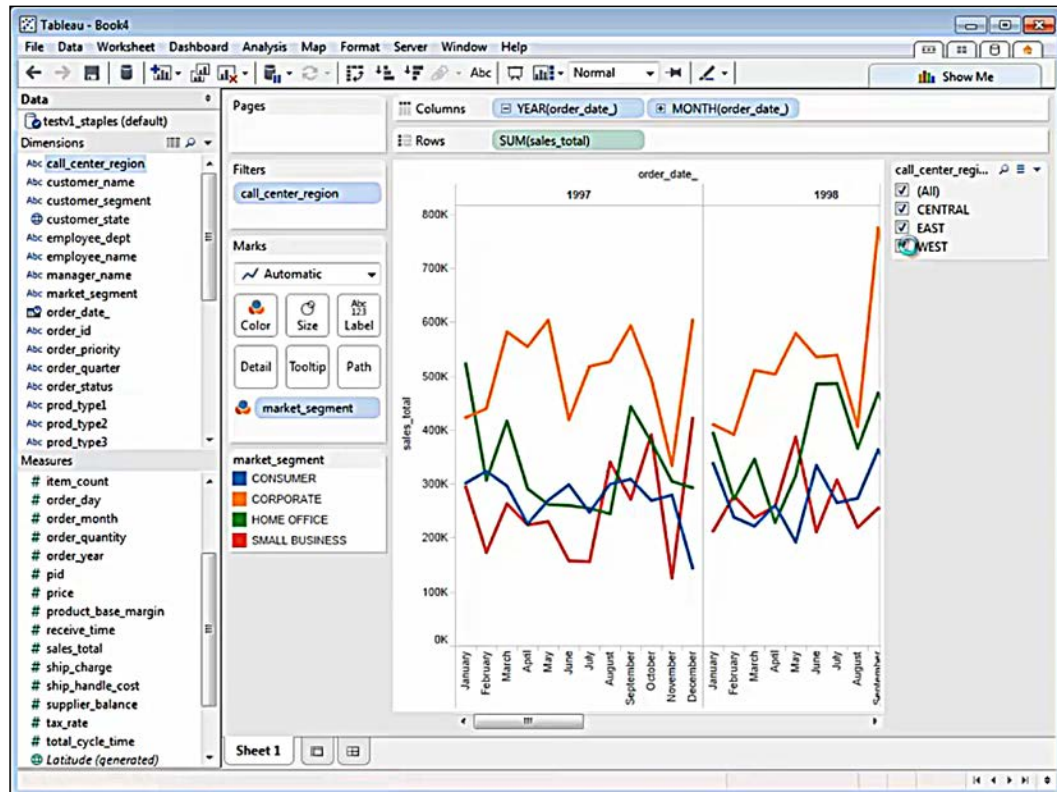
Tableau Software supports Impala by providing access to tables on Impala using the Impala ODBC connector provided by Tableau. Tableau is one of the most prominent data visualization software technologies in recent days and is used by thousands of enterprises daily to get intelligence out of their data. Tableau is available for Windows OS, and an ODBC connector is provided by Cloudera to make this connection a reality. You can visit the following link to download the Impala connector for Tableau:

http://go.cloudera.com/tableau_connector_download

Once the Impala connector is installed on a machine where the Tableau software is running and configured correctly, Tableau is ready to work with Impala. In the following screenshot, Tableau is connected to an Impala server at port 21000 and then a table located in Impala is selected:



Once a table is selected, particular fields are selected, and the data is displayed in a graphical format in various mind-blowing visualizations. The following screenshot displays one example showing such a visualization:



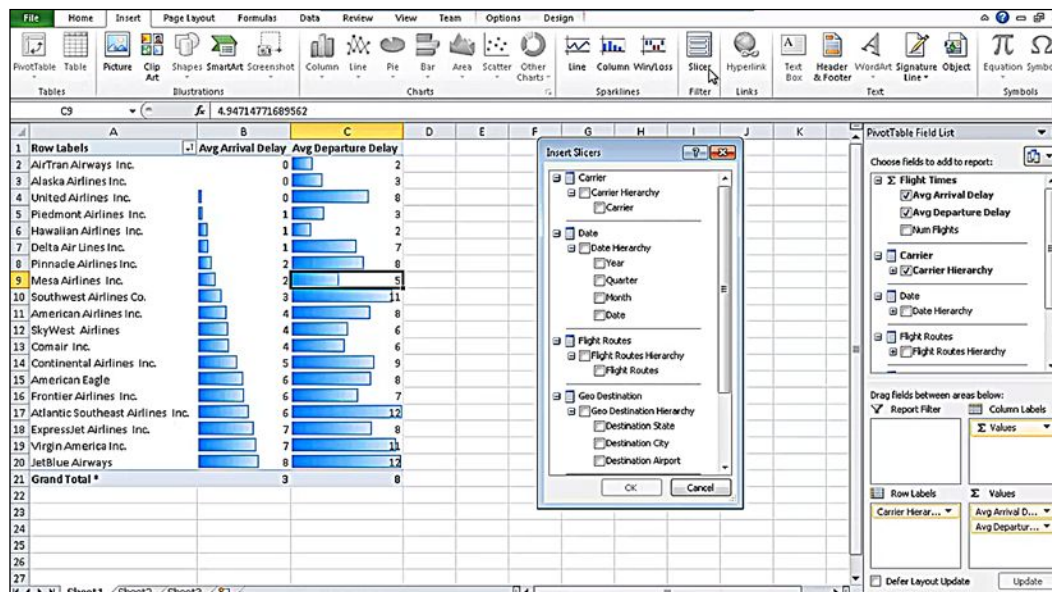
Microsoft Excel and Impala

Microsoft Excel is one of the most widely adopted data processing applications used by business professionals worldwide. You can connect Microsoft Excel with Impala using another ODBC connector provided by Simba Technology. You can download the connector from their website at the following URL:

<http://www.simba.com/data-connections>

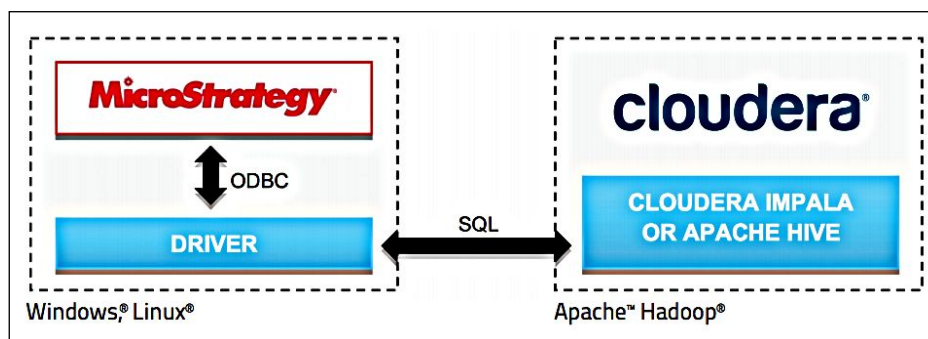
Microsoft OLE DB for OLAP, also known as ODBO, defines multidimensional expressions, or MDX, that are used as a query language to report multi-dimensional data stores. Most of the OLAP servers support interaction through MDX queries by Business Intelligence applications and many other third-party applications. MDX provides flexibility and multidimensional functionality to answer real-world business questions asked by Business Intelligence applications.

Business users can use Simba MDX Provider to connect to Cloudera Impala tables from Microsoft Excel PivotTables, by just installing the driver and configuring it correctly to access Cloudera Impala. In the following screenshot, Microsoft Excel PivotTable is connected to Cloudera Impala using Simba MDX:



Microstrategy and Impala

Microstrategy is another big player in data analysis and visualization software and uses an ODBC drive to connect to Impala to render amazing looking visualizations. The connectivity model between Microstrategy software and Cloudera Impala is shown as follows:





You can use the following URL to learn more about using the Cloudera ODBC connector for Microstrategy:

<http://www.cloudera.com/content/cloudera-content/cloudera-docs/Connectors/Cloudera-Connector-for-MicroStrategy/Cloudera-Connector-for-MicroStrategy.html>

Zoomdata and Impala

Zoomdata is considered to be the new generation of data user interfaces, as it addresses streams of data instead of sets of data. The Zoomdata processing engine performs continuous mathematical operations across data streams in real time to create visualizations on a multitude of devices. The visualization updates itself as new data arrives and is recomputed by Zoomdata.

As shown in the following screenshot, you can see that the Zoomdata application uses Impala as a source of data, which is configured underneath to use one of the available connectors to connect to Impala:

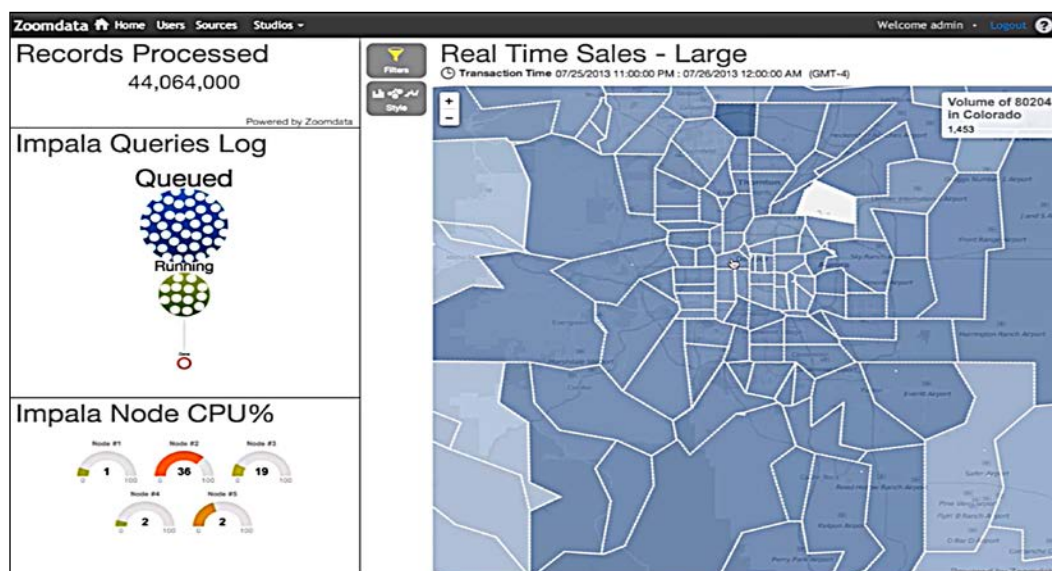
Add Source

CSV Upload API Twitter Impala Salesforce Google SQL DataSift LivePerson Splunk

Manage Sources

Source Name	Fields	Visualizations	Data	Remove Data	Enabled	Delete
4 Billion Rows/day	Manage Fields	Manage Visualizations	CSV	Remove Data	<input checked="" type="checkbox"/>	X
Ad Activity	Manage Fields	Manage Visualizations	CSV	Remove Data	<input checked="" type="checkbox"/>	X
Big Data Twitter	Manage Fields	Manage Visualizations	CSV	Remove Data	<input checked="" type="checkbox"/>	X
ICMA - Donations	Manage Fields	Manage Visualizations	CSV	Remove Data	<input type="checkbox"/>	X
ICMA - Dues	Manage Fields	Manage Visualizations	CSV	Remove Data	<input type="checkbox"/>	X
Impala Node CPU%	Manage Fields	Manage Visualizations	CSV	Remove Data	<input type="checkbox"/>	X
Impala Queries Log	Manage Fields	Manage Visualizations	CSV	Remove Data	<input checked="" type="checkbox"/>	X
Oracle	Manage Fields	Manage Visualizations	CSV	Remove Data	<input type="checkbox"/>	X
Oracle 2	Manage Fields	Manage Visualizations	CSV	Remove Data	<input type="checkbox"/>	X
Ramin Web Test	Manage Fields	Manage Visualizations	CSV	Remove Data	<input type="checkbox"/>	X
Real Time Sales	Manage Fields	Manage Visualizations	CSV	Remove Data	<input checked="" type="checkbox"/>	X

Once the connections are made, the user can see amazing data visualizations, as shown in the following screenshot:



Real-time query with Impala on Hadoop

Impala is marketed as a product that can do real-time queries on Hadoop by its developer, Cloudera. Impala is an open source implementation based on the previously mentioned Google Dremel technology that is available free for anyone to use. Impala is available as a package product that is free to use or can be compiled from its source, which can run queries in memory to make them real time. In some cases, depending on the type of data, if the Parquet file format is used as the input data source, it can expedite the query processing to a multifold speed.

Real-time query subscriptions with Impala

Cloudera provides a **Real-time Query (RTQ)** subscription as an add-on to a Cloudera Enterprise subscription. You can still use Impala as a free, open source product; however, opting for the RTQ subscription allows you to take advantage of the Cloudera paid service to extend its usability and resilience. By accepting the RTQ subscription, you can not only have access to Cloudera Technical support, but you can also work with the Impala development team to provide ample feedback to shape up the product design and implementation.

What is new in Impala 1.2.0 (Beta)

At the time of writing this book, Impala 1.2.0 Beta was available to test with CDH 5.0. Impala 1.2.0 has several features visible to users; however, lots of other features are under the hood to improve performance, security, and flexibility. A few notable features are as follows:

- Impala supports **user-defined functions (UDF)** natively, and users can write scalar UDF and **user-defined aggregate functions (UDA)**.
- Functions written in C++ and Java can work with Impala as they are.
- Currently, `REFRESH` statements are required after every use of table-specific SQL commands, such as `CREATE TABLE`, `ALTER TABLE`, `DROP TABLE`, `INSERT`, and `LOAD DATA`, to update information to the whole cluster. Impala now has an automatic synchronization mechanism, so there is no need for `REFRESH` or `INVALIDATE METADATA` SQL commands. With the automatic synchronization mechanism, a newly created service takes charge of updating table or metadata specific information to the whole Impala cluster as the changes are available.
- Another big update is integration with YARN, in which Impala uses the YARN resource management framework for adequate resource management during query processing.



According to Cloudera, Impala 1.2.0 Beta is packaged with Cloudera CDH 5.0 (Beta) and only works with Cloudera CDH 5.0. Please visit the following URL for more details:

http://www.cloudera.com/content/cloudera-content/cloudera-docs/Impala/1.2.0-beta/Cloudera-Impala-Release-Notes/cirn_new_features.html

Index

Symbols

- B command 33
- c command 32
- ! command 35
- database database_name command 33
- d database_name command 33
- delimited command 33
- d option 31
- f query_file_name command 33
- h command 31
- help command 31
- I hostname command 32
- impalad=hostname command 32
- k command 34
- kerberos command 34
- kerberos_service_name=Kerberos_service_name command 34
- o filename command 33
- output_file filename command 33
- p command 33
- q option 34, 35
- q query command 33
- query=query command 33
- quiet command 32
- r command 32
- refresh_after_connect command 32
- show_profiles command 33
- s Kerberos_service_name command 34
- v command 32
- verbose command 32
- version command 32

A

- ABS(DOUBLE a) function 63
- aggregation functions
 - about 55
 - AVG 55
 - COUNT 56
 - MAX 56, 57
 - MIN 56, 57
- ALL privilege 23
- alter command 38
- ALTER TABLE statement 44, 45
- ANALYZE TABLE statement 91
- Apache Hive 21
- ASCII(String str) function 64
- auditing 24, 25
- authentication
 - through Kerberos 24
- authorization
 - about 23
 - ALL privilege 23
 - INSERT privilege 23
 - SELECT privilege 23
- AVG aggregation function 55
- Avro file format 114
 - URL 114

B

- BETWEEN operator 53
- BIGINT data type 49
- BIN(BIGINT a) function 63
- block locality issue 94

block location tracking
enabling 85

BOOLEAN data type 48

built-in functions 63

C

CASE function 64

CAST() function 50, 65

CAST operator 48

Clause

about 57

FROM clause 57

GROUP BY clause 59

HAVING clause 59

LIMIT clause 59

ORDER BY clause 59

WHERE clause 58

WITH clause 58

Cloudera Manager

administration with 82, 83

Impala events, checking 104

Impala, installing without 13

Impala log analysis 99-101

Impala Maintenance Mode, using 103

Impala statestore web interface, using 102

Impala upgrading, packages used 17

Impala upgrading, parcels used 17

Impala, upgrading with 18

URL 12

used, for installing Impala 11, 12

used, to troubleshoot platform issues 98

using, for Impala 27-29

web interface 101

cluster statistics

URL 93

COALESCE function 64

command-line options

about 30, 31

general 31, 32

query-specific options 33

secure connectivity-specific options 34

command-line options, connection-specific

-d database_name or -database

database_name 33

-I hostname or -impalad=hostname 32

-r or -refresh_after_connect 32

about 32

command-line options, general

-c 32

-h or --help 31

--quiet 32

-V or --verbose 32

-v or --version 32

command-line options, query-specific

-B or --delimited 33

-f query_file_name or -query_file=query_
file_name 33

-o filename or --output_file filename 33

-p or --show_profiles 33

-q query or -query=query 33

**command-line options, secure connectivity-
specific**

-k or --kerberos 34

-s Kerberos_service_name or -kerberos_
service_name=Kerberos_service_name
34

commands

general commands 35

query-specific commands 36, 37

table- and database-specific commands 38

commands, general

! command 35

connect command 35

exit command 35

help command 35

history command 35

quit command 35

refresh command 35

shell command 35

version command 35

commands, query-specific

explain command 37

profile command 37

set command 36

unset command 36

commands, table- and database-specific

about 38

alter command 38

describe command 38

drop command 38

insert command 38

select command 38

use command 38

- compression types**
 - about 110, 111
 - processing 111, 112
- CONCAT(String a, String b.)**
 - function 64
- Configuration-related issues**
 - about 93
 - block locality issue 94
 - native checksumming issues 94
- Configuration Variables List**
 - URL 93
- connect command 35**
- connectivity issues**
 - between Impala shell and Impala daemon 94, 95
 - JDBC-specific connectivity issues 95
 - ODBC-specific connectivity issues 95
- COS(Double a) function 63**
- COUNT aggregation function 56**
- count SQL command 78**
- CREATE DATABASE statement 41**
- CREATE EXTERNAL TABLE**
 - statement 44, 48
- CREATE TABLE command 40**
- CREATE TABLE statement 43**

D

- data**
 - loading, from HDFS 62
 - loading, into Impala table,
 - from HDFS 70, 71
 - loading, into Impala tables 69
 - visualizing, Impala used 120
- database-specific statements**
 - about 41
 - CREATE DATABASE statement 41
 - DROP DATABASE statement 41
 - SHOW DATABASES statement 42
 - using, in example 42
- Data Definition Language (DDL) 39**
- Data Manipulation Language (DML) 40**
- DataNode**
 - short-circuit read, performing 86
- dataset**
 - example 67, 68

- data type**
 - about 48
 - BIGINT 49
 - BOOLEAN 48
 - DOUBLE 50
 - FLOAT 50
 - INT 49
 - SMALLINT 49
 - STRING 51
 - SUM 51
 - TIMESTAMP 52
 - TINYINT 50
- DATEDIFF(date1, date2) function 64**
- describe command 38, 73, 74**
- DESCRIBE statement 45, 46**
- distinct command 76**
- DISTINCT operator 53, 54**
- distinct SQL command 78**
- DOUBLE data type 50**
- Dremel 119**
- drop command 38**
- DROP DATABASE statement 41**
- DROP TABLE statement 45**

E

- example scenario, creating**
 - about 67
 - automobiles (automobiles.txt) 68
 - data and schema, considerations 69
 - motorcycles (motorcycles.txt) 68
- exit command 35**
- EXPLAIN clause 60, 78**
- explain command 37, 79**
- external table 48**
- Extract Transform Load (ETL) 62, 106, 107**

F

- file format**
 - about 111
 - selecting 89
- FLOAT data type 50**
- FLOOR(Double a) function 63**
- FORMAT() function 55**
- FROM clause 57**

functions

- aggregation function 55
- Scalar function 55

G

Google Dremel

- URL 120

GROUP BY clause 59

H

hardware

- dependency 10

HAVING clause 59

HBase

- about 22
- and Impala 108, 109
- URL 22

HBase tables

- querying, Impala used 109, 110

HDFS

- about 22
- data, loading into Impala table 70, 71
- specific commands 69, 70

HDFS-specific problems 98

help command 35

High Availability (HA) 84

Hive

- and Impala 106
- and Impala, differences 106, 107
- dependency, for Impala 10

HiveQL statements 66

I

IF function 64

Impala

- administration 81
- and Extract, Transform, Load (ETL) 106, 107
- and HBase 108, 109
- and Hive 106
- and Hive, differences 106, 107
- and MapReduce 105
- and Microsoft Excel 122
- and Microstrategy 123
- and Tableau 121, 122

and Zoomdata 124

benefits 8, 9

built-in function support 63-65

Cloudera Manager, using 27-29

compression types 110, 111

compression types, processing 111, 112

configuring, after installation 14, 15

core components 18

dependency on Hive 10

dependency on Java 10

example, scenario 67

execution architecture 21

file formats 110, 111

file format, selecting 89

file formats, processing 111, 112

hardware dependency 10

High Availability (HA) 84

installing 11

installing, with Cloudera Manager 11, 12

installing, without Cloudera Manager 13

issues, URL 117

networking requisites 11

processing, strategy 108

Real-time query, on Hadoop 125

Real-time query subscription 125

requisites 9

resources 117

restarting 16

security 22

single point of failure 85

SQL statements, comments 62

SQL statements, unsupported 65, 66

starting 15

statestore UI 84

stopping 16

technology 119

troubleshooting 93

unsupported features 116

upgrading 16

upgrading, parcels with Cloudera Manager
used 17

upgrading, with Cloudera Manager 18

used, for data visualization 120

user account requisites 11

using, to query HBase tables 109, 110

VIEWS, defining 61

with Apache Hive 21

- with HBase 22
- with HDFS 22
- Impala 1.2.0 (Beta)**
 - about 126
 - URL 126
- Impala, core components**
 - about 18
 - Impala daemon 19
 - Impala metadata and metastore 20
 - Impala statestore 19
 - programing interface 20, 21
- impalad 27**
- Impala daemon 19**
- Impala Daemon (impalad) 28**
- Impala Download**
 - URL 117
- Impala events**
 - checking 104
- Impala metadata and metastore 20**
- Impala nodes**
 - adding 87
- Impala performance, fine tuning**
 - about 90
 - join queries 90
 - partitioning 90
 - table and column statistics 91
- Impala performance, improving**
 - about 85
 - block location tracking, enabling 85
 - Impala, enabling 86
 - Impala nodes, adding 87
 - memory usage, optimizing 87
 - native checksumming, enabling 86
 - query execution 87
 - resource isolation, using 87
- Impala Query Language 39**
- Impala Query Planner 107**
- Impala, security**
 - auditing 24, 25
 - authentication 24
 - authorization 23
 - guidelines 25
 - Impala specific guidelines 25
 - system specific guidelines 25

- Impala Shell**
 - and Impala daemon,
 - connectiivty issues 94, 95
 - command-line options 30, 31
 - commands 34, 35
 - connecting, to remotely located impalad
 - daemon 30
 - connection-specific options 32
 - database commands 72, 73
 - launching 29, 72
 - query-specific options 33
 - secure connectivity-specific options 34
 - table commands 72, 73
- impala-shell command 30**
- Impala Source**
 - URL 117
- Impala statestore 19**
- Impala Statestore Daemon (statstored) 28**
- Impala tables**
 - Avro file format 114
 - data, loading from HDFS 70, 71
 - data, loading in 69
 - Parquet file format 115
 - RCFile file format 114
 - regular Text file format 113
 - SequenceFile file format 115
- Impala v1.x Latest Documentation**
 - URL 117
- Input file format-specific issues 98**
- insert command 38**
- INSERT INTO statement 47**
- INSERT OVERWRITE statement 47**
- INSERT privilege 23**
- INSERT statement 47**
- installation**
 - Impala 11
- INT data type 49**
- internal table 48**
- ISNULL function 64**

J

- Java**
 - dependency, for Impala 10

- JDBC-specific connectivity issues 95
- JOIN clause 61
- join queries 90, 91
- JOIN query 91

L

- LCASE() function 55
- LEN() function 55
- LENGTH(STRING s) function 64
- LIKE operator 54
- LIMIT clause 59
- LOAD DATA statement 62
- log analysis
 - Cloudera Manager used 99, 100

M

- Maintenance Mode 103
- MapReduce 105
- Massively Parallel Processing. *See* MPP
- MAX aggregation function 56
- Memory consumption details
 - URL 93
- Microsoft Excel 122
- Microstrategy
 - about 123
 - URL 124
- MID() function 55
- MIN aggregation function 56, 57
- MPP 7

N

- native checksumming
 - enabling 86
 - issues 94
- NOW() function 55, 64

O

- ODBC-specific connectivity issues 95
- ODBO 122
- OLAP 122
- operator
 - about 52
 - BETWEEN 53
 - DISTINCT 53

- LIKE 54
- ORDER BY clause 59

P

- packages
 - with Cloudera Manager, used for upgrading Impala 17
- parcels
 - with Cloudera Manager, used for upgrading Impala 17
- Parquet file format 115
- PARTITIONED BY method 90
- partitioning 90
- PI() function 64
- platform-specific issues
 - about 97
 - HDFS-specific problems 98
 - Impala port mapping issues 97
- port mapping issues 97
- profile command 37

Q

- query execution
 - memory usage, optimizing on 87
 - on memory 87
- query_file=query_file_name command 33
- query performance, testing
 - about 88
 - data locality, verifying 88
 - queries, benchmarking 88
- query-specific issues 96
- query-specific SQL statements
 - about 60
 - EXPLAIN clause 60
 - JOIN clause 61
 - REFRESH clause 60
- quit command 35

R

- RAND(INT seed) function 64
- RCFile file format 114
- Real-time query
 - with Impala, on Hadoop 125
- Real-time Query. *See* RTQ
- REFRESH clause 60

- refresh command** 35
- regular Text file format** 113
- resource isolation**
 - using 87
- REVERSE(String a) function** 64
- ROUND() function** 55
- RTQ** 125

S

- Scalar functions**
 - about 55
 - FORMAT() function** 55
 - LCASE() function** 55
 - LEN() function** 55
 - MID() function** 55
 - NOW() function** 55
 - ROUND() function** 55
 - UCASE() function** 55
- select command** 38, 79
- SELECT privilege** 23
- SELECT statement** 47
- SequenceFile file format** 115
- set command** 36
- short-circuit read**
 - performing, on DataNode 86
- show databases command** 35
- SHOW DATABASES statement** 42
- SHOW TABLES statement** 45
- SMALLINT data type** 49
- SQL join operation**
 - SQL statements, types 77, 79
 - with example database 77
- SQL language statements**
 - database-specific statements 41
 - table-specific statements 43
- SQL queries**
 - against example database 74, 76
- SQL statements**
 - using 79
- statestore web interface**
 - using 102
- STRING data type** 51
- substr SQL command** 78
- SUM data type** 51

T

- Tableau** 121, 122
- table-specific statements**
 - about 43
 - ALTER TABLE statement** 44, 45
 - CREATE EXTERNAL TABLE statement** 44
 - CREATE TABLE statement** 43
 - DESCRIBE statement** 45, 46
 - DROP TABLE statement** 45
 - external table 48
 - INSERT statement** 47
 - internal table 48
 - SELECT statement** 47
 - SHOW TABLES statement** 45
- TIMESTAMP data type** 52
- TINYINT data type** 50
- TO_DATE(String date) function** 64
- troubleshooting**
 - configuration-related issues 93
 - connectivity issues 94
 - input file format-specific issues 98
 - platform-specific issues 97
 - query-specific issues 96
 - User Access Control (UAC)-specific issues 97
- type-conversion function** 65

U

- UCASE() function** 55
- unset command** 36
- use command** 38
- User Access Control (UAC)-specific issues** 97
- user-defined aggregate (UDA)** 116, 126
- User Defined Aggregation Functions (UDAF)** 65
- user-defined functions (UDF)** 126
- User Defined Table Generating Functions (UDTF)** 65

V

- version command** 35
- VIEWS** 61

W

web interface

for monitoring 101

for troubleshooting 101

WHERE clause 58

WITH clause 58

Y

YEAR(String date) function 64

Z

Zoomdata 124



Thank you for buying Learning Cloudera Impala

About Packt Publishing

Packt, pronounced 'packed', published its first book "*Mastering phpMyAdmin for Effective MySQL Management*" in April 2004 and subsequently continued to specialize in publishing highly focused books on specific technologies and solutions.

Our books and publications share the experiences of your fellow IT professionals in adapting and customizing today's systems, applications, and frameworks. Our solution based books give you the knowledge and power to customize the software and technologies you're using to get the job done. Packt books are more specific and less general than the IT books you have seen in the past. Our unique business model allows us to bring you more focused information, giving you more of what you need to know, and less of what you don't.

Packt is a modern, yet unique publishing company, which focuses on producing quality, cutting-edge books for communities of developers, administrators, and newbies alike. For more information, please visit our website: www.packtpub.com.

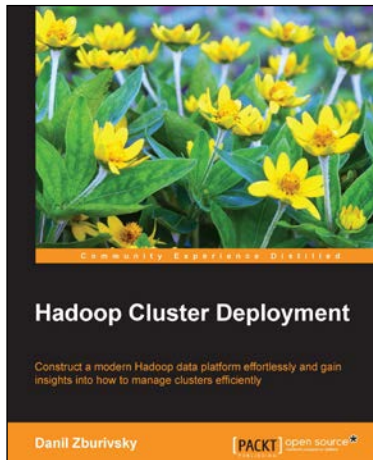
About Packt Open Source

In 2010, Packt launched two new brands, Packt Open Source and Packt Enterprise, in order to continue its focus on specialization. This book is part of the Packt Open Source brand, home to books published on software built around Open Source licenses, and offering information to anybody from advanced developers to budding web designers. The Open Source brand also runs Packt's Open Source Royalty Scheme, by which Packt gives a royalty to each Open Source project about whose software a book is sold.

Writing for Packt

We welcome all inquiries from people who are interested in authoring. Book proposals should be sent to author@packtpub.com. If your book idea is still at an early stage and you would like to discuss it first before writing a formal book proposal, contact us; one of our commissioning editors will get in touch with you.

We're not just looking for published authors; if you have strong technical skills but no writing experience, our experienced editors can help you develop a writing career, or simply get some additional reward for your expertise.



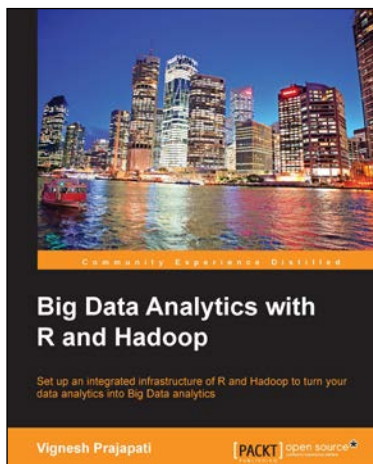
Hadoop Cluster Deployment

ISBN: 978-1-78328-171-8

Paperback: 126 pages

Construct a modern Hadoop data platform effortlessly and gain insights into how to manage clusters efficiently

1. Choose the hardware and Hadoop distribution that best suits your needs
2. Get more value out of your Hadoop cluster with Hive, Impala, and Sqoop
3. Learn useful tips for performance optimization and security



Big Data Analytics with R and Hadoop

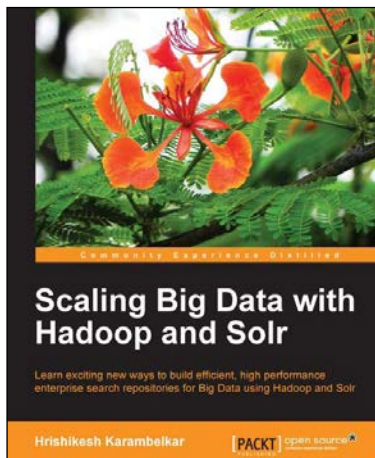
ISBN: 978-1-78216-328-2

Paperback: 238 pages

Set up an integrated infrastructure of R and Hadoop to turn your data analytics into Big Data analytics

1. Write Hadoop MapReduce within R
2. Learn data analytics with R and the Hadoop platform
3. Handle HDFS data within R
4. Understand Hadoop streaming with R
5. Encode and enrich datasets into R

Please check www.PacktPub.com for information on our titles



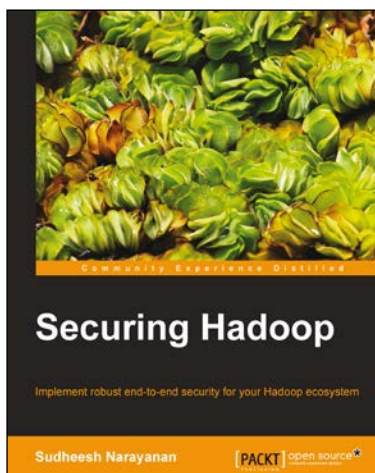
Scaling Big Data with Hadoop and Solr

ISBN: 978-1-78328-137-4

Paperback: 144 pages

Learn exciting new ways to build efficient, high performance enterprise search repositories for Big Data using Hadoop and Solr

1. Understand the different approaches of making Solr work on Big Data as well as the benefits and drawbacks
2. Learn from interesting, real-life use cases for Big Data search along with sample code
3. Work with the Distributed Enterprise Search without prior knowledge of Hadoop and Solr



Securing Hadoop

ISBN: 978-1-78328-525-9

Paperback: 116 pages

Implement robust end-to-end security for your Hadoop ecosystem

1. Master the key concepts behind Hadoop security as well as how to secure a Hadoop-based Big Data ecosystem
2. Understand and deploy authentication, authorization, and data encryption in a Hadoop-based Big Data platform
3. Administer the auditing and security event monitoring system