## ORIENTATION TO DATA IN CLUSTERS AND CLOUD STORAGE

LEARNING OBJECTIVES

- Use Hue or SQL utility statements to browse existing databases, tables, and table schema
- Explore files in HDFS and AWS S3 buckets, with Hue and with the command line

Introduction

## WELCOME TO THE COURSE

Welcome to this course, Managing Big Data in Clusters and Cloud Storage. I'm Ian Cook. I'm Glynn Durham. This course is the third in the specialization, Modern Big Data Analysis with SQL from Cloudera. In the first course, I teach the key concept behind relational databases, SQL or SQL, and big data. In the second course, I teach the fundamentals of the SELECT statement in SQL.

The statement you use to query data. In both of those courses, we emphasize open-source distributed SQL engines that can query extremely large datasets, Hive and Impala. But before you can run SQL queries using Hive or Impala, you need to have some data to query. The data needs to be loaded onto a cluster or in cloud storage, and it needs to be structured in tables with rows and columns. Data doesn't usually load and structure itself, somebody needs to do the work to get it that way.

If you have someone else that does that work for you, congratulations. That's great. Maybe you don't need to take this course. But if you'd like to gain the skills to do it yourself, then you're in the right place. That's what we'll teach you in this course. Knowing how to manage big data in clusters and cloud storage, is a valuable and marketable skill.

If you're working as a data analyst, knowing how to move data around and create your own tables, gives you a lot more power, more autonomy, and more career opportunities. These are essential skills for data engineers, data scientists, and

database administrators today. Managing large-scale data in a modern data warehouse is very different than it was in the age of traditional relational database systems. The information and the skills we'll teach you, are in growing demand today.

The skills we teach in this course and in the other courses in this specialization, are also designed to provide excellent preparation for the Cloudera Certified Associate Data Analyst Certification Exam. Earning this credential is a great way to stand out and be recognized by potential employers. You can earn it by taking a hands-on practical exam using the same SQL engines that we use throughout this specialization, Hive and Impala.

If you're interested in this certification, you should take the Honors track of this course. In the Honors lessons, you'll learn some advanced skills for overcoming obstacles, like when your data has complex structure, or your queries take too long to run. These are skills you'll need to practice data management in the real world. If you have not already taken the first two courses in this specialization, we strongly encourage you to take them, unless you already have a solid grasp of the material they cover.

Please go ahead now and read the next lesson item, review and preparation, which gives a whirlwind review of those two courses. That should help you decide whether to take them if you have not already. Maybe you're looking to elevate yourself from just querying data to loading and structuring data in clusters and cloud storage. Or maybe you're eager to update your database administration skills, and establish your credentials as a modern large-scale analytic database manager. This course and this specialization is part of will help you get there.

## REVIEW AND PREPARATION

The following is a review of the elements from the first and second courses of this specialization that are essential for you to be prepared for this third course. If you already took the first and second course, that's great, and this will just be a quick recap for you. If not, then this reading should help you understand what they're about so you can decide whether to take one or both of them before continuing with this third course.

***Please do not consider this a substitute for the first two courses.*** There is a lot more background in Course 1, and a lot more detail in Course 2, that will add much to your

understanding of these topics. If you find yourself getting lost in these recaps, please take the time to go through the other courses.

### The Virtual Machine Exercise Environment

This course uses the same virtual machine (VM) used by Courses 1 and 2 for practicing the material presented, and sometimes for completing quiz questions. If you have not already used this VM for a previous course, look at the technical requirements first, then (if your machine meets those requirements) follow the instructions for downloading and installing it.

### Course 1 Review

Moving into Course 3, it's important to understand how big data systems store and process data, including in the cloud, and how the tools access the data. These points were addressed in Course 1, along with more in-depth background information.

## Storing and Accessing Data, Comparison

**An RDBMS system** keeps your table definitions (that is, the schema) in a ***data dictionary***, which is ***tightly coupled*** with your tables: it's always kept in exact alignment, accurately describing the tables you create. This tight coupling also means that the schema governs what is allowed to be stored as data. These systems are called ***schema on write*** because the schema is applied before the data is stored. Databases manage all insertions and updates, and they typically throw an error if you try to do something like insert a character string value into a numeric column. If the data doesn't fit the schema, it can't be added to the table.

In a conventional RDBMS, data storage is ***encapsulated*** by your database software. Other programs cannot access the data storage directly: file access is usually blocked to any program other than the database software, and even with access, files are usually of a proprietary format that is not useable except through the database software. Figure 1 shows a SQL RDBMS with four attempts to access the data; the three using SQL are successful (green arrows) while the fourth, using another method (orange arrow with red X), is not.

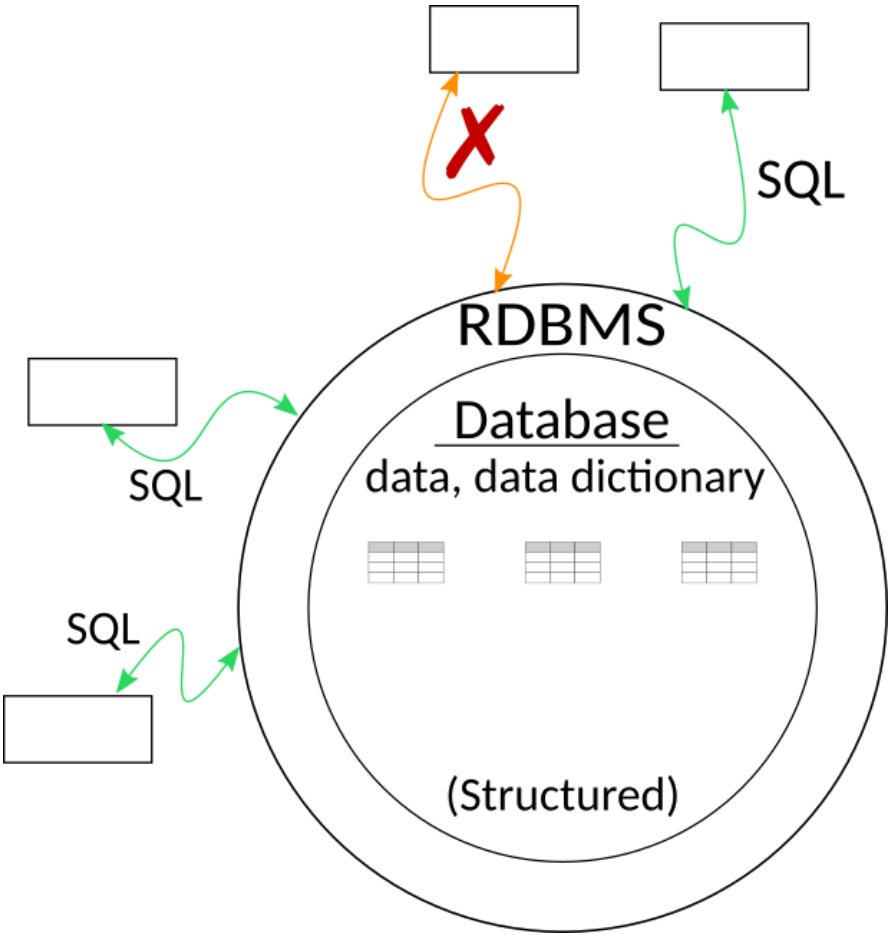



**Figure 1**

While the traditional RDBMS supports only structured data with access only through SQL, a **big data system** supports a variety of data, and also a variety of ways to access and use the data. The data store in your big data system can be called a ***data lake***, or ***data reservoir***, or ***enterprise data hub.*** The data lake can retain large varieties of data, of all sorts. Some contents might be structured and so easily usable by a SQL engine like Apache Hive or Apache Impala, and some might not be. See Figure 2.

Some programs read and write content directly to the data lake, using direct file access. These can be simple programs, written in languages like Python or Java or C, or large-scale distributed applications, like MapReduce or Spark programs. These programs can access files of potentially any format and type, and are suitable for working with structured or unstructured data.

In order to use SQL on your data, you create table definitions in a ***metastore***, which—for Hive and Impala—happens to be called the ***Hive metastore*** because of its origin as a part of Hive. The metastore takes the place of the data dictionary in an RDBMS: it contains table definitions that enable table-like access to some of the contents in the data lake. The metastore is not kept directly in the data lake, but alongside it.

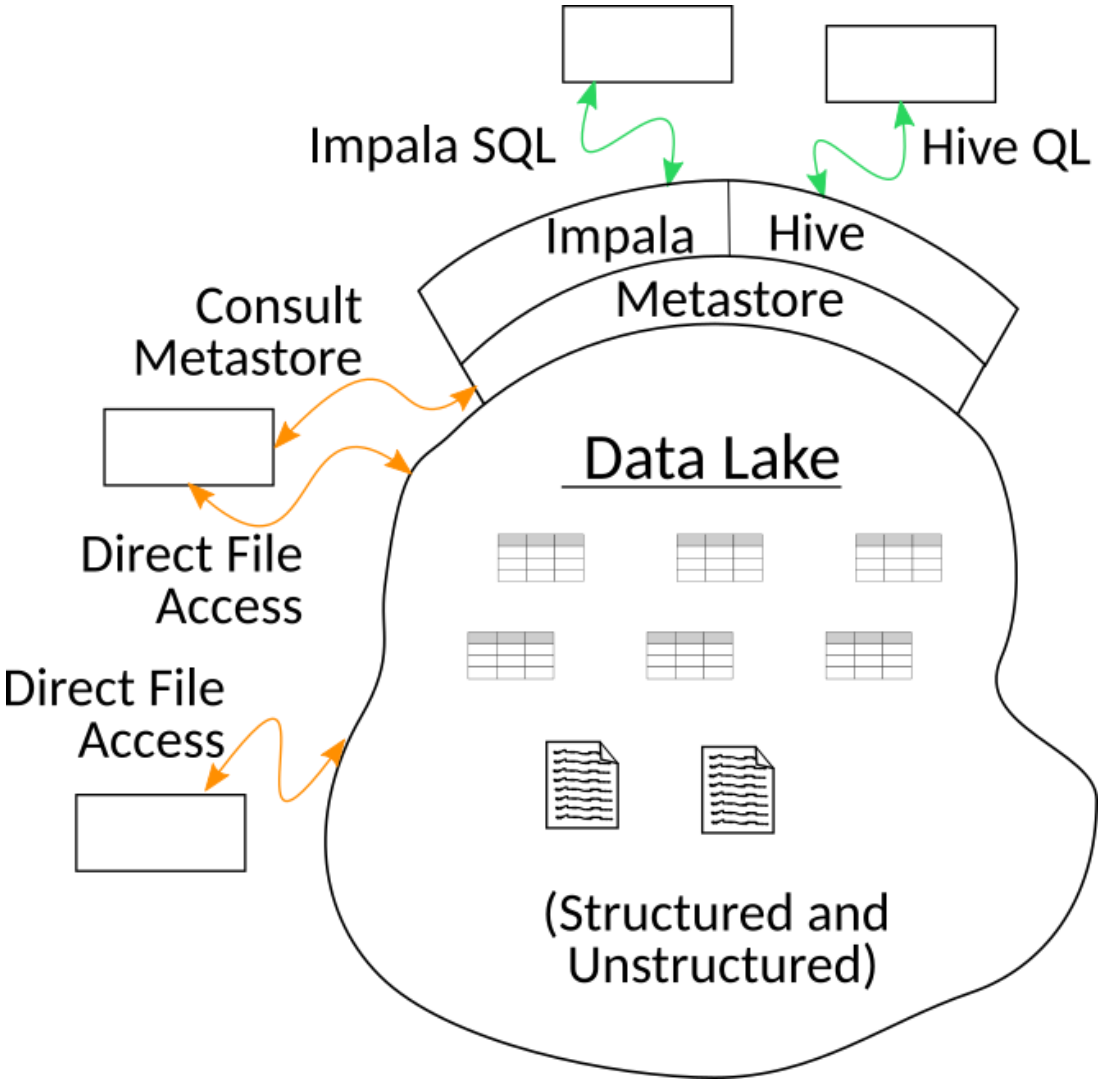



**Figure 2**

Because of the way big data systems separate your data and metadata, you can create table definitions that are ***loosely coupled*** to files. When your data and metadata are loosely coupled, your table definitions are not necessarily in lock-step with all your data. In fact, some files may reside in the data lake without any information about them in the metastore.

The table definitions also do not ***govern*** the file contents, but instead provide ***schema on read*** to let you view the files in table form. That is, the tool you use to access the data validates it when it's accessed (read), not when it's added (written) to the data lake. This lets your data lake accept files of any sort. The advantage is that loading data can be very fast, because it's nothing more than copying files. The disadvantage is that any errors in the data files will not be discovered until a query is performed. Missing or invalid data is generally represented as **NULL** values in query results, with two notable exceptions: missing **STRING** fields may be

represented as empty strings instead of NULLs; and in Impala, out-of-range numeric values are returned as the minimum or maximum allowed value for the data type.

Impala and Hive share the one metastore to find table and column definitions, and then access files in the data lake on your behalf when you issue SQL statements. Other applications, like Spark programs, can optionally consult the metastore to find out about table definitions for data, but this is not required in order to access the data. A single file may be used by an Impala query, a Hive query, a general-purpose Spark program, or any number of other programs. This is especially true when the file has structured or semi-structured contents.

## Distributed Storage and Processing

At the time of this writing, a single disk drive usually stores around a terabyte of data, or a handful of terabytes. Now consider if you want to store 30 terabytes, or 30 petabytes, or more! In modern technology there is no choice but to store your data across multiple disk drives, and the largest data stores must necessarily span thousands of disk drives. So, a big data store relies on ***distributed storage***, splitting data into pieces and scattering the pieces across many disks, instead of storing a single large file. Figure 3 shows a file split into pieces, sometimes called ***blocks***, with those blocks distributed across multiple disks for storage of the file. A typical size for a single block is 128 megabytes.

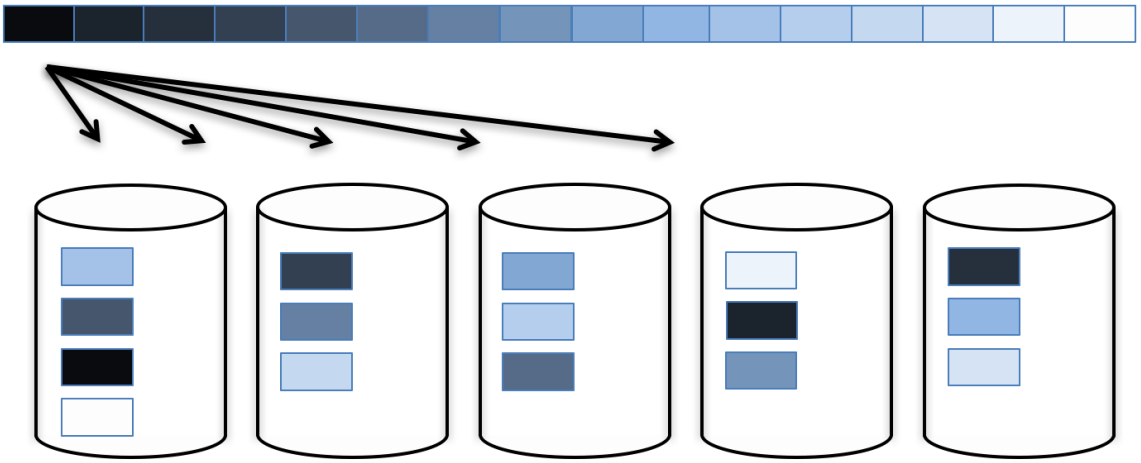


Figure 3

With this idea, if any one disk fails, then a part of your file is lost. With many disks, failure must be an expected, common occurrence. In order to keep files available, a distributed system keeps redundant copies of blocks on different disks. The system notices the failure of a disk, and it automatically makes additional copies of the lost blocks using the existing replicas.

An installation of a cluster of machines in a corporate data center is usually made up of banks of standard computers, each with its own memory, processors, and disk storage, combined in a

single computing cluster. Such a ***Hadoop cluster*** pairs computing power—memory and CPU—together with the disk storage.

Figure 4 illustrates parallel reads and concurrent processing for a program that finds all the records and the total net amounts for the purchases and payments of a few customers.

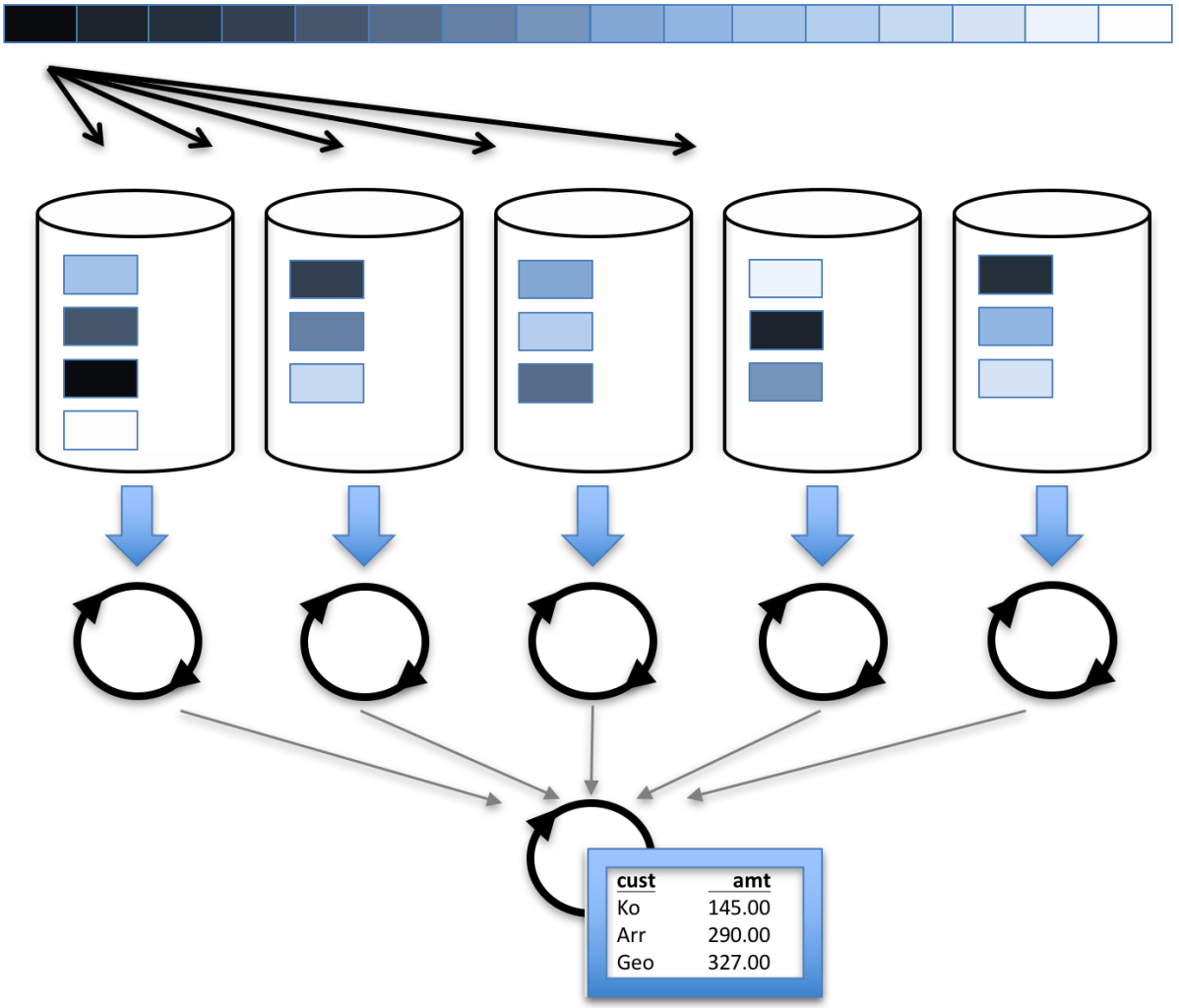



**Figure 4**

If your program calculation is sufficiently complicated, then there may be additional stages of distributed processing, in which intermediate results are reorganized by grouping or sorting, then processed further, and then there is final collection and display. See Figure 5. This intermediate reorganization of interim results can be called a ***shuffle*** of the data. In principle, a program may require any number of processing stages, with a shuffle between each stage.

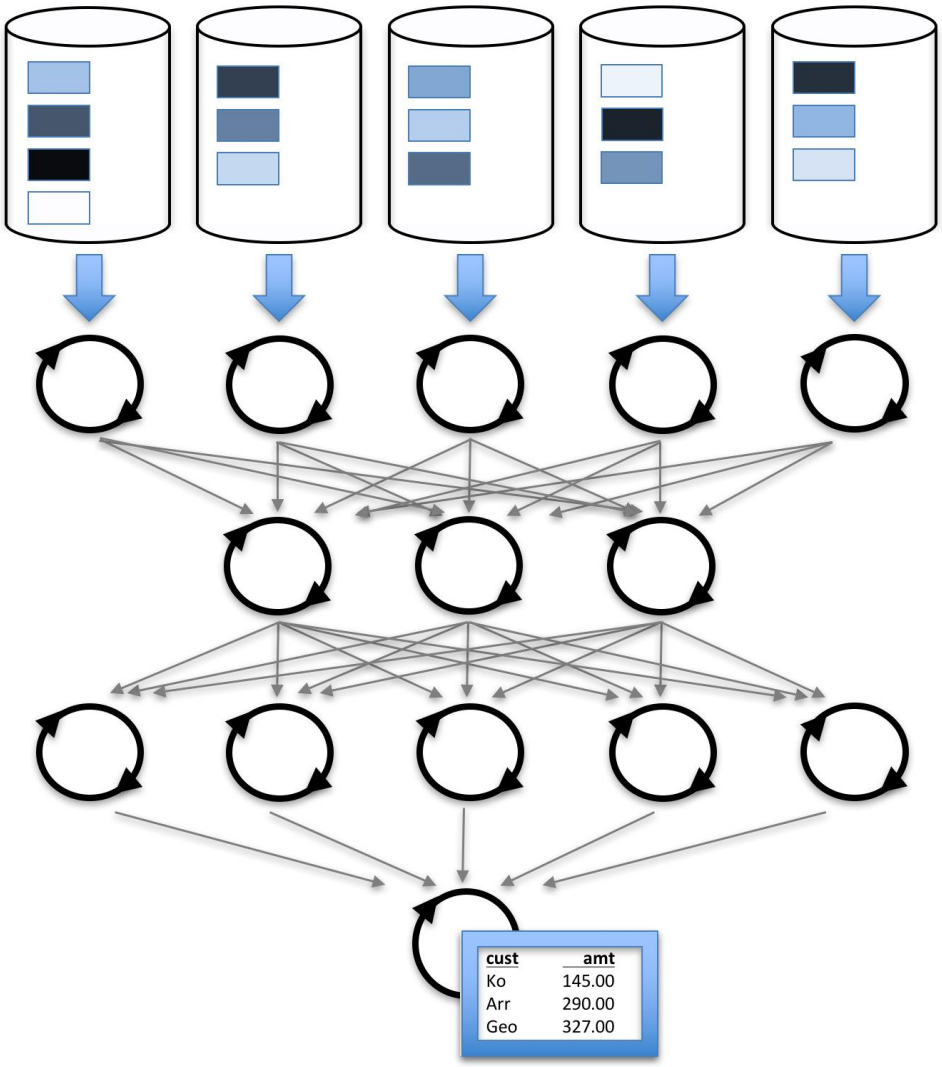



**Figure 5**

In the general case, your big data program may not produce a small amount of data to display, but may instead read a large data set, and produce another new large dataset. In this case, your program will perform not just distributed reads, but also distributed writes. See Figure 6.

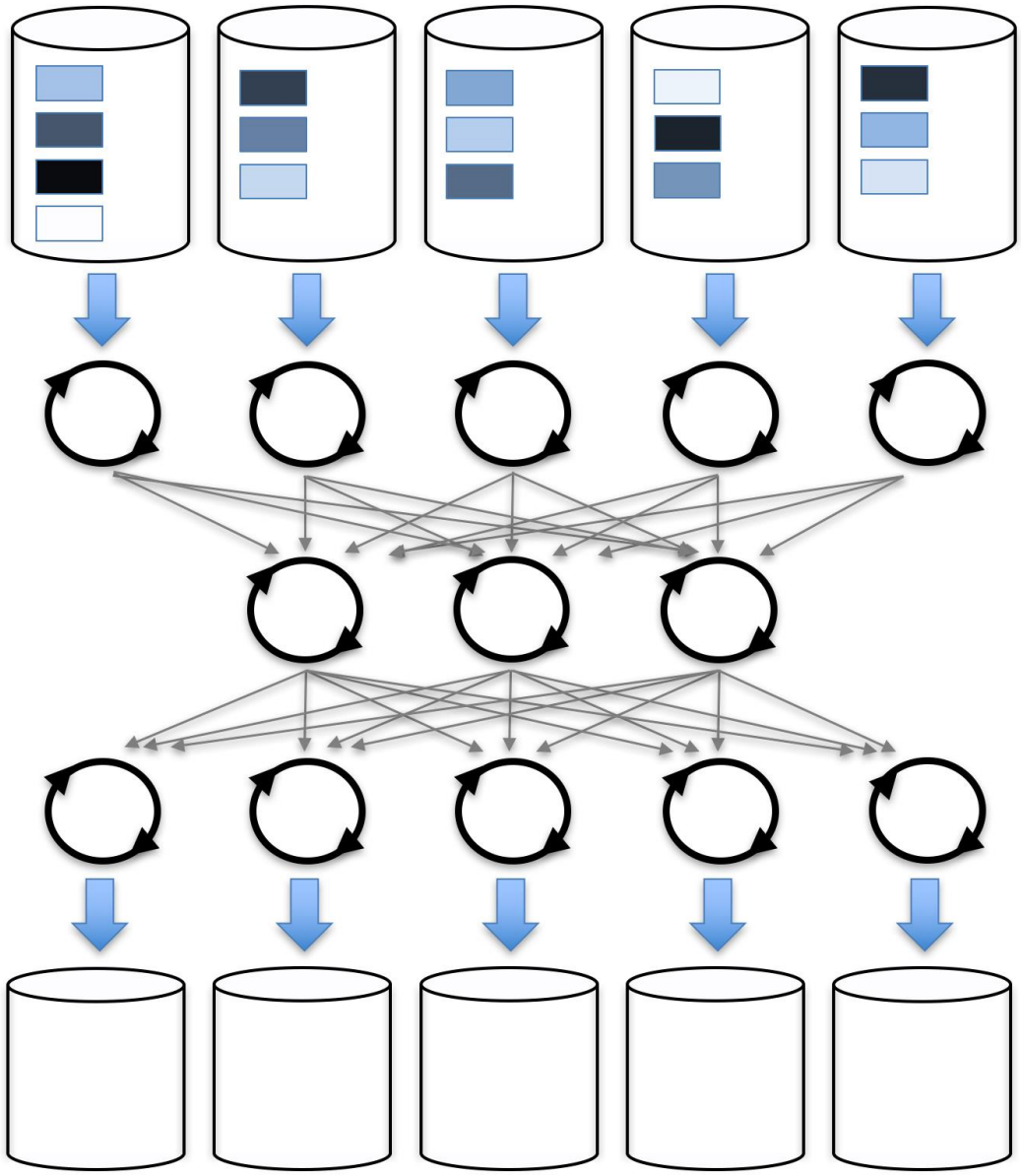


Figure 6

## Storage Options (Cloud, On Premises, or Hybrid)

The internet and modern computing have produced explosive growth in data volumes, but also new ways to store data. When storing petabytes of data for analysis, you have some options. Keep in mind, it's not just that you want to ***store*** your data—you also want to ***analyze*** it to gain new insights. So you really have related decisions to make, about storage and processing for your big data.

Some organizations physically store their data on servers in their own data centers. (A ***server*** is simply a computer that provides services to other computers through a network.) Storing data on servers in your own data center is called **on-premises** or ***on-prem*** storage. For a typical setup, each server in the cluster contributes both data storage (with a set of disk drives) and processing capacity (with CPUs and random access memory). Data storage on the disk drives in an on-prem ***Hadoop*** cluster is typically managed by the file system software called ***Hadoop Distributed File System*** (HDFS). With an on-prem cluster like this, you can increase storage and computing capacity together by adding more servers—also called ***nodes*** or ***hosts***—to the cluster.

You can store your data just as well using **cloud** services, such as Amazon Web Services (AWS), Microsoft Azure, or Google Cloud Platform (GCP). These services let companies choose to keep storage and computing power together, or keep data in cloud storage, and use (and pay for) computing power only when needed.

Or, you can take a **hybrid** approach, maintaining some data and computing power on premises, and some in the cloud.

## COURSE 2 REVIEW

Course 2 presents an in-depth look at the **SELECT** statement, the cornerstone of SQL-based data analysis. It introduces Hue as a tool for exploring databases and tables, and for running queries. Some utility statements for SQL are also presented.

### Databases and Tables

In Course 2, the word ***database*** refers to a logical container for a group of tables. Tables are organized into databases. Within one database, the tables all need to have different names, but two different tables can have the same name if they are in different databases.

However, this word ***database*** also has broader meanings. Any organized collection of data could be called a ***database***. SQL engines in general are often called ***databases***. And a specific ***instance*** of a SQL engine is often called a ***database***. In Course 1, the word ***database*** sometimes refers to these broader meanings. But in Course 2 and Course 3, the word ***database*** is primarily used to mean a logical container for a group of tables.

### Hue

Hue is a web browser-based analytics workbench. Among other things, you can use it to view tables, along with their schema and sample data, or to run queries using different SQL engines including Apache Hive and Impala. In the virtual machine (VM) supplied for these courses, Hue is assumed to be the main access point.

In Figure 7, panel 1 allows you to explore the databases and the tables within them. Panel 2 is the query editor, where you can enter SELECT statements or other statements you will learn about in this course. Panel 3 is an assistant panel that presents easy access to information about tables used in your queries.

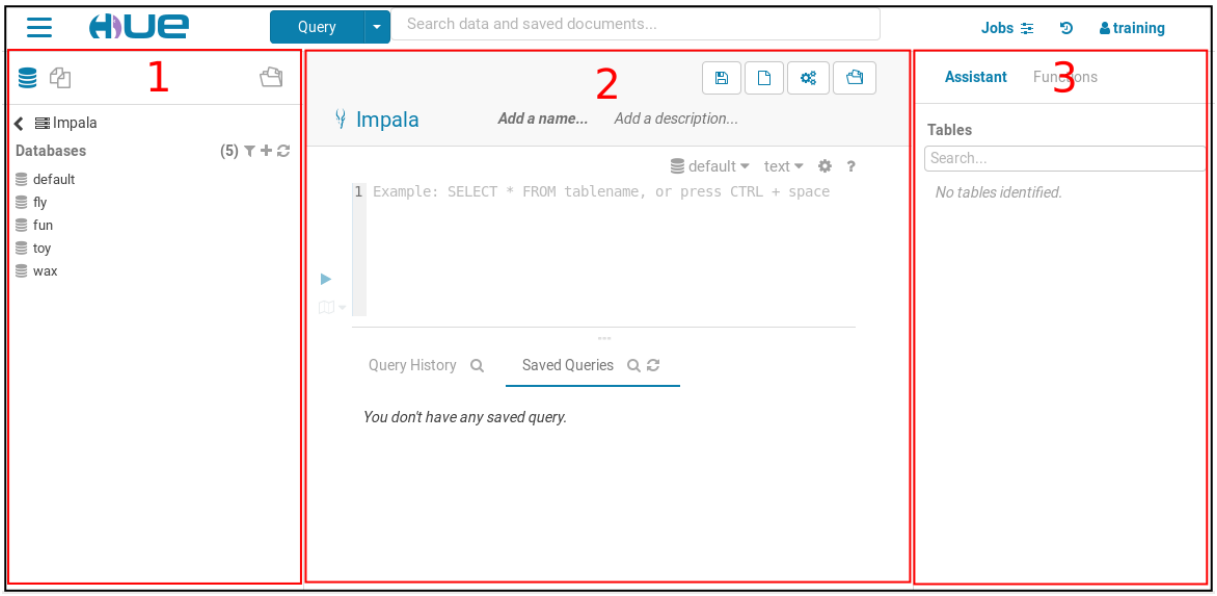



**Figure 7**

The "hamburger menu" (three horizontal lines) in the top left corner of Figure 7 provides access to additional areas of Hue.

## Utility Statements

Some SQL interfaces do not have the point-and-click capabilities of Hue, but you can use the following statements to do some of the exploration of databases and tables that Hue provides:

- **SHOW DATABASES;**
- **USE *database_name*;**
- **SHOW TABLES;**
- **DESCRIBE *table_name*;**

## SELECT Statements and Clauses

The **SELECT** keyword is the basis of every ***query*** performed in SQL. Following are the most common parts of a **SELECT** statement, all of which are covered in Course 2. If you are unfamiliar with any of these, please do complete Course 2 before moving on.

## The SELECT list

This specifies the columns that will be included in the result set. These might be columns from the source table or expressions that might or might not incorporate columns from the source table.

## FROM

The **FROM** clause is used to specify which table or tables will supply the data and columns used in the query. Multiple tables can be specified using **UNION**s or **JOIN**s (including implicit join syntax without explicitly using the **JOIN** keyword).

## WHERE

The **WHERE** clause filters the result rows, returning only rows for which a conditional (Boolean) expression evaluates to **true**. Rows for which the expression is **false** or **NULL** will not be returned.

## GROUP BY

Usually combined with an ***aggregate function*** in the **SELECT** list, the **GROUP BY** clause specifies a column or columns to use when combining multiple rows from the source table or tables into a single row in the result set.

## HAVING

This is another filtering clause that allows filtering ***after*** grouping using the **GROUP BY** clause. (The **WHERE** clause is applied before grouping and aggregation is calculated.)

## ORDER BY

The **ORDER BY** clause specifies how to sort the result set, if desired. Sorting can be presented in ascending order (using **ASC** or the default behavior) or descending order (using **DESC**).

## LIMIT

The **LIMIT** clause specifies how many rows are returned.

# INSTRUCTIONS FOR DOWNLOADING AND INSTALLING THE EXERCISE ENVIRONMENT

## ReadingTroubleshooting the VM

As you work with the VM, you might from time to time come up against some odd issues. Please consult this document as needed to help you troubleshoot and resolve the issues.

If you have worked through all the suggestions here and still have trouble, please reach out to your fellow students or the instructors through the forums for this course. Help your fellow students as you can, but your instructors will do their best to help you as soon as possible.

***Please try to resolve the issues on your own first.*** We understand how frustrating it can be, but you'll learn more if you try it on your own!

*Note:* Many issues can be solved by restarting the VM, so this should be the first thing you try. Restart the machine by clicking **System > Shut Down** from the menu bar, then click **Restart**. (Do not just pause or suspend the VM or quit the software running it.) Many times the issue is a service (such as Hive, Impala, Hue, HDFS, or one of the underlying services those rely on) going down, and restarting the machine will restart all these services.

## VM CPU and RAM Requirements

The VM for this class is designed to use one processor core (CPU core) and 4GB RAM. Reducing the amount of RAM to below 4GB is not recommended and is likely to cause failures when running some queries on large tables. Increasing the amount of RAM to some amount greater than 4GB is unnecessary, but it will not cause problems so long as your computer has sufficient RAM to allow it. You should always leave at least about 4GB available for the operating system outside the VM to use. For example, if your computer has 8GB total ram, you should never configure the VM to use more than about 4GB. If your computer has 16GB total RAM, you should never configure the VM to use more than about 12GB.

However, you should ***not***increase the number of processor cores (CPU cores) used by the VM. If you do increase the number of processor cores used by the VM, then it is absolutely necessary to also increase the amount of RAM. For example, with two processor cores, you should use at least 6GB RAM. Increasing the number of processor cores without also increasing the amount of RAM is likely to cause failures.

## VM Is Slow

If the VM is running slowly, it might be that you are using too many resources for the memory available to the VM. If you're using Hue, first try closing the browser and reopening it. This sometimes clears out the resources.

If that doesn't help, then in the VM, go to **System**> **About this Computer**> **Resources** to see how much CPU and memory (RAM) is being used.

If your RAM usage is high, close all applications and browser windows or tabs except the one you're using. Avoid having Hue open in multiple browser windows or tabs, because this can use a lot of RAM.

## SERVICES NOT AVAILABLE

Occasionally you might find that a service or process on the VM has failed and needs restarting. The simplest way to do this is to restart the VM. See the note above the table of contents, above.

### Errors in Hue

Clicking around in certain parts of Hue that are not part of the exercises might result in error messages. The Job Browser is one such example. Clicking it might show a red popup layer in the browser with an error message similar to this:

**HTTPConnectionPool(host='localhost', port=11000): Max retries exceeded**

**with url: /oozie/v1/jobs?len=100&doAs=training&filter=user%3Dtraining %3Bstartcreatedtime%3D-7d&user.name=hue&offset=1&timezone=America %2FLos_Angeles&jobtype=wf (Caused by NewConnectionError('<requests.packages.urllib3.connection.HTTPConnection object at 0x7f4f1c53f090>: Failed to establish a new connection: [Errno 113] No route to host',))**

This occurs because the Job Browser depends on a component called Oozie, which we do not include in the VM. Oozie is not used in any exercises for this course; installing it would make the VM larger and require more memory to run, which would reduce performance.

Note that the job browser is not the only place where errors like this might occur. Other areas of Hue might yield errors related to other components that have not been installed. In general, we have tested that the exercises related to Hue work without error. If you deviate from the exercise instructions, then you might encounter errors such as the one described above.

### Difficulty Connecting to Beeline

Your first step when a command doesn't work as expected should always be to check ***carefully*** for typos! Be sure the command you are using to start Beeline is

**beeline -u jdbc:hive2://localhost:10000**

It's very easy to type **jbdc** instead of **jdbc**, for example, and easy to overlook that typo.

It's also easy to use the wrong number of 0s at the end. (There should be four 0s.)

## Difficulty Connecting to Impala Shell

Your first step when a command doesn't work as expected should always be to check ***carefully*** for typos! Be sure the command you are using to start Impala Shell is

**impala-shell**

It's very easy to forget to use a dash and instead use a space, for example.

## Difficulty Connecting for S3 or Other Internet Services

Some commands (such as any commands that interact with S3, the cloud service we're using for these courses) require that the VM itself be connected to the internet. In the upper right corner, find one of these icons to determine the connection state and what you should do:

The VM is connected; if there is a problem, check your computer's connection rather than the VM's connection.

(Animated) The VM is trying to connect; give it a moment until it resolves to one of the other two icons.

The VM is disconnected; click the icon and choose **Auto Ethernet** to reconnect. If the problem persists even after you reconnect the network, then restart the VM.

## BROWSING TABLES IN THE METASTORE

In this course you'll learn to create tables for Hive and Impala and load data into them. But before you can create new tables, you'll need to see what tables already exists. The easiest way to do this is by using Hue.

Hue is a web browser based analytics workbench that provides a user interface to Hive, Impala, and other tools. It's widely used in the real world and it's installed on the VM that we use in this specialization. If you completed the first and second courses in this specialization, you already learned how to use Hue, so some of this should be familiar.

With the VM set up and running, you can access Hue by opening the web browser in the VM, and clicking the link for Hue in the Bookmarks toolbar. Hue includes a number of different interfaces. Many of which, you will not use in this course. There are just a few interfaces that you will use. Recall from the earlier courses in this special edition that one of the ones you do need to use is the table browser. Click the Menu icon in the upper left corner, then under browsers click Tables. Here in the table browser, you can click Databases to see what databases exist.

You can see there are databases named default, fly, fun, toy, and wax. You can click the name of a database to see what tables are in it. I'll click fun. You can see that the fun database has tables named card_rank, card_suit, games, and inventory. Then you can click on the name of the table to see more details about that table. I'll click games. If you go to the Columns tab, you can see that this games table has 8 columns, id, name, inventor, year, mean age, mean players, max players, and least price. You can also click the sample tab to see a sample of the data in this table.

So the table browser in Hue provides a convenient interface for browsing tables through simple point and click actions. And if you completed the first and second course in this specialization, you should recall it as well. In a big data environment where you're running Hive or Impala, the information about what databases and tables exist, and what columns are defined in them, is stored in something called the metastore. This is sometimes called the Hive metastore because of its origin as a part of Hive. So when you use Hue to browse the databases and tables that

are available to Hive and Impala, Hue gets that information from the metastore and presents it to you. Recall that Hue also has an interface for running SQL queries on tables. Notice the BigQuery button in the top bar in Hue. When you click the right side of this button, a drop down menu opens. Under Editor, in the drop down menu, you'll see options for Impala and Hive. There are some other options below that.

You can ignore those for now. I'll click Impala. And this opens up the query editor for Impala. The query editor for Hive is nearly identical. It says Hive at the top instead of Impala. But besides that, it looks the same. All the features I'll describe here are also available in the Hive query editor. When you're in the query editor, you can use this assist panel on the left side to browse the databases and tables. If the assist panel is hidden, you can click to show it. And you should make sure that the SQL mode is active at the top of the assist panel.

Click this database icon to make sure that's the active mode. You can use the assist panel to see what databases exist. If you're already in one of the databases, you'll have to click this back arrow to go back to the list of all the databases. You can click the name of the database to see what tables are in it. If you click the name of a table, you'll see the columns in it. And you can click the letter i icon to the right of a table name to see more details about the table.

This is all very similar to what you can do through the table browser I showed earlier in this video. Recall that Hive and Impala both access the same tables. These are two different engines operating on one set of underlying tables and data. So on the VM when you choose to use the Hive query editor or the Impala query editor, you're simply choosing which SQL engine will run the queries on that shared set of tables. Sometimes people call these shared tables Hive tables because the information about them is stored in Hive's metastore.

But if you hear this terminology, Hive tables, it does not mean that those tables are available only to Hive. They're also available to Impala and to some other engines like Apache Spark. So if you're using a Hadoop cluster or a big data platform that's based on Hadoop, multiple engines and other tools can all access the shared tables that are defined in the metastore. However, there are other SQL engines that do not run in Hadoop environments. They do not use the metastore, they maintain their table structures and store their data in other places. For

example on the VM, you might notice that we have two other SQL engines installed, MySQL and PostgreSQL. And you can run queries using these engines in Hue, but the tables and data that these engines use is totally separate. And in this course we will not use MySQL or PostgreSQL. We'll only be using Hive and Impala. In the upcoming weeks of this course, you'll learn how to create new databases and tables in Hue and how to modify or remove existing ones.

## BROWSING TABLES WITH HUE

In this course you'll learn to create tables for Hive and Impala and load data into them. But before you can create new tables, you'll need to see what tables already exists. The easiest way to do this is by using Hue. Hue is a web browser based analytics workbench that provides a user interface to Hive, Impala, and other tools. It's widely used in the real world and it's installed on the VM that we use in this specialization. If you completed the first and second courses in this specialization, you already learned how to use Hue, so some of this should be familiar.

With the VM set up and running, you can access Hue by opening the web browser in the VM, and clicking the link for Hue in the Bookmarks toolbar. Hue includes a number of different interfaces. Many of which, you will not use in this course. There are just a few interfaces that you will use. Recall from the earlier courses in this special edition that one of the ones you do need to use is the table browser. Click the Menu icon in the upper left corner, then under browsers click Tables.

Here in the table browser, you can click Databases to see what databases exist. You can see there are databases named default, fly, fun, toy, and wax. You can click the name of a database to see what tables are in it. I'll click fun. You can see that the fun database has tables named card_rank, card_suit, games, and inventory. Then you can click on the name of the table to see more details about that table. I'll click games. If you go to the Columns tab, you can see that this games table has 8 columns, id, name, inventor, year, mean age, mean players, max players, and least price. You can also click the sample tab to see a sample of the data in this table. So the table browser in Hue provides a convenient interface for browsing tables through simple point and click actions. And if you completed the first and second course in this specialization, you should recall it as well. In a

big data environment where you're running Hive or Impala, the information about what databases and tables exist, and what columns are defined in them, is stored in something called the metastore. This is sometimes called the Hive metastore because of its origin as a part of Hive.

So when you use Hue to browse the databases and tables that are available to Hive and Impala, Hue gets that information from the metastore and presents it to you. Recall that Hue also has an interface for running SQL queries on tables. Notice the BigQuery button in the top bar in Hue. When you click the right side of this button, a drop down menu opens. Under Editor, in the drop down menu, you'll see options for Impala and Hive. There are some other options below that. You can ignore those for now.

I'll click Impala. And this opens up the query editor for Impala. The query editor for Hive is nearly identical. It says Hive at the top instead of Impala. But besides that, it looks the same. All the features I'll describe here are also available in the Hive query editor. When you're in the query editor, you can use this assist panel on the left side to browse the databases and tables. If the assist panel is hidden, you can click to show it. And you should make sure that the SQL mode is active at the top of the assist panel.

Click this database icon to make sure that's the active mode. You can use the assist panel to see what databases exist. If you're already in one of the databases, you'll have to click this back arrow to go back to the list of all the databases. You can click the name of the database to see what tables are in it. If you click the name of a table, you'll see the columns in it. And you can click the letter i icon to the right of a table name to see more details about the table. This is all very similar to what you can do through the table browser I showed earlier in this video.

Recall that Hive and Impala both access the same tables. These are two different engines operating on one set of underlying tables and data. So on the VM when you choose to use the Hive query editor or the Impala query editor, you're simply choosing which SQL engine will run the queries on that shared set of tables. Sometimes people call these shared tables Hive tables because the information about them is stored in Hive's metastore. But if you hear this terminology, Hive tables, it does not mean that those tables are available only to Hive. They're also

available to Impala and to some other engines like Apache Spark. So if you're using a Hadoop cluster or a big data platform that's based on Hadoop, multiple engines and other tools can all access the shared tables that are defined in the metastore.

However, there are other SQL engines that do not run in Hadoop environments. They do not use the metastore, they maintain their table structures and store their data in other places. For example on the VM, you might notice that we have two other SQL engines installed, MySQL and PostgreSQL. And you can run queries using these engines in Hue, but the tables and data that these engines use is totally separate. And in this course we will not use MySQL or PostgreSQL. We'll only be using Hive and Impala. In the upcoming weeks of this course, you'll learn how to create new databases and tables in Hue and how to modify or remove existing ones.

---

## BROWSING TABLES WITH SQL UTILITY STATEMENTS

Recall from the previous video that Hue enables you to see what databases exist, switch into a particular database. See what tables are in it, and look at the columns in those tables all through point and click actions. In this video, I'll show how, for each of those tasks, you can write and run a SQL utility statement that does the same thing as the point and click action. So if you could do something by pointing and clicking, why would you want to write a SQL statement to do it?

Well, for one, not all SQL interfaces have a graphical user interface like Hue does. Sometimes the only way to perform simple tasks like these is by entering and running SQL statements. Also, an interface like Hue enables you to perform some simple tasks without using SQL. But as you'll see beginning in the next video, you can achieve many more types of tasks by entering and running SQL statements. The first SQL utility statement I'll talk about is SHOW DATABASE.

This is often the very first statement you would run when connecting to an instance of a SQL engine for the first time, it tells you what databases exist. I'll run this statement in the Impala Query Editor in Hue on the VM to show you what it returns. In the editor, I'll enter SHOW DATABASES, and I'll terminate the statement with a semicolon. The convention in SQL is to use a semicolon to indicate the end of every statement. But here in Hue, if you're just running a

single statement, the semicolon is optional, so you could leave it off. Then I'll click this Execute button to run the statement. You can also use the keyboard shortcut Ctrl+Enter to do this. After the statement runs, the result appears directly below. Running the statement with Impala, you can see the first row of the results lists a system database named impala_builtins.

You can safely ignore that, below that you can see the actual databases. Each one has a name and, optionally, a comment. You can see there are databases named default, fly, fun, toy, and wax. When you're using a SQL engine, there is always one particular database that you're connected to. This is called the current database, or the active database. When you first login to Hue and open the Hive or Impala Query Editor, the current database is typically the one named default.

Other SQL engines also have particular databases that they connect to by default at the start of a new session. But they're not generally named default, and they can vary from user to user. Usually, if you're going to be working with a particular table, you'll want to set the current database to the database that contains that table. To set which database is the current database, you can run a USE statement. The USE statement is very simple, it's just the keyword use followed by the name of a database. Hue, which you'll use throughout this course, actually does not support the USE statement. Instead, in Hue, you always use point and click actions to set the current database. Just above the editor, there is an active database selector.

You can use that to see what the current database is, right now it's default, and to change the current database. I'll select fun, and now the current database is fun. Alternatively, you can click the name of a database in the assist panel on the left side, and Hue will set that as the current database. In the assist panel, I'll click the back arrow to go back to the list of all the databases, then I'll click the wax database. And you can see in the active database selector that wax is now the current database.

The current database persists for the duration of your session or until you change it again, I'll change it back to fun. If you're using a SQL interface that lacks a point and click interface for switching databases, then you would instead need to execute a USE statement, like USE fun. Also, keep in mind that your selection of the current database only affects the particular session in which it is run. Other

users in other sessions have their own current databases. Recall that a database in SQL is just a logical container for a group of tables. So after you see what databases exist and change the current database, often the next step is to see what tables exist in the current database. To do this, run the statement SHOW TABLES, I'll enter and run SHOW TABLES in Hue. And the result shows the names of the four tables in the current database, which, remember, is the fun database. The tables are card_rank, card_suit, games, and inventory.

The final utility statement I'll talk about is the DESCRIBE statement. You can use the DESCRIBE statement to see what columns are in a table. The syntax is simple, following the keyword describe, you put the name of the table whose columns you want to see. I know that one of the tables in the fun database is named games. So in Hue, I'll enter and run DESCRIBE games. The result shows that this table has eight columns, id, name, inventor, year, min_age, min_players, max_players, and list_price.

Each column also has a data type and, optionally, a comment. You'll learn about data types later in this course. So for now, the column names and the order they're in are what you should pay attention to. So now you've seen how to use SQL utility statements to explore and to navigate databases and tables. SHOW DATABASES shows you what databases exist. The USE statement changes the current database. SHOW TABLES shows what tables are in the current database. And the DESCRIBE statement shows what columns are in a table.  Browsing Files in HDFS

## BROWSING HDFS WITH THE HUE FILE BROWSER

Recall that Hive and Impala are SQL engines that run on clusters or big data platforms that are based on Hadoop. Hadoop based clusters or platforms include a system for files storage. It's called the Hadoop Distributed File System or HDFS. The first course in this specialization introduces HDFS and describes how it's different from other file systems like the File System on your local computer.

The VM that you use throughout this specialization has HDFS installed on it and all the data in the tables on the VM is stored in files in HDFS. Each Hive and Impala table has two components. One, its metadata which is stored in the metastore as I described in a previous video. Two, its data which is typically stored in HDFS. If

you took the first course in this specialization, you should be familiar with this. You should recall that metadata and data are stored in this separate places and are loosely coupled. When Hive or Impala receives a query, it needs to use both the metadata from the metastore and the data typically from HDFS to generate the query results.

So here's what happens when you run a query with Hive or Impala. First, it accesses the metastore to determine the structure of the table that you specified in your query. The metastore tells Hive or Impala what columns are in the table. It also says where to find the data for that table. Typically, this is a path to a directory in HDFS. Then Hive or Impala retrieves the table data from the files in that directory in HDFS. It processes those files to generate the query results. So that's the background you need to remember about how Hive and Impala use the metastore and HDFS.

In the previous lesson, you learned how to browse tables in the metastore. In this lesson, you learn how to browse files in HDFS. The easiest way to browse files in HDFS is through Hue. In Hue, click the menu icon in the upper left corner. Then under browsers click files. This takes you to Hue's file browser. This interface lets you browse the directories and files in HDFS. When you first open the file browser, it takes you to the directory slash-user slash-training. On the VM, this is your home directory in HDFS. You can see the directory path slash-users slash-training here.

In a real-world environment, your HDFS home directory is typically slash, user, slash, your username. In a real-world environment that is shared with other users, your home directory is the place where you can experiment with creating subdirectories, uploading files and so on. It's like your own personal workspace. So what you do there will not interfere with other users files. You should have full permissions on your own home directory so you can experiment there on your own without any help from an administrator. On the VM at your home directory should be empty at first like you see here.

The file listing here shows a dot representing this directory and an upper arrow representing the parent directory. I'll click the upper arrow and this takes me to the parent directory slash-user. I can return to my home directory by clicking training or by clicking the home icon here. Now, I'm back in slash-user slash-

training. You can also use the path editor here to navigate directories. If you click the slash at the beginning of the path, that takes you to the root directory of HDFS. If you click a blank area of the path editor, it changes to a text field where you can enter the path you want to navigate to. I'll enter slash-user slash- training and press enter to return to my home directory.

You can also click a directory name in this path editor to navigate to that directory. I'll click user to go to the slash user directory. Here in this slash-user directory, you can see your own home subdirectory training and some other subdirectories. One of these is named hive. I'll click to navigate there and in it there is a subdirectory named warehouse. I'll click to navigate into that. This directory in HDFS at the path slash-user slash-hive slash-warehouse is a special directory known as the Hive Warehouse directory. The Hive Warehouse directory is the place where Hive and Impala store table data by default.

In this directory, you'll see some subdirectories with names that are familiar if you've been through Course 2, in this specialization. Customers, employees, offices, orders and salary grades, these are the names of tables in the default database. For each table in the default database, there is a subdirectory of the same name here in the Hive warehouse directory. Let's see what's in one of these subdirectories. I'll navigate into the one named orders, in it there's a file named orders.txt. I'll click that file to view it.

This is a tab separated text file. You can see there are five lines in the file and each line has four values, separated by tabs. This is the data in the Orders table. Recall from looking at the Orders table in the table browser that that table has four columns. Order id, cast id, ample id and total. The values in those columns are the values you see in these four columns in this text file. The column names and data types are not stored here in HDFS. They are stored in the metastore.

Only the values are stored here in this file. I'll click to return to the Hive warehouse directory. Notice that there are two subdirectories here named investors and investors parquet. Those both contain data but we have not yet created tables to query that data from Hive or Impala. We'll come back to those subdirectories later in this course. Notice that there are some other subdirectories here with names ending in .DB; fly.DB, fun.DB, toy.DB, and wax.DB. These represent the databases. Recall that there are databases named fly, fun,

toy, and wax. For each database, except the one named default. There's a corresponding subdirectory in the Hive warehouse directory named database name.DB. Let's see what's in one of those. I'll click to go into fun.DB.

There you can see four subdirectories named card rank, card suit, games and inventory. These contain the data for the four tables with the same names. I'll click to go into the games subdirectory and in it there's a file named games.csv. I'll click to view that. This is a comma separated text file. It has five lines and each line has eight values separated by commas. This is the data in the games table, in the fun database.

You can download a file from HDFS to your local file system by clicking this download button in Hue. To open a file with most applications that run on your local computer, you need to download the file from HDFS to your local file system. For example, after downloading this file to the local file system in this VM. I can open it in the text editor on the VM. Besides using hue's file browser, you can also use the assist panel on the left side to browse HDFS. If the assist panel is hidden, you can click to show it.

Then switch the assist panel from SQL mode to HDFS mode by clicking this pages icon. This interface lists the files and directories in HDFS and lets you navigate through them. For example, I can go into the Hive warehouse directory at slash-user, slash-hive, slash-warehouse. This is similar to what you can do through the file browser. I'll click the database icon to switch the assist panel back into SQL mode. For most of this course, you'll find it most useful in SQL mode. So I showed the file containing the data for the Orders table in the default database and the file containing data for the games table in the fun database.

One was a tab separated text file. The other was a comma separated text file. This demonstrates an important point about Hive and Impala. There is not just one fixed format for storing table data. Hive and Impala support storing table data in a variety of different formats. These two files I showed both had their data stored in plain text files but with different delimiters. Data can also be stored in other file formats like Avro and Parquet. You'll learn about those later in this course. For now, just know that if you see any files in table directories in HDFS that do not look like plain text, it's because they're in one of these other file formats. Also, these files I showed were both under the Hive warehouse directory. That's the

place where Hive and Impala store table data by default. But it is possible to specify some other location in HDFS as the place where the data for a particular table is stored. For example, for all the tables in the fly database the data is not stored under the Hive warehouse directory. Instead, it's stored in the directory named fly under the root directory of HDFS. Later in the course, you'll learn more about this.

For now just don't be surprised if you noticed that some tables do not have data within the Hive warehouse directory. Finally, as you browse these directories in HDFS please do not move, copy, modify, or delete anything and don't upload any new files to HDFS. You'll learn how to do all of that later in the course. But in the meantime, please don't because that could cause the VM to behave in unexpected ways. A quick note for anyone who's using an older version of Hue, the link to access the file browser might be in a different place. If you don't see a menu like this with a files link in it, then you're probably using an older version of Hue. If so, look for a file browser button at the top right with a page icon that will take you to the file browser.

## BROWSING HDFS FROM THE COMMAND LINE

In the previous video, you learned how to use Hue to browse HDFS. Hue provides a web browser-based graphical user interface to HDFS. In this video, you'll learn how you can also browse HDFS from the command line. So if it's possible to use an easy graphical user interface to browse HDFS, why would you want to use the command line? Well, there are several reasons. Maybe you're just an old school computer nerd and you love the command line.

If so, that's cool. But there are some other good reasons to use the command line to interact with HDFS. Entering commands is a more systematic way of accomplishing a task. If you can perform a task on the command line, that means you can also script it, automate it, schedule it, and more. By saving your commands in a file, you can effectively document the steps you performed and make the task reproducible. To interact with HDFS on the command line, you use the command hdfs dfs, followed by various commands and options and arguments. These are called HDFS shell commands or Hadoop file system shell commands. I'll demonstrate a few of these now. With the VM running, open the

terminal. This is the command line for the Linux operating system on the VM. At the command line, enter hdfs dfs, followed by a command indicating what action to perform. For example, to list the contents of a directory in HDFS, You can use dash ls. This command ls derives from the word list. After dash ls, specify the directory path.

For example, I'd like to see the contents of the Hive warehouse directory. So I'll specify /user/hive/warehouse/. The trailing slash is optional but I like to include it. It makes it clearer that this is a path to a directory. When I press Enter, you can see that the listing of the contents of that HDFS directory is printed to the screen. Besides the paths to the items in this directory shown in the rightmost column, this listing also shows whether the item is a directory represented by the letter d, or a file represented by a dash.

The permissions, read, write and execute on the item for the owner, group and world. You'll learn more about HDFS permissions later in the course. This column, which you can ignore, it's related to symbolic links which are outside the scope of this course. The owner of the item, the group owner of the item, the size of the item in bytes which is zero if it's a subdirectory, and the date and time when the item was last modified. You can see there's a subdirectory here named orders. Recall that that's where the data for the orders table in the default database is stored. I'd like to list the contents of that subdirectory. So at the command line, I'll press the up arrow to recall the previous command and I'll add orders/ to the end of the path.

I'll press Enter, and then you can see the listing of this directory. There's just one file in it, orders.txt. To print the contents of this file to the screen, you can use a different command after hdfs dfs. It's dash cat. This command cat derives from the word concatenate. After a dash cat, you specify the full path to a file in HDFS. For example, I want to look at that orders.txt file. So I'll specify /user/ hive/ warehouse/orders/orders.txt. When I press Enter, the contents of that file are printed to the screen. The final hdfs dfs command I'll demonstrate in this video is dash get. You use this command to download or copy a file from HDFS to your local file system. I'll press the up arrow to recall the previous command. I'll replace dash cat with dash get.

At the end of the command after a space, I'll add dot, indicating that I want to download this file to the current working directory in my local file system. Then after I press Enter, this file orders.txt is downloaded. If you have any experience with Linux shell commands, you'll recognize that some of these HDFS shell commands, ls and cat, are named after the analogous Linux shell commands. For example, I can use the ls command to list the contents of the working directory in the local file system.

If I use the dash l option, which stands for long format, then it formats the listing similarly to the way the corresponding HDFS command formats it. In this listing, you can see the file orders.txt which I just downloaded here from HDFS. You can print the contents of this local file to the screen using the command, cat orders.txt. The output of this is identical to the output from the corresponding HDFS command.

So many of these HDFS shell commands are directly analogous to Linux shell commands. But there are also many differences. For example, the hdfs dfs dash get command has no direct analog. Also, when you use an ls command to list the contents of a directory in your local file system, if you do not specify a directory path, then it lists the contents of the current working directory. But on HDFS, there is no concept of a current working directory. So what happens if you run hdfs dfs dash ls and leave off the directory path? Is it lists the contents of your Home directory in HDFS.

Recall that on the VM, that's the directory/user/training. There's nothing in that directory at this time, so the command prints nothing. You should be aware that there is an alternate syntax for calling HDFS shell commands. Instead of hdfs dfs, you can use hadoop fs. These commands are synonymous. They do exactly the same thing. In this course, we'll be using hdfs dfs. So in this video, I showed a few ways to interact with HDFS from the command line using HDFS shell commands. Later in this course, you'll see how to use HDFS shell commands to perform other tasks like creating directories and copying data into HDFS.

But in the meantime, please do not run any HDFS shell commands besides the ones I demonstrated in this video, the ls, cat and get commands. Running some of the other commands could cause the VM to behave in unexpected ways. As I showed, you can run these HDFS shell commands on the VM for this course.

That's because the HDFS shell application is installed on the VM, and it's connected to the instance of HDFS that's running on the VM. But if you try to run these commands on your own computer outside of this VM, they probably will not work. You need to run these commands on a computer that's connected to a Hadoop Cluster.

Often, this will be a gateway node, also called an edge node. A gateway node is a computer that provides an interface between the Hadoop cluster and the outside network. A gateway node typically does have the HDFS shell application installed on it, and it usually has tools like Beeline and Impala shell installed also. So you can run Hive and Impala queries from the command line. So if you're using a real-world Hadoop cluster, ask your cluster administrator how you can access the command line on a gateway node.

---

## BROWSING FILES IN S3

### UNDERSTANDING S3 AND OTHER CLOUD STORAGE PLATFORMS

In addition to using HDFS for storage, you can store your data using cloud services such as Amazon web services, Microsoft Azure, or Google Cloud platform. Today, some companies store big data on-premises in HDFS, some store it in Cloud Storage, and some use a hybrid approach using both HDFS and cloud storage. The major reasons why companies use cloud storage are cost and scalability. Usually, it costs less to store some amount of data in cloud storage than it would to store it in HDFS.

As the amount of data you need to store grows larger and larger, it's often easier to pay incrementally larger amounts of money to a cloud storage provider than it is to purchase new hard disks and new servers and install them in a data center. Amazon has many cloud services but their storage service is called S3 which is short for Simple Storage Service. S3 is the most popular cloud storage platform, and it's the one you will use in this course when you're using something other than HDFS. Hybrid Impala can use S3 very much like they use HDFS. So most of the time when you're querying a table, you won't even notice if the data is in S3 or in HDFS. S3 organizes data into buckets. Buckets are like the folders at the top or highest level of a file system. Buckets in S3 must have globally unique names.

So if anyone else in the world is using a specific name for a bucket, you must pick a different name. Within a bucket, you can store files and folders. Technically, S3 stores all the files in your bucket in a flat file system, and it simulates folder structures by using slashes in the filenames. But that's not something you need to be concerned with for this course.

S3 is connected to the internet. The data you store in S3 can be accessed from anywhere. S3 provides ways to control who has access to the data though you can make it public or restrict access to certain users or networks. There is only one instance of S3 and it's operated by Amazon and runs across Amazon's data centers globally. HDFS, on the other hand, is a file system that exists on a Hadoop cluster. There are many incidences of HDFS. There's one on every Hadoop cluster.

Data stored in HDFS is generally not accessible from everywhere. Access is usually restricted to specific private networks. The major way that S3 is different from HDFS is that S3 provides storage and nothing more. S3 cannot process your data. It can only store it and provide it when requested. HDFS, on the other hand, typically stores files on the same computers that also provide processing power to your big data system. So if you're using HDFS to store files, then the files on HDFS reside on the same computers where data processing engines like Hive and Impala run.

When you run a query in Hive or Impala, if the data for the table you're querying is stored in HDFS, then Hive or Impala can routinely read that data off the hard disk on the computer where it's running. This is called data locality or just locality for short. The processing happens on the same location where the data is stored. If you store your data in cloud service like S3 and there is no data locality, Hive or Impala must fetch the data from S3 over the network before it can process it. This makes queries run a little bit slower, but nowadays the networks that connect data centers together are so fast that the difference is often insignificant. The readings in this course we'll show how to access S3 from the VM.

But I want to point out to you now that when you're working on the VM, HDFS is local because we stimulate an entire HDFS cluster within the single VM. So you do not need internet access while using HDFS in this course. If you using S3 though, you will need a network connection. You also won't be able to browse S3 files using the Hue file browser. It currently requires you to have right access to a

bucket if you want to browse it directly, and we are not able to provide write access to all Coursera learners. You have read access only to the S3 bucket you'll use for this course, but you can use Hue to work with Hive and Impala tables that use S3 for their storage.

---

## BROWSING S3 BUCKETS FROM THE COMMAND LINE

Although you can't use Hue to browse s3 buckets on the course VM, you can use the command line. In this demo, I'll show you a few commands you can use on the VM to do this. Open a terminal window as you did to browse HDFS from the command line. I'm going to stretch this window a bit so that my longer commands will appear on one line for easy reading. Most of the commands you used to browse HDFS will also work for S3.

If you specify the path as an s3 URL using the s3a protocol. For example, recall that you can use hdfs dfs-ls to list the contents of a directory in HDFS. You can use the same command to list the contents of an s3 bucket by specifying the path as s3a colon slash slash and then the bucket name. With this course, we created a bucket named training dash coursera one. So, to list the contents of this bucket, you would specify the path as s3a://training.coursera1, with a slash at the end. With S3 this trailing slash is required.

And the result shows that there are two directories in this S3 bucket named employees and jobs. To see the contents of the employees directory, I'll press the up arrow key to recall the previous command, and I'll add employees/ at the end of the path. The results shows that there is a file named employees.csv in this employees directory. To copy a file from an s3 bucket to the local file system, you can use hdfs dfs -get. And again, you must specify the path beginning s3a:// followed by the bucket name.

I'll copy the file employees.csv to the current working directory in the local file system on this VM using a dot to indicate that. You can run an ls command to see the file employees.csv here in the current working directory. I'll run an rm command to remove it from here. Of course this file still exists in the s3 bucket. I've just removed it from the local file system. To print the contents of a file in s3 to the screen, you can use hdfs dfs -cat with a qualified s3 path. These hdfs shell

commands work with s3 buckets because the Hadoop software includes a connector to s3, it's called s3a. In the real world, you might encounter some environments where there's an older version of the connector installed. In that case, you might need to use s3 or s3n. Instead of s3a but in most environments and on SVM, you'll use s3a.

To run these HDFS shell commands, you need the HDFS shell application to be installed on the computer you're using. That's true regardless of whether you are interacting with files in HDFS or in s3. But it can be impractical to install and configure the HDFS shell application. So I'll also teach you another way, to work with files in s3 from the command line. Amazon provides a tool called the AWS command line interface or the AWS CLI. This tool provides commands for interacting with s3 and some of Amazon's other AWS services.

In most cases, this tool is easier to install than the HDFS shell application. There are instructions for installing it at aws.amazon.com/cli. The AWS CLI is pre-installed on the course VM. And you can use it to do the same things I just showed how to do with HDFS shell commands. AWS CLI commands begin with AWS, and the commands for interacting with s3 begin with aws s3. After that, you specify a command to run. One of these commands is ls. To list the commands of an s3 bucket or directory in an s3 bucket.

After ls you specify an s3 URL, beginning with s3://. For example, to list the contents of the bucket named training-coursera1, you would run this command. A few things to notice, I did not use as dash before the ls. And I used s3, not as s3a or the protocol. Notice also, that the results are different in this case than they were when using the hdfs dfs command. Less information is given. The PRE or P-R-E values in the output indicate these items are directories. This is because s3 does not actually store directories, instead it simulates directory structures by prepending directory names to the beginning of file names separated by slashes.

For our purposes, you can understand these as directories. So, the output shows that this s3 bucket contains two directories named employees and jobs. To use the aws cli to copy or download a file from s3 to the local file system, use the command aws s3 cp, followed the source and destination. For example, for the source I'll specify the s3 URL or the file employees.csv, in the employees directory in this bucket. And for the destination, I'll specify a dot to indicate the current

working directory in the local file system. I'll run an ls command to show this file, employees.csv was copied here to the current working directory. The aws cli does not provide a separate command to print the contents of a file in s3 to the screen, but there is a trick you can use to do it. I'll press the up arrow key to recall the previous command, and instead of using a dot as the destination, I'll use a dash. This causes the contents of the file to be printed to the screen. When you use a dash as this destination, the file is not stored in the local file system.

# WEEK 2

Learning Objectives

- Create databases and tables, optionally specifying attributes such as storage location and file type
- Specify appropriate advanced attributes such as SerDes and TBLPROPERTIES when creating a table
- Examine and make changes to existing table schemas
- Describe the implications of making changes to table schemas and removing tables and databases
- Choose when to use Hive or Impala when managing tables
- Choose the appropriate command for refreshing Impala's metadata cache

**Introduction**
**Week 2 Introduction**

---

## CREATING DATABASES AND TABLES

In Hive and Impala, each table belongs to a specific database. Databases are helpful for organizing tables; by using multiple databases, you can have tables with the same name in different databases. In this way, they serve as ***namespaces***. Even if you don't use tables with the same name, separating databases is helpful for organization and for restricting user access to data they don't need to access, or should not access.

Creating databases and tables using Hue's Table Browser is fairly easy. As you go through this reading, or after you've read through it once, use the VM to create some test databases and tables. You can drop (delete) the test databases and tables when you're done.

**To perform the steps described in this reading, you will need to use Hue on the VM. If you do not already have the VM installed and running, please follow the instructions in the reading *Downloading and Installing the VM* in the first week of this course. Then open the web browser on the VM and click the link for Hue in the bookmarks toolbar.**

After this reading, the remainder of this week will be about using SQL commands to create databases and tables. After you've learned both methods, you can choose which method you prefer for a given task.

## Creating a New Database

You can start the process for creating a database by going directly into the Table Browser, or by using the data source panel. Try both, and see which you prefer.

When you create a database using these methods, Hive or Impala creates the directory **/user/hive/warehouse/*databasename*.db** (where ***databasename*** is the name you entered) unless a different location is specified. Tables created in a database will be placed as a subdirectory within this database directory.

## Using the Table Browser

Enter the Table Browser by clicking the hamburger menu (three horizontal lines) and choosing **Browsers > Tables**. The main (center) panel will show the **default** database for Hive and Impala. To add a new database:

1. Click **Databases** in the breadcrumbs at the top. (See Figure 1 below.)
2. Hover over the **+** symbol on the far right; it should say **Create a new database**. Click that symbol.
3. Give your database a name such as **test**, and (optionally) a description in the appropriate field.
4. ***Optional:*** If you want to store the database files in a location other than HDFS (in S3, for example), uncheck the **Default Location** box and supply the location.
5. Click **Submit**.
6. The Task History pop-up will appear; the top should say **Creating database *name*** with a green line underneath. (See Figure 2 below.) Click the **x** on the right to dismiss the window.
7. Verify the creation was successful by clicking **Databases** in the breadcrumbs, and note that your new database now appears in the list. If you like, you can check the HDFS file structure to see that **/user/hive/warehouse/*name*.db** exists.
8. To drop (delete) your test database, check the box next to it and click the **Drop** button above the list, then click **Yes**. This will drop the database

and delete the directory from HDFS. Be careful not to drop any databases that have tables in them!


Figure 1

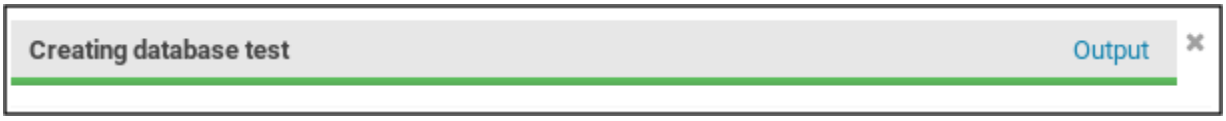



Figure 2

## USING THE DATA SOURCE PANEL

The data source panel appears with all areas of Hue, so you can easily use this method without the need to switch to the Table Browser first.

1. Be sure the data source panel is in the database mode (and not in the files mode). The database icon (stacked disks) should be blue. (See Figure 3 below.)
2. Navigate to the Impala source. (See Figure 4 below. You can also use Hive, and the database will be available for both engines.)
3. Hover over the + symbol; it should say **Create database**. Click that symbol. ***(Note: From this point on, the steps are the same as in the previous section.)***
4. In the main panel, give your database a name such as **test**, and (optionally) a description in the appropriate fields.
5. ***Optional:*** If you want to store the database files in a location other than HDFS (in S3, for example), uncheck the **Default Location** box and supply the location.
6. Click **Submit**. As in the previous section, the Task History pop-up will appear. Click the **x** to dismiss it.
7. You'll now be in the Table Browser, but in the **default** database. Verify the creation was successful by clicking **Databases** in the breadcrumbs, and note

that your new database now appears in the list. ***You can also navigate in the data source panel back to the Impala source and see the list of databases there.***

8. To drop your test database, check the box next to it and click the **Drop** button above the list, then click **Yes**.

**Figure 3**


**Figure 4**

## CREATING A NEW, EMPTY TABLE

When you create a table, Hive or Impala creates the directory **/user/hive/warehouse/*tablename*** or, if it's in a database other than **default**,**/user/hive/warehouse/*databasename*.db/*tablename***. This directory will be empty at first; you will learn in a later week of the course how to populate the table with data by loading data files into it.

You will learn how to drop these tables in a later lesson in this Week, so don't worry about dropping the examples you create now.

As with the database creation, you can start this process with the data source panel (Option A, below), or you can go directly to the Table Browser (Option B). Decide which you want to try from your experiences creating test databases:

- ***Option A:*** On the data source panel on the left, navigate to the database where you want to put the table. In this case, choose the **default** database in the Impala source. Hover over the **+** symbol; it should say **Create table**. (See Figure 5.) Click that symbol.


**Figure 5**

- ***Option B:*** Enter the Table Browser by clicking the hamburger menu (three horizontal lines) and choosing **Browsers > Tables**. The main (center) panel will show the **default** database. (You can switch to a different database if you like, but this isn't necessary for testing purposes. Stay in the **default** database.) Again, hover over the **+**symbol on the far right; it should say **Create a new table**. Click that symbol.

In the main panel, you have the opportunity to indicate a file in HDFS that will be the data for this table. This can be very handy, because Hue will try to guess the columns, including their data types, based on this file. You will see this later in this course, so for now, you'll create an empty table with no data.

1. Click the **Type** field and change it to **Manually**; then click the **Next** button.
2. In the **DESTINATION** area, name your table. For this example, use **default.test**, or just **test** if you are in the **default** database.
3. For **PROPERTIES**, you can set the file format and storage location, if desired. For now, leave the defaults (**Text** format and **Store in Default location** checked).
4. Under **FIELDS**, you can specify the columns for your table; click **+Add Field** for each column and specify the name and data type. Give your test table a couple of columns, such as **id** and **title**. You can leave the field type as **string** for each. You'll learn more about choosing data types later in this course.
5. Click **Submit**. The Task History pop-up will appear; the top should say **Creating table *name*** (for the test case, ***name*** is **default.test**) with a green line underneath. Click the **x** on the right to dismiss the window.

6. Verify the creation was successful by looking at the data source panel. Navigate to the database (Impala **default**) if necessary, and note that your new table now appears in the list. You might need to click the refresh button (two curved arrows) to refresh the display; choose **Clear cache** if you do this.
7. For your test, check the HDFS file structure to see that **/user/hive/warehouse/test/** exists. This directory will be empty, because the table has no data in it.

---

## CREATING A NEW TABLE TO QUERY EXISTING DATA

The steps above described how to create an ***empty***table, with no data in it. You can also use Hue to create a table to query data that already exists in HDFS.

For example, in the HDFS directory **/old/castles/**, there is a file named **castles.csv**. Review this file: The fields are separated by commas, the names of the columns (**name** and **country**) are provided in the header line, and both columns contain character string data. Using the steps below, you can create a table to query this data.

1. Using the data source panel (in the database mode) or the table browser, click the **+** symbol to create a new table.
2. Click the **Type** field and change it to **Manually**; then click the **Next** button.
3. In the **DESTINATION** area, name your table. For your example table, use **default.castles**.
4. For **PROPERTIES**, you can set the file format. The example file is a text file, so leave the default format (**Text**) checked.
5. Uncheck the **Store in Default location** checkbox. When you do this, you will see an **External location** field appear below it.
6. Enter the HDFS directory path (**/old/castles** for the example) into the **External location** field. Alternatively, you can click the **..** icon on the right side and use the dialog to select this folder. (See Figure 6 below.)
7. Directly below there, click the **Extras** icon, which looks like three sliders. You can add an optional description of the data location in the **Description** field (leave it blank for now). You can also specify the character that separates the fields here by checking the **Custom char delimiters** checkbox. Check the box and notice that three drop-down menus appear. You can specify different kinds of delimiters here. For this

example, you only need the **Field** drop-down menu. The field separator is already set to **Comma (,)**which is the field separator (delimiter) used in the file **castles.csv**, so keep this selection.

8. Under **FIELDS**, specify the columns for this table; click **+Add Field** for each column and specify the name and data type. Recall that the file **castles.csv**, which contains the data for the example table, has two columns: **name** and **country**. Add both (in that order), both as **string** types.
9. Click **Submit**. The Task History pop-up will appear; the top should say **Creating table *dbname.tablename*** with a green line underneath. Click the **x** on the right to dismiss the window.
10. Verify the creation was successful by looking at the data source panel. Navigate to the database (Impala **default**) if necessary, and note that the new table (**castles**) now appears in the list. You might need to click the refresh button (two curved arrows) to refresh the display; choose **Clear cache** if you do this.
11. Hover the cursor over the table name then click the **i** icon to the right of the table name. Click the **Sample** tab to verify that the data appears in the sample results.


**Figure 6**

## LIMITATIONS

As described above, Hue provides several options for creating databases and tables. However, these options have some limitations. For example, when creating a new table to query existing data stored in text files (as demonstrated above), Hue assumes that the data files have header rows. There is currently no way to specify in Hue that the data files do ***not*** have header rows.

To overcome these and other limitations of Hue, you can instead use SQL commands to create databases and tables. Furthermore, using SQL commands (instead of Hue user interface actions) is a more systematic way of performing tasks like this. SQL commands can be scripted, automated, and scheduled. By

saving your SQL commands in a file, you can effectively document the steps you performed, and you can make the steps reproducible.

In the next reading, you'll learn how to create databases and tables by running SQL statements.

## PERMISSIONS TO CREATE DATABASES AND TABLES

In the VM for this specialization, you have the ability to create databases and tables. This requires special permissions, and in a real-world environment, you might not have those permissions.

Administrators of a real-world environment can use a DCL (Data Control Language) command, **GRANT**, to set permissions to users based on roles, groups, or individuals. How this works depends on many factors, including what tools you're using and how they are configured. The details are beyond the scope of this course, so talk to your IT administrator if you have questions about what level of permissions you have in your work environment.

## THE CREATE TABLE STATEMENT

### Introduction to the CREATE TABLE Statement

The create table statement creates a new table and specifies its characteristics. When you execute a create table command, Hive or Impala adds the table to the metastore and creates a new subdirectory in the warehouse directory in HDFS to store the table data. The location of this new subdirectory depends on the database in which the table is created.

Tables created in a default database are stored in subdirectories directly under the warehouse directory. Tables created in other databases are stored in subdirectories under those database directories. The basic syntax of the create table statement, should be familiar to anyone who has created tables in a relational database. After create table, you optionally specify the database name. Then give the name of the new table, and a list of the columns, and their data

types. If you omit the database name, then the new table will be created in the current database. If you're using Q, recall that the current database is the one that's selected in the active database selector. If you're using the command line,the current database is the one you specified when you launch to be aligned or Impala shell, or the one you specified in the most recent use command in the session. If you did not specify a current database in one of those ways, then the current database is the default database.

The list of column names and data types is enclosed in parentheses, with each name type pair given as column name, space, data type. The name type pairs are separated by commas. You can optionally include line breaks after the commas, and spaces to indent the lines to break up the list across the multiple lines. This makes it more readable when there are many columns. Column names, and also table names should contain only alphanumeric characters and underscores. If any uppercase characters are used in column names or table names, they will be converted to lowercase.

Regarding data types, you'll learn more about those in the next week of this course. For this week don't worry about the specifics of the data types. By default Hive and Impala, create what are called managed or internally managed tables. When you drop a managed to table, the tables storage directory in the file system is deleted, and any data files within that directory are deleted. In some cases, you may want to avoid this behavior. You can do this by creating an unmanaged or externally managed table. To do that, use the keyword external as shown in the example here.

When you drop an externally managed table, the table metadata is removed from the metastore, but the table data remains in HDFS. An externally managed table in Hive and Impala is different from what's sometimes called an external table in relational database systems. So if you have some notion of what an external table is from the relational database world, set that aside and remember that with Hive and Impala, externally managed, just means that Hive and Impala will leave the data files in place if you drop the table. Hive has an option to create what's called a temporary table. This is a table that's visible only to you, and only in the current session. The table and any data stored in it are deleted at the end of your current session. You can create a temporary table with Hive by adding the keyword temporary, as shown in the example here. This can be useful if you're using a real-

world environment, and you want to test a create table statement before running it for real. However, temporary tables are not supported by Impala and they have some other limitations. So in this course we will not be using temporary tables. So that's the basic syntax of the create table statement for Hive and Impala. Again, if you've ever created tables in a relational database, it should look familiar to you. However there are three optional clauses that you can use in a create table statement.

They are unique to distributed SQL engines like Hive and Impala. These are: the row format clause, the stored as clause and the location clause. You'll learn about those clauses in the remainder of this lesson. You'll also learn a few other options for creating tables. You'll have a chance to practice creating tables in the VM, and you will see a full demonstration of a create table example that uses all these clauses.

---

## THE ROW FORMAT CLAUSE

As you're probably aware, data files can come in different formats. For example, a file might be ***comma-delimited,*** which means the comma (,) is used to mark ***(delimit)*** when one column's value ends and the next column's value begins. The tab character is often used instead, giving a ***tab-delimited*** file.

As part of the **CREATE TABLE** statement, you can specify how the data is delimited in its files. This is done using the **ROW FORMAT** clause. The syntax for this clause is:

```
ROW FORMAT DELIMITED

    FIELDS TERMINATED BY character
```

For example, consider this row from a data file:

```
1,Data Analyst,135000,2016-12-21 15:52:03
```

There are four fields here, separated by commas: an ID, a job title, a salary, and a timestamp when the data was recorded. The following statement will create the table:

```
CREATE TABLE jobs (id INT, title STRING, salary INT, posted TIMESTAMP)
    ROW FORMAT DELIMITED
    FIELDS TERMINATED BY ',';
```

The **ROW FORMAT DELIMITED** portion of this clause specifies ***that*** you are using a delimiter. You also need the **FIELDS TERMINATED BY** portion, to specify ***which*** delimiter you are using. In this case, the delimiter is the comma (**,**). If the delimiter were the tab character, the clause would use **\t** in quotes (because **t** is the ***escape sequence*** representing the tab character):

```
ROW FORMAT DELIMITED
    FIELDS TERMINATED BY '\t'
```

The **FIELDS TERMINATED BY** portion is a part of the **ROW FORMAT** clause. It ***must*** be preceded by **ROW FORMAT DELIMITED**. The line break and indentation used in these examples is optional.

If you omit the **ROW FORMAT** clause, Hive and Impala will use the default field delimiter, which is the ASCII Control+A character. This is a non-printing character, so when you attempt to view this character using a text editor or a **cat** command, it might render as a symbol (such as a rectangle with 0s and 1s in it) and other characters might overlap with it.

In the following exercises, you can see how the **ROW FORMAT** clause dictates storage of new data for a table, and how it tells Hive and Impala how to correctly read a table in existing data files.

On the VM:

1. Log in to Hue and go to the Impala query editor.

2. Do the following to create a comma-delimited table, fill it with one row of data, and look at the resulting file on HDFS:

   a. Execute the **CREATE TABLE** statement:

```
CREATE TABLE jobs
```

```
(id INT, title STRING, salary INT, posted TIMESTAMP)

ROW FORMAT DELIMITED

FIELDS TERMINATED BY ',';
```

b. Load one row of data by executing the following statement. (**Note:** This statement is ***not*** a good way to add a lot of data to a table in a big data system, for reasons described later in the course. Here, we're only adding one row for demonstration purposes.)

```
INSERT INTO jobs

VALUES (1,'Data Analyst',135000,'2016-12-21 15:52:03');
```

c. Use the File Browser or the data source panel on the left side (choosing the files icon rather than the database icon) and find the **/user/hive/warehouse/jobs** directory. If you don't see the **jobs** subdirectory, refresh the display by clicking the refresh button (two curved arrows). Find a file with a name that's just a string of letters and numbers, and click that file.

d. You can see the contents of the file in the main panel. Notice that you have a comma-delimited row of data.

e. Notice that the string and timestamp values stored in this file are ***not*** enclosed in quotes. Quotes were used in the **INSERT** statement to enclose the literal string and timestamp values, but Hive and Impala do ***not*** store these quotation marks in the table's data files.

3. Now go back to the Impala query editor and create a tab-delimited table, fill it with the same data, and compare the resulting file on HDFS to what you had for the comma-delimited file:

a. Execute the **CREATE TABLE** statement (the only differences are the table name and the delimiting character):

```
CREATE TABLE jobs_tsv

    (id INT, title STRING, salary INT, posted TIMESTAMP)
```

```
ROW FORMAT DELIMITED

  FIELDS TERMINATED BY '\t';
```

b. Load one row of data by executing the following statement. (**Note:** This statement is ***not*** a good way to add a lot of data to a table in a big data system, for reasons described later in the course. Here, we're only adding one row for demonstration purposes.)

```
INSERT INTO jobs_tsv

  VALUES (1,'Data Analyst',135000,'2016-12-21 15:52:03');
```

c. Use the File Browser or the data source panel on the left side (choosing the files icon rather than the database icon) and find the **/user/hive/warehouse/jobs_tsv** directory. If you don't see the **jobs_tsv** subdirectory, refresh the display by clicking the refresh button (two curved arrows). Find a file with a name that's just a string of letters and numbers, and click that file.

d. You can see the contents of the file in the main panel. Notice that this time, you have a ***tab***-delimited row of data.

e. Notice that the string and timestamp values stored in this file are ***not*** enclosed in quotes. Quotes were used in the **INSERT** statement to enclose the literal string and timestamp values, but Hive and Impala do ***not*** store these quotation marks in the table's data files.

4. Drop the **jobs** and **jobs_tsv** tables. (Look back at the "Creating Databases and Tables…" readings for how to drop tables, if necessary.)

5. When you're creating a table for existing files, you'll want to specify how the file is already delimited. Do the following steps to see this at work.

a. First, examine the data in the **/user/hive/warehouse/investors** directory. This will be the default location for a table named **default.investors**. There is no table for this data yet, so you will create one. Notice that the file is ***comma-delimited***.

b. Create an ***externally managed*** **investors** table using the following statement (which purposefully does ***not*** specify the delimiter). ***It's important to use the EXTERNAL keyword,*** so you can drop the table without deleting the data.

```
CREATE EXTERNAL TABLE default.investors
    (name STRING, amount INT, share DECIMAL(4,3));
```

c. Use Hue to look at the table. It only has a few rows, so you can use **SELECT * FROM investors;** in the query editor, or you can use the data source panel to view the sample data. Notice that the entire row ended up in the **name** column! This is because the command used the default delimiter, Control+A, rather than the comma.

d. Drop the table by entering and executing the command **DROP TABLE investors;** Check that the **/user/hive/warehouse/investors** directory still exists and has a file in it. (If you made a mistake and the directory is gone, see below!)

e. Now, create another **investors** table using the **ROW FORMAT** clause to specify the delimiter:

```
CREATE EXTERNAL TABLE default.investors
    (name STRING, amount INT, share DECIMAL(4,3))
  ROW FORMAT DELIMITED
    FIELDS TERMINATED BY ',';
```

f. Use Hue to look at the table. Now each column should have values. Keep this table, you will use it again later.

*If you accidentally deleted your data:*

Open a Terminal window. (You can do this by clicking the icon in the menu bar that looks like a computer.) Enter and run the following command (on one line), which will copy the file from your local disk to the proper place in HDFS. Do ***not*** include the **$**; that's the prompt to indicate this is a command-line shell command.

```
$ hdfs dfs -put
~/training_materials/analyst/scripts/static_data/default/investors  /user/hive/
warehouse/
```

---

## THE STORED AS CLAUSE

The data for each table is stored in files. The **ROW FORMAT** clause specifies the delimiters between column values in those files, but there are different file formats you can use. (You will learn more about these file formats in Week 3 of this course.) The **STORED AS** clause in the **CREATE TABLE** statement allows you to specify what file format you want a new table to use. To create a table that uses existing data files, you need to match the format of the file.

The syntax for this clause is simply:

```
STORED AS filetype
```

For example, a table creation statement might look like this:

```
CREATE TABLE jobs (id INT, title STRING, salary INT, posted TIMESTAMP)

  STORED AS TEXTFILE;
```

The default file format is **TEXTFILE**—simple text format, which is human readable—so in this example, the **STORED AS** clause is optional. Without it, Hive or Impala would still use **TEXTFILE** for the file format. However, other file formats (which will often look like gibberish if you try to view the files directly) must be specified explicitly.

On the VM:

1. Log in to Hue and go to the Impala query editor.

2. Do the following to create a table, fill it with one row of data, and look at the resulting file on HDFS:

a. Execute the following **CREATE TABLE** statement:

```
CREATE TABLE jobs_txt
    (id INT, title STRING, salary INT, posted TIMESTAMP)
  STORED AS TEXTFILE;
```

b. Load one row of data by executing the following statement.

```
INSERT INTO jobs_txt
  VALUES (1,'Data Analyst',135000,'2016-12-21 15:52:03');
```

c. Use the File Browser or the data source panel (choosing the files icon rather than the database icon) and find the **/user/hive/warehouse/jobs_txt** directory. If you don't see the **jobs_txt** subdirectory, refresh the display by clicking the refresh button (two curved arrows). Find a file with a name that's just a string of letters and numbers, and click that file.

d. You can see the contents of the file in the main panel. Notice that you can clearly see each of the values you added to the table.

3. Now create another table using a different format, and see that the resulting file looks different:

a. Execute the following **CREATE TABLE** statement, which configures the table to store data in **PARQUET** format:

```
CREATE TABLE jobs_parquet
    (id INT, title STRING, salary INT, posted TIMESTAMP)
  STORED AS PARQUET;
```

b. Load one row of data by executing the following statement.

```
INSERT INTO jobs_parquet
  VALUES (1,'Data Analyst',135000,'2016-12-21 15:52:03');
```

c. Use the File Browser or the data source panel (choosing the files icon rather than the database icon) and find the **/user/hive/warehouse/jobs_parquet** directory. If you don't see the **jobs_parquet** subdirectory, refresh the display by clicking the refresh button (two curved arrows). Find a file with a name that's just a string of letters and numbers, and click that file. You'll get an error message that says Hue can't read the file.

d. Open a Terminal window. (You can do this by clicking the icon in the menu bar that looks like a computer.) Enter and run the following command, which will show the contents of the Parquet file. (Do not include the **$**; that's the prompt to indicate this is a command-line shell command, not a query.) Notice that the output includes a lot of non-ASCII characters, so you can't really read most of it.

```
$ hdfs dfs -cat /user/hive/warehouse/jobs_parquet/*
```

4. Drop both tables (**jobs_txt** and **jobs_parquet**) since you won't need either again.

5. Now try creating a table using data from an existing Parquet file. When you're done, keep this table, because you'll use it again later. (If you use the **EXTERNAL** keyword as directed below, then dropping the table will not delete the data, so you could drop it now and come back and recreate the table later if you wish.)

a. A Parquet version of the **investors** data is also stored in HDFS, at **/user/hive/warehouse/investors_parquet** (which will be the default location for a table named **default.investors_parquet**). Examine the file in the same way as you examined the jobs Parquet file: In the Terminal window, issue the command **hdfs dfs -cat /user/hive/warehouse/investors_parquet/investors.parq**. Again you'll see it's not really in human-readable format.

b. Now create the table from the query editor:

```
CREATE EXTERNAL TABLE default.investors_parquet
    (name STRING, amount INT, share DECIMAL(4,3))
  STORED AS PARQUET;
```

c. Use the data source panel or run a **SELECT *** query to verify that the contents of the new table are correct. (It should look identical to the other **investors** table you created in "The ROW FORMAT Clause" reading.)

---

## THE LOCATION CLAUSE

Unless otherwise specified, Hive and Impala store table data under the warehouse directory, which is by default the HDFS directory **/user/hive/warehouse/**. However, this may not be where you want to store some table data. For example, the data may already exist elsewhere in HDFS, and you may not have permission to move or copy it. If there's a lot of data, copying it into the warehouse directory would be inefficient compared to querying it in its current location. Even if there's not a lot of data, moving a copy to the warehouse directory means that the copy will be outdated as soon as changes are made to the original version.

The **LOCATION** clause allows you to create a table whose data is stored at a specified directory, which can be outside the warehouse directory. The syntax is simply:

```
LOCATION 'path/to/location/'
```

The example shown below creates a new table named **jobs_training** whose data resides in the HDFS directory **/user/training/jobs/**. The specified storage directory may already exist, but if it does not, Hive or Impala will create it.

```
CREATE TABLE jobs_training

    (id INT, title STRING, salary INT, posted TIMESTAMP)

  LOCATION '/user/training/jobs_training/';
```

To specify a storage directory outside of HDFS, you will need a fully qualified path in the **LOCATION** clause, including a protocol at the beginning of the path. For example, to specify a storage directory in Amazon S3, you will typically need to use **LOCATION 's3a://*bucket/folder/***'.

## DO NOT CONFUSE THIS WITH THE EXTERNAL KEYWORD

In practice, the **EXTERNAL** keyword is often used in conjunction with the **LOCATION** clause to specify a storage directory outside the warehouse directory. But when you use the keyword **EXTERNAL**, that does ***not*** necessarily mean that the data is stored outside the warehouse directory. In fact, if you use the keyword **EXTERNAL**, but you do not specify a directory with the **LOCATION** clause, then Hive and Impala will store the table data under the warehouse directory. On the other hand, if you use the **LOCATION** clause without the keyword **EXTERNAL**, dropping the table could delete the data, even if it's stored outside the warehouse directory.

So think of **EXTERNAL** as meaning externally ***managed*** (that is, managed by some software other than Hive or Impala) and ***not*** meaning externally ***stored***. Remember that the **LOCATION** clause, not the **EXTERNAL** keyword, determines where the table data is stored.

1. If you haven't already, use the Hue File Browser or the data source panel to examine the **/user/hive/warehouse** directory. (Using the data source panel, you need to click the files icon rather than the database icon.) You should see directories for databases (with **.db** at the end) and for any test tables that you've created in the **default** database and didn't drop. (If you haven't created any tables yet, you should still see some directories corresponding to tables in the **default** database, such as **customers** and **employees**.) Look inside one of these table directories and note whether there is a file inside. If the table has data, it will have at least one file here.

Next, do the following to create and examine a table whose data does ***not*** go into that default directory.

2. Go to the Impala query editor, select **default** as the active database, and execute the following statement:

```
CREATE TABLE jobs_training
    (id INT, title STRING, salary INT, posted TIMESTAMP)
LOCATION '/user/training/jobs_training/';
```

3. Use the data source panel (so you can leave the query editor in the main panel) to find the table directory for this new table. Note that it's ***not*** in **/user/hive/warehouse/** as the other tables in the **default** database are. It should be under **/user/training/** instead. Check that the new **jobs_training** directory is empty, since the table was created without data and none has been inserted.

4. Insert some data into the new table using the following statement in the query editor:

```
INSERT INTO jobs_training

    VALUES (1,'Data Analyst',135000,'2016-12-21 15:52:03');
```

5. Look again inside the **jobs_training** directory and verify that a new file has been added. You might need to click the refresh button to refresh the display. You can take a look at the file to see that it's the data you just inserted.

6. Drop the table.

Now create a table to query some existing data that's located in an S3 bucket. We've created a bucket and given read-only access to the VM. The same row of data you have been using in these tests is saved in a delimited text file in the S3 bucket named **training-coursera**, in a folder named **jobs**. The file uses the default delimiter (Control+A) so you do not need a **ROW FORMAT** clause.

7. Execute the following statement:

```
CREATE EXTERNAL TABLE jobs_s3

      (id INT, title STRING, salary INT, posted TIMESTAMP)

    LOCATION 's3a://training-coursera1/jobs/';
```

8. Verify that the table has one row of data, either using the data source panel or by executing a **SELECT *** query.

9. Drop the table. ***Note:*** Typically you need to use the **EXTERNAL** keyword to ensure you don't accidentally delete data when you drop the table. In this case, because you don't have write access to the bucket, you will not delete the data

even if you forgot the **EXTERNAL** keyword. Still, it's a good practice to use **EXTERNAL** for data that other people might be relying on, too, so you don't risk destroying their work as well as your own!

## CREATE TABLE SHORTCUTS

Following are two tips for creating databases and tables.

### USING IF NOT EXISTS

If you try to create a database or table using a name for one that already exists, Hive or Impala will throw an error. To avoid this, you can add the keywords **IF NOT EXISTS** to your **CREATE DATABASE** or **CREATE TABLE** statement. The syntax is as follows:

```
CREATE DATABASE IF NOT EXISTS database_name;

CREATE TABLE IF NOT EXISTS table_name;
```

When you use **IF NOT EXISTS**, then Hive or Impala will ***not*** throw an error if that name is already in use. They will instead do nothing.

This is particularly helpful if you're using a script to create a database or table. If the script is run multiple times, using **IF NOT EXISTS** will let the script create the database or table on the first run, and it will also complete without error on subsequent runs.

You can also use **IF NOT EXISTS** together with the **EXTERNAL** keyword, and together with the other optional **CREATE TABLE** clauses described in this week of the course.

## CLONING A TABLE WITH LIKE

If you need a new table defined with exactly the same structure as an existing table, then Hive and Impala make it very easy to create the new table. This is called ***cloning*** a table, and it's done using the **LIKE** clause. The new table will have the same column definitions and other properties as the existing table, but no data. The syntax is

**CREATE TABLE *new_table_name* LIKE *existing_table_name*;**

The example shown below creates a new empty table named **jobs_archived** with the same structure and properties as the existing table named **jobs**.

**CREATE TABLE jobs_archived LIKE jobs;**

It is possible to specify a few of the table properties for the new table by including the appropriate clauses in the **CREATE TABLE … LIKE** statement. Of the clauses covered in this course, currently only the **LOCATION** and **STORED AS** clauses can be used. If you need to change other properties, use **ALTER TABLE** after creating the table to set those properties.

Try how things work if you create a database using **IF NOT EXISTS**. You'll create a database named **dig**; do not drop this one, you'll use it in later lessons in this course.

1. First, create a database named **dig** by executing the command:

**CREATE DATABASE IF NOT EXISTS dig;**

Verify that you now have a database named **dig**.

2. Now try creating the database again using **CREATE DATABASE dig;** (without the **IF NOT EXISTS** phrase). What happens?

3. Now try the original command again, and notice how the response is different:

**CREATE DATABASE IF NOT EXISTS dig;**

Now try cloning.

4. Clone one of the tables in the **default** database using the **LIKE** clause—any of the tables will do, just be sure to use a slightly different name for the new table. (For example, you might clone the **customers** table in the default database and name it **customers_clone**.)

5. Verify that the structure (column names with their data types) is the same as the table you cloned. Also verify that the new table has no data in it, and the original table still has its data.

6. Drop the cloned table.

If this topic of creating Hive and Impala tables is new to you, you might not immediately understand some of the implications around it. One implication that's especially tricky to grasp is how the loose coupling of table definitions and the underlying data makes Hive and Impala radically different form traditional relational database systems. A good way to highlight this radical difference is to demonstrate that you can create two or more tables. That query the same underline data files.

I will demonstrate that now. In this example, I will use a data set stored in a text file in S3, it's in the bucket named training dash Coursera one, in a subdirectory named months. I'll the command, hdfs dfs -ls s3a://training-coursera1/months/ to list the files in that directory. The result shows that there is just one file there named months.txt. I'll run an hdfs dfs -cat command to print the contents of that file. This data describes the 12 months of the year and it looks like each record represents three values. First, the number of the month, second the three letter abbreviated name and third, the number of days in that month. But this file does not use delimiters or field separators in the usual way. It has a greater than sign, separating the first and second fields. And a comma, separating the second and third fields, so I would like to create a table to query this data but I'm not sure which delimiter to specify.

So, to demonstrate that it is possible to create more than one table on top of the same data files, I will create two tables, one using the greater than sign as the delimiter and the other using the comma. In the Impala query editor in Hue, I'll run a create table statement to create the first of these. The data is being managed externally, so I'll use the external keyword, CREATE EXTERNAL TABLE. I'll

name it months_a. For this first one, I'm going to use the greater than sign as the delimiter. So the columns will be the number of the month, which is an integer, that's what's on the left side of the greater than sign and on the right side there's a concatenation of name and days, which we can represent in a string. To specify the delimiter, I'll use Row, Format, Delimited. Fields Terminated by ">". The data is stored in a text file, so I'll specify, stored as text file.

Since text file is the default file format, I don't need to specify this, but I'll include it. And finally, I'll include the location clause to specify the directory in S3, where the data is stored. I'll run this statement to create the table. Now I'll modify this statement to create the second table. I'll make the table name months_b. For this table I'm going to use a comma as the delimiter. So the columns will be a concatenation of the month number and the three letter month name, which we can represent in a string. That's what's to the left of the comma, and on the right is the number of days in the month, which is an integer. In the row format clause, I'll change the delimiter to a comma and I'll leave everything else as it is. I'll run this statement to create the second table. Now we have two tables named months_a and months_b, which query the same underlying data files but which have different schemas. So when you query these two tables, you get different result columns, even though the data behind the two tables is identical.

The column with the concatenated together values is not very useful in this form. But you could use Hive or Impalas' built in string functions to extract the parts from it. For example, in Impala, you could run a query that uses the split_part function to return the two parts of each name and days value, on the left and right sides of the comma, as two separate columns named name and days. The function split_part is not available in Hive, but there are two similar built-in functions in Hive named split and substring_index. You could use one of those instead. I'll drop these two tables, we won't need them anymore. Here's an example of a different situation in which it might be useful to create tables with different schemas on the same data. In the training-coursera2 bucket in S3, there is a file named company_email.txt under the directory company_email. I'll run an hdfs dfs- cat command to show the contents of this file.

As you can see, this is a comma separated text file. It has three fields, representing an ID, name and email address. But look at the second and third lines. They contain quote characters and, even worse, the third line includes a

comma, the field separator, as a part of the second field. In the Impala query editor in Hue, I'll write and run a CREATE TABLE statement that creates a table named company_email. I'll treat this like a normal comma separated test file with three columns. Then I'll query this table. But look what happens. Those quotation marks appear in the results, and Impala splits the third record on the comma inside the quoted company name. So, the email address value in this third row is missing. And instead, it shows the part of the quoted company name that came after the comma.

So, it's clear that specifying a comma as the field separator is not going to work. One alternative is to create a table with just one column. A string column. Instead of attempting to split up each line of the file into separate columns, Hive or Impala will just return the whole line as a string value. I'll modify the previous CREATE TABLE statement to create a second table that works this way. I'll name it company_email_raw and I'll specify just a single column, named line of type STRING. For the field separator, I need to choose some character that does not occur anywhere in the data. That way, Hive and Impala will not find any instances of the field separator, so they will not split up the lines into separate fields. One character that does not occur anywhere in this table's data file is Hive and Impala's default field separator, the ASCII control A character.

To use this default field separator for this table, I can simply remove the whole row format clause. I'll run the statement to create the table, then I'll query this table. And as you can see, the result has just a single column, a character string column named line. Each result row consists of a single field. This table schema is not really useful but you can work with it. You can use the built in character string functions to parse each row value and break it up into the fields you're looking for. For example, the function regex_extract allows you to use regular expression matching to extract text from the field. Regular expressions are extremely powerful, but getting the right pattern can be tricky. If you're not familiar with regular expressions, this might be a little hard to follow. That's okay. Don't worry about the details.

Here I'll query the table and have it extract the three fields. First, the regular expression looks at the start of the line for any number of digits up to a comma and extracts just the digits. That's the ID field. Next is the second field. The company name. Here, if there are quotes in the line, the regular expression will

capture everything after the first quote and up until the second one. If there are no quotes then it will skip the first digits and the first comma, but then capture everything until it reaches another comma. That yields the name field. Finally, for the third field, the regular expression will skip over the ID and name fields and capture whatever is left. That's the e-mail field.

In the from clause I'll put the name of this table, company_e-mail_raw. Then, when I run this query, I get the result I was looking for. Again, don't worry about the details of these expressions. The point is that even when the data files are formatted in a way that Hive and Impala cannot easily handle, there are work arounds like this, that enable you to get the results you need. Later in this course, you'll learn about several possibilities for what you could do next. You could store the results of this query in another table, you could use this query to create what's called a view, or you could use an alternative approach to avoid writing complex regular expressions like this in the first place. You'll learn about that alternative approach using what are called SerDes in the next lesson. But for now I will drop these two tables.

We won't need them anymore. The examples I showed in this video all used data files stored in S3. But these files could have been in HDFS. This technique of multiple tables on one set of data files works regardless of the file system. Whenever you create more than one table on the same underlying data, you should use the external keyword to make these tables externally managed. That way, you can drop the tables without losing the data. You might, in some cases, have one table that's internally managed, then have any other tables that query the same underlying data be externally managed. The important thing is just to avoid the possibility of inadvertently deleting the data when you drop the table.

## USING DIFFERENT SCHEMAS ON THE SAME DATA

If this topic of creating Hive and Impala tables is new to you, you might not immediately understand some of the implications around it. One implication that's especially tricky to grasp is how the loose coupling of table definitions and the underlying data makes Hive and Impala radically different form traditional relational database systems. A good way to highlight this radical difference is to demonstrate that you can create two or more tables. That query the same underline data files.

I will demonstrate that now. In this example, I will use a data set stored in a text file in S3, it's in the bucket named training dash Coursera one, in a subdirectory named months. I'll the command, hdfs dfs -ls s3a://training-coursera1/months/ to list the files in that directory. The result shows that there is just one file there named months.txt. I'll run an hdfs dfs -cat command to print the contents of that file. This data describes the 12 months of the year and it looks like each record represents three values. First, the number of the month, second the three letter abbreviated name and third, the number of days in that month. But this file does not use delimiters or field separators in the usual way. It has a greater than sign, separating the first and second fields.

And a comma, separating the second and third fields, so I would like to create a table to query this data but I'm not sure which delimiter to specify. So, to demonstrate that it is possible to create more than one table on top of the same data files, I will create two tables, one using the greater than sign as the delimiter and the other using the comma. In the Impala query editor in Hue, I'll run a create table statement to create the first of these. The data is being managed externally, so I'll use the external keyword, CREATE EXTERNAL TABLE. I'll name it months_a. For this first one, I'm going to use the greater than sign as the delimiter. So the columns will be the number of the month, which is an integer, that's what's on the left side of the greater than sign and on the right side there's a concatenation of name and days, which we can represent in a string.

To specify the delimiter, I'll use Row, Format, Delimited. Fields Terminated by ">". The data is stored in a text file, so I'll specify, stored as text file. Since text file is the default file format, I don't need to specify this, but I'll include it. And finally, I'll include the location clause to specify the directory in S3, where the data is stored. I'll run this statement to create the table. Now I'll modify this statement to create the second table. I'll make the table name months_b. For this table I'm going to use a comma as the delimiter. So the columns will be a concatenation of the month number and the three letter month name, which we can represent in a string. That's what's to the left of the comma, and on the right is the number of days in the month, which is an integer.

In the row format clause, I'll change the delimiter to a comma and I'll leave everything else as it is. I'll run this statement to create the second table. Now we

have two tables named months_a and months_b, which query the same underlying data files but which have different schemas. So when you query these two tables, you get different result columns, even though the data behind the two tables is identical. The column with the concatenated together values is not very useful in this form. But you could use Hive or Impalas' built in string functions to extract the parts from it. For example, in Impala, you could run a query that uses the split_part function to return the two parts of each name and days value, on the left and right sides of the comma, as two separate columns named name and days.

The function split_part is not available in Hive, but there are two similar built-in functions in Hive named split and substring_index. You could use one of those instead. I'll drop these two tables, we won't need them anymore. Here's an example of a different situation in which it might be useful to create tables with different schemas on the same data. In the training-coursera2 bucket in S3, there is a file named company_email.txt under the directory company_email. I'll run an hdfs dfs- cat command to show the contents of this file. As you can see, this is a comma separated text file. It has three fields, representing an ID, name and email address. But look at the second and third lines.

They contain quote characters and, even worse, the third line includes a comma, the field separator, as a part of the second field. In the Impala query editor in Hue, I'll write and run a CREATE TABLE statement that creates a table named company_email. I'll treat this like a normal comma separated test file with three columns. Then I'll query this table. But look what happens. Those quotation marks appear in the results, and Impala splits the third record on the comma inside the quoted company name. So, the email address value in this third row is missing. And instead, it shows the part of the quoted company name that came after the comma. So, it's clear that specifying a comma as the field separator is not going to work. One alternative is to create a table with just one column. A string column. Instead of attempting to split up each line of the file into separate columns, Hive or Impala will just return the whole line as a string value. I'll modify the previous CREATE TABLE statement to create a second table that works this way.

I'll name it company_email_raw and I'll specify just a single column, named line of type STRING. For the field separator, I need to choose some character that does not occur anywhere in the data. That way, Hive and Impala will not find any

instances of the field separator, so they will not split up the lines into separate fields. One character that does not occur anywhere in this table's data file is Hive and Impala's default field separator, the ASCII control A character. To use this default field separator for this table, I can simply remove the whole row format clause. I'll run the statement to create the table, then I'll query this table.

And as you can see, the result has just a single column, a character string column named line. Each result row consists of a single field. This table schema is not really useful but you can work with it. You can use the built in character string functions to parse each row value and break it up into the fields you're looking for. For example, the function regex_extract allows you to use regular expression matching to extract text from the field. Regular expressions are extremely powerful, but getting the right pattern can be tricky. If you're not familiar with regular expressions, this might be a little hard to follow. That's okay. Don't worry about the details. Here I'll query the table and have it extract the three fields. First, the regular expression looks at the start of the line for any number of digits up to a comma and extracts just the digits.

That's the ID field. Next is the second field. The company name. Here, if there are quotes in the line, the regular expression will capture everything after the first quote and up until the second one. If there are no quotes then it will skip the first digits and the first comma, but then capture everything until it reaches another comma. That yields the name field. Finally, for the third field, the regular expression will skip over the ID and name fields and capture whatever is left. That's the e-mail field. In the from clause I'll put the name of this table, company_e-mail_raw. Then, when I run this query, I get the result I was looking for. Again, don't worry about the details of these expressions. The point is that even when the data files are formatted in a way that Hive and Impala cannot easily handle, there are work arounds like this, that enable you to get the results you need. Later in this course, you'll learn about several possibilities for what you could do next. You could store the results of this query in another table, you could use this query to create what's called a view, or you could use an alternative approach to avoid writing complex regular expressions like this in the first place.

You'll learn about that alternative approach using what are called SerDes in the next lesson. But for now I will drop these two tables. We won't need them anymore. The examples I showed in this video all used data files stored in S3. But

these files could have been in HDFS. This technique of multiple tables on one set of data files works regardless of the file system. Whenever you create more than one table on the same underlying data, you should use the external keyword to make these tables externally managed. That way, you can drop the tables without losing the data. You might, in some cases, have one table that's internally managed, then have any other tables that query the same underlying data be externally managed. The important thing is just to avoid the possibility of inadvertently deleting the data when you drop the table.

## ADVANCED CREATE TABLE TECHNIQUES

In the previous lesson, you learned about the structure of the CREATE TABLES statement and about three important clauses that you can use to specify how and where the table data is stored. The ROW FORMAT clause, the STORED AS clause and the LOCATION clause. These three clauses are optional, and so is the EXTERNAL keyword. But you'll use them often to override Hive and Impala's default behaviors when creating a new table. There is another optional clause that you probably won't use as often, but you should still know about it. It's the TBLPROPERTIES clause. The keyword that begin this clause is spelled T-B-L, PROPERTIES, but I'll pronounce it as table property. This clause allows you to set some special properties for the table you're creating. For example, if the data for the table is in files that include a column header in the first line, then you can set the TBLEPROPERTIES('skip.header.line.count'='- 1'), as shown here. So we'll skip the first line. Be warned, though, that this will skip the first line in every file in the table's data directory, not just in the first file. So if your data is in multiple files, as it often is with big data, then you need every file to have the header line in it. Otherwise, Hive and Impala will not read in the data in the first line of those other files. Also, some systems, such as Apache Spark, will ignore this skip.header.line.count property when querying a table. If you want to learn more about this, you can read the entry in the Apache Spark issue tracking system that describes it. See that in the links for this video. So you can specify TBLPROPERTIES in a CREATE TABLE statement, as shown here. You can now also specify TBLPROPERTIES in an ALTER TABLE command. You'll learn more about that and about other situations when you might need to use table properties in later lessons. For the remainder of this lesson, you'll learn about a different and more advanced way to use the ROW FORMAT clause.

## SPECIFYING TBLPROPERTIES

In the previous lesson, you learned about the structure of the CREATE TABLES statement and about three important clauses that you can use to specify how and where the table data is stored.

The ROW FORMAT clause, the STORED AS clause and the LOCATION clause.

These three clauses are optional, and so is the EXTERNAL keyword. But you'll use them often to override Hive and Impala's default behaviors when creating a new table. There is another optional clause that you probably won't use as often, but you should still know about it. It's the TBLPROPERTIES clause.

The keyword that begin this clause is spelled T-B-L, PROPERTIES, but I'll pronounce it as table property. This clause allows you to set some special properties for the table you're creating. For example, if the data for the table is in files that include a column header in the first line, then you can set the TBLEPROPERTIES('skip.header.line.count'='- 1'), as shown here.

So we'll skip the first line. Be warned, though, that this will skip the first line in every file in the table's data directory, not just in the first file. So if your data is in multiple files, as it often is with big data, then you need every file to have the header line in it. Otherwise, Hive and Impala will not read in the data in the first line of those other files. Also, some systems, such as Apache Spark, will ignore this skip.header.line.count property when querying a table.

If you want to learn more about this, you can read the entry in the Apache Spark issue tracking system that describes it. See that in the links for this video. So you can specify TBLPROPERTIES in a CREATE TABLE statement, as shown here. You can now also specify TBLPROPERTIES in an ALTER TABLE command. You'll learn more about that and about other situations when you might need to use table properties in later lessons. For the remainder of this lesson, you'll learn about a different and more advanced way to use the ROW FORMAT clause.

# USING HIVE SERDES

Traditional data processing relies on data being stored in structured tables with well-defined rows and columns. However, data is often not structured this way. A lot of data exists in ***unstructured*** or ***semi-structured*** text files, such as log files, free-form notes in electronic medical records, different types of electronic messages, and product reviews. This kind of data can provide valuable insights, but working with it requires a different approach.

Hive provides features for working with unstructured text data, semi-structured data in formats like JSON, and data that lacks consistent delimiters. Note that the features discussed in this reading are supported only by Hive, not by Impala.

Recall that the clause **ROW FORMAT DELIMITED** is used when creating a table with data stored in text files. When you create a table using this clause, the data in the underlying text files must be organized into rows and columns with consistent delimiters.

---

## HIVE SERDES

Hive provides interfaces called ***SerDes*** that can read and write data that is ***not*** in a structured tabular format. SerDe stands for ***serializer/deserializer***. ***Serializing*** is the process for converting data to bytes so it can be stored; ***deserializing*** is the reverse process—decoding when reading the stored file.

You can specify a table's SerDe when you create the table. Actually, every table in Hive has a SerDe associated with it, whether you realize it or not. Typically, the SerDe is automatically set to the default for a given file format, based on the **ROW FORMAT** and **STORED AS** clauses.

For a table stored as text files, the default SerDe is called **LazySimpleSerDe**. The **LazySimpleSerDe** gets its name because it uses a technique called ***lazy initialization*** for better performance. That means it does not instantiate objects until they're needed.

In addition to **LazySimpleSerDe**, Hive includes several other built-in SerDes for working with data in text files. If you want to use one of these, you must explicitly specify the SerDe in the **CREATE TABLE** statement. The statement includes the clause **ROW FORMAT SERDE** followed by the fully qualified name of the Java class that implements the SerDe, enclosed in quotes.

For example, Hive includes the **OpenCSVSerde**, which can process data in comma-separated values (CSV) files. But why does Hive need a special CSV SerDe when the default SerDe can handle comma-delimited text files (when you specify **ROW FORMAT DELIMITED FIELDS TERMINATED BY ','**)?

Although Hive's default SerDe can work with simple comma-delimited text data, the actual CSV format allows things like commas embedded within fields, quotes enclosing fields, and missing fields. Hive's default SerDe does not support these, but the **OpenCSVSerde** does. The **OpenCSVSerde** also supports other delimiters such as the tab and pipe characters.

The Try It! section of "The ROW FORMAT Clause" reading presented an example where you incorrectly create a table from a comma-delimited file, because the delimiter was not specified. You corrected this by using a **ROW FORMAT DELIMITED** clause, but you also could have corrected it using the **OpenCSVSerde**:

```
CREATE EXTERNAL TABLE default.investors

    (name STRING, amount INT, share DECIMAL(4,3))

ROW FORMAT SERDE 'org.apache.hadoop.hive.serde2.OpenCSVSerde';
```

Two other examples are **RegexSerDe**, which identifies individual fields within each record based on a regular expression, and **JsonSerDe**, which processes data in JSON format. See the next reading for more on these SerDes.

In addition to using SerDes to read and write text data, Hive also uses SerDes to read to and write from binary and columnar formats like Avro and Parquet, but Hive does this automatically and hides the details from the user. Note, however, that the SerDe is not the same as the file type, but there is a close connection—file types require particular SerDes so Hive can read and write those file types. You'll learn more about file types in the next Week's materials. For now, just remember that SerDes define ***processes*** for reading data files.

While **OpenCSVSerde** allows you to both read and write files using the specified formatting, some SerDes will only read files; you cannot use them to write. You should test your SerDes or read the documentation to determine if writing files is supported.

Do the following to create a table named **tunnels** in the **dig** database.

1. Examine the data in the file **training_materials/analyst/data/tunnels.csv** on the local file system of the VM. Note that it's a comma-delimited text file.

2. In the ***Hive*** query editor, execute the following statement. Note that it uses the **OpenCSVSerde** rather than the **ROW FORMAT DELIMITED** syntax you used in the previous lesson. Also you ***must*** use Hive, not Impala.

```
CREATE TABLE dig.tunnels

    (terminus_1 STRING, terminus_2 STRING, distance SMALLINT)

ROW FORMAT SERDE 'org.apache.hadoop.hive.serde2.OpenCSVSerde';
```

3. In a Terminal window, run the following statement (all on one line) to move the **tunnels.csv** into the table directory. Do not include the **$**; that's the prompt to indicate this is a command-line shell command, not a query.

```
$ hdfs dfs -put ~/training_materials/analyst/data/tunnels.csv
/user/hive/warehouse/dig.db/tunnels/
```

You'll learn more about this statement later in this course.

4. Use the data source panel or a Hive **SELECT *** statement to verify that the **tunnels** table has the data, with values in the correct columns. Remember that you can query this table only with Hive, not with Impala.

## Working with Unstructured and Semi-Structured Data

The sections below describe two SerDes for working with unstructured and semi-structured data: **JsonSerDe** and **RegexSerDe**.

## JsonSerDe

Hive's **JsonSerDe** is useful for semi-structured data stored in JSON (JavaScript Object Notation) format. To use the SerDe, the **CREATE TABLE** statement for the table should include the clause **ROW FORMAT SERDE** followed by the fully qualified name of the Java class that implements the **JsonSerDe**, enclosed in quotes.

For example, a couple of records in a JSON file might look like this:

```
{"id":393930, "name":"Aleks Norkov", "city":"Berkeley", "state":"CA"}

{"name":"Christine Goldbaum", "city":"Boulder City", "state":"NV", "id":82800}
```

The **CREATE TABLE** statement might look like this:

```
CREATE TABLE subscribers

    (id INT, name STRING, city STRING, state STRING)

  ROW FORMAT SERDE 'org.apache.hive.hcatalog.data.JsonSerDe';
```

The table would include the correct data, even though the records were not organized exactly the same.

## RegexSerDe

Hive's **RegexSerDe** is useful for semi-structured or unstructured data. It identifies the fields within each record based on a ***regular expression*** (a way to describe patterns in text). For example, using the **RegexSerDe**, Hive can directly read a log file that lacks consistent delimiters. As with the other SerDes, the **CREATE TABLE** statement for the table should include the clause **ROW FORMAT SERDE** followed by the fully qualified name of the Java class that implements the **RegexSerDe**, enclosed in quotes. To specify the regular expression, use **WITH SERDEPROPERTIES("input.regex"="*regular expression*").** (If you're unfamiliar with regular expressions, see this Regular Expressions Tutorial.)

For example, here are two sample records from a log file. The fields are separated by a space, but there are also spaces within the quoted comment string. This makes it difficult to use the usual **ROW FORMAT DELIMITED** clause, because it would not be able to distinguish the spaces within the quotes from the delimiting spaces.

```
05/23/2016 19:45:19 312-555-7834 CALL_RECEIVED ""

05/23/2016 19:48:37 312-555-7834 COMPLAINT "Item damaged"
```

You could do some processing of the data to convert it, but the **RegexSerDe** allows you to work directly with the raw data through Hive. (This could be helpful for log files that are constantly being generated, adding to the data you want to use.) To do this, use a **CREATE TABLE** statement like the following.

```
CREATE TABLE calls (

    event_date STRING, event_time STRING,

    phone_num STRING, event_type STRING, details STRING)

  ROW FORMAT SERDE 'org.apache.hadoop.hive.serde2.RegexSerDe'

    WITH SERDEPROPERTIES ("input.regex" =

      "([^ ]*) ([^ ]*) ([^ ]*) ([^ ]*) \"([^\"]*)\"");
```

Inside the regular expression, the parentheses define capture groups. The value of each field is the text matched by the pattern within each pair of parentheses. The first four fields capture any number of non-space characters, with the fields separated by one space. The last field begins after the literal quote character and captures any number of non-quote characters, then ends before the following quote character. The quote characters must be escaped with backslashes because they are themselves within a quoted string.

The five captured fields become the five columns of the table: **event_date**, **event_time**, **phone_num**, **event_type**, and **details**. Although this example uses the data type **STRING** for all five columns, the **RegexSerDe** does support other data types.

Note that **RegexSerDe** is for deserialization (reading), but it doesn't support serialization (writing). You can use **LOAD DATA** to load an existing file, but **INSERT** will not work.

In the VM, try creating a couple of tables using these SerDes.

First, do the following to create a table using a JSON file for its data.

1. Examine the file in **/user/hive/warehouse/subscribers/**. This is the same example data used above.

2. Create the **subscribers** table. In Hive, execute the following statement:

```
CREATE EXTERNAL TABLE default.subscribers

    (id INT, name STRING, city STRING, state STRING)

ROW FORMAT SERDE 'org.apache.hive.hcatalog.data.JsonSerDe';
```

3. Execute a **SELECT *** statement to verify that the table has been created with the correct values in each column. Note that although the two rows gave the column values in different orders, each row has the correct values (the two IDs are **39390** and **82800**, for example).

4. ***Optional:*** You can drop the table if you like. If you used **EXTERNAL** as noted above, dropping the table will not delete the data, so you can come back and recreate it later, if you like.

Now use the **RegExSerDe**. The data (see the example below) has fields that are not delimited; instead they use fixed widths.

**1030929610759620160829012215Oakland  CA94618**

| Field | Length |
|---|---|
| cust_id | 7 |
| order_id | 7 |
| order_dt | 8 |
| order_tm | 6 |
| city | 20 |
| state | 2 |
| zip | 5 |

5. ***Optional:*** The sample data is in **/user/hive/warehouse/fixed**. (It's just the one row provided above.) You can take a look if you like.

6. Take a look at the regular expression in this **CREATE TABLE** statement, and see how it will split the row of data into the seven fields. In a regular expression, **\d** matches any digit, a dot (**.**) matches any character, and **\w** matches any word character (letters, numbers, or the underscore character). The first **** in **\\d** and **\\w** is to escape the second **** character. This lets Hive know this second **** is part of the regular expression and not the start of an escape sequence representing a special character.

```
CREATE EXTERNAL TABLE fixed
    (cust_id INT, order_id INT, order_dt STRING, order_tm STRING,
    city STRING, state STRING, zip STRING)
  ROW FORMAT SERDE 'org.apache.hadoop.hive.serde2.RegexSerDe'
  WITH SERDEPROPERTIES ("input.regex" =
```

```
"(\\d{7})(\\d{7})(\\d{8})(\\d{6})(.{20})(\\w{2})(\\d{5})");
```

7. In Hive, execute the statement above.

8. Verify the results using the data source panel or a **SELECT *** query.

9. ***Optional:*** You can drop the table if you like. If you used **EXTERNAL** as noted above, dropping the table will not delete the table, so you can come back and recreate it later, if you like.

## MANAGING EXISTING TABLES

## EXAMINING, MODIFYING, AND REMOVING TABLES

In the previous few lessons, you learned how to create databases and tables. But what if you make a mistake when doing this? For example, what if you realize after creating a table, that you created it in the wrong database, or that one of the columns needs to be renamed or needs to have a different data type? Or what if at a later date, you need to add new columns to an existing table, or perhaps completely drop an existing table? These are just a few examples of cases when you'll need to modify or remove a database or table. In this lesson, you'll learn how, using alter and drop statements. In fact, you've already seen some incidental examples of alter and drop statements, in some of the earlier lessons in this course. In this lesson, we'll introduce these statements more systematically and describe all the implications of using them. For example, what happens to a table's data files when you drop that table? What changes must you make to the data files if you add a new column to a table? Often, before you modify a table, you want to examine it to see its existing schema and other properties. So in this lesson, we'll also review how to do that using the describe statement. We'll introduce a variation on the described statement, and we'll introduce another statement that's useful for seeing how a table was created.

## EXAMINING TABLE STRUCTURE

In real-world contexts, it's common that ***after*** you create a table with Hive or Impala, you realize that the data would be better represented using a different schema, or that some other property of the table needs to be modified. So you'll often need to make changes to tables. You'll see how to do this in the "Modifying Existing Tables" reading later in this lesson.

But before you modify a table, you'll want to examine it to see its existing schema and other properties. There are two commands that are useful to help you understand the state of your tables: **DESCRIBE** and **SHOW CREATE TABLE**. Additionally, **SHOW CREATE TABLE** is useful when you need to recreate tables in a different environment.

### DESCRIBE (and DESCRIBE FORMATTED)

You might recall the **DESCRIBE** utility statement. You can use the **DESCRIBE** statement to see what columns are in a table, by running the command **DESCRIBE *tablename***. The results show the names and data types of all the columns, and sometimes a comment for each column.

To see more detailed information about a table, use the **DESCRIBE FORMATTED** command. That command shows additional details, including the file format and storage location of the table's data files.

### SHOW CREATE TABLE

Another way to understand the structure and properties of a table is to see the **CREATE TABLE** statement that created it. With Hive and Impala, you can do this using the **SHOW CREATE TABLE** statement.

Furthermore, if you made changes to the schema or other properties of a table after creating it, then the output of the **SHOW CREATE TABLE**statement will reflect all of those changes. This makes it particularly useful for recreating a table; instead of issuing the original **CREATE TABLE** statement, followed by a series of other statements to modify the table, you can use **SHOW CREATE TABLE** to display a single **CREATE TABLE** statement to recreate the table in its current state. You can copy that **CREATE TABLE** statement and execute it in a different environment that doesn't share the same metastore. This is especially useful when migrating tables from a development or test environment to a production environment.

First compare the results of a **DESCRIBE** statement with and without the **FORMATTED** keyword.

1. Execute the following commands in Hive and notice the difference in the details provided. (You must use Hive because the **dig.tunnels** table was created using a Hive SerDe.)

   **DESCRIBE dig.tunnels;**

   **DESCRIBE FORMATTED dig.tunnels;**

Do Steps 2 and 3 to see how you can tell if a table is (internally) managed or unmanaged (that is, externally managed).

2. Look again at the **DESCRIBE FORMATTED** results for **dig.tunnels**. Look down the results for **Table Type**; the value should be **MANAGED_TABLE**. This means it was created ***without*** the **EXTERNAL** keyword.

3. Compare that to an externally managed table. You can run this in Hive or Impala:

   **DESCRIBE FORMATTED default.investors;**

   Again look for **Table Type** and note the value for it.

4. The **dig.tunnels** table was created with a SerDe. Look again at the results of the **DESCRIBE FORMATTED** command for that table, or re-run the command in Hive if necessary. Look for **SerDe Library** in the **col_name** column, and see what the **data_type** value is for that.

5. Although you haven't made any modifications to a table, try the **SHOW CREATE TABLE** statement with the **default.investors** table:

   **SHOW CREATE TABLE default.investors;**

   Take some time to review the result. You'll see a lot of familiar things (like **EXTERNAL** and **ROWS DELIMITED** keywords); but you'll probably see some things that are not so familiar. For this table, you'll see **TBLPROPERTIES** settings that you don't recall setting and you might not understand. These are settings that happen invisibly, by default. Don't worry about it—this is not the statement you should have used to create the table; instead, it's ***one possible*** statement that you ***can*** use to produce the exact table that you currently have.

## DROPPING DATABASES AND TABLES

As you saw in the "Creating Databases and Tables…" readings, you can remove (drop) both databases and tables using **DROP DATABASE** or **DROP TABLE** statements. As with the **CREATE** statements, you can conditionally drop a database by using **IF EXISTS** in the statement. This will avoid an error in the case that the database or table does not exist. The syntax is:

**DROP DATABASE IF EXISTS *database_name*;**
**DROP TABLE IF EXISTS *table_name*;**

## Behavior with Managed or Unmanaged (External) Tables

When you drop a ***managed*** table (that is, one that you created ***without*** the **EXTERNAL** keyword), the table's storage directory will be deleted. That means ***you will delete all the data*** for that table! This is true whether or not the table's storage directory is under the Hive warehouse directory. (The only exception to this rule is if Hive or Impala does not have the permission required to delete the files in the storage directory, for example if they are in an S3 bucket to which Hive and Impala have only read access.)

However, when you drop an ***unmanaged*** (also called ***externally managed***) table, the data in the table's storage directory will ***not*** be deleted. In this case Hive and Impala understand that this data is intended to be managed outside of their control, it will not delete the directory that holds the data. (This is true even if that directory is under the Hive warehouse directory.)

Always exercise caution when issuing a **DROP TABLE** statement, and be sure you understand what data, if any, will be lost.

## Dropping a Database That Contains Tables

As a safety feature, Hive and Impala will throw an error if you attempt to drop a database that contains tables. This is to prevent unintended removal of data.

You can override this safety feature by using the **CASCADE** keyword in the **DROP DATABASE** statement. The syntax is:

**DROP DATABASE *database_name* CASCADE;**

Use this with ***great*** caution! Not only will it remove the database, it will remove all tables within it, ***including deleting the data*** for all managed tables within the database.

## MODIFYING EXISTING TABLES

After creating a table with Hive or Impala, you might need to modify the table definition. This might be because there was a mistake in your **CREATE TABLE** statement, or because the structure of the underlying data has changed, or perhaps because you need to use a different naming convention.

To modify a table definition, use the **ALTER TABLE** statement. The general syntax is

**ALTER TABLE *tablename ACTION parameters***

The keyword used in place of ***ACTION*** depends on what kind of modification you want to make. The following sections present some of the most common modifications. There are other possibilities in addition to the ones presented here. You can look at the documentation (Impala ALTER TABLE statement or Hive Language Manual DDL: Alter Table) if you want more information, but at this point in the course, the options presented here probably are sufficient.

***Note:*** Sometimes it's easier to just drop your table and recreate it, rather than making changes to the table—but you must be careful not to delete your data in the process. In traditional RDBMSs this is not feasible, but in the big data systems, the data and metadata are separated. If you created an externally managed table using **EXTERNAL**, then dropping the data is purely a metadata operation and does not affect the data. If you have an internally managed table, you could make it externally managed first (see the "Changing to an Unmanaged (External) Table" section below). It's up to you whether you would rather make the changes to a table, or drop it and recreate the table.

## RENAMING A TABLE

To rename a table, use

**ALTER TABLE *old_tablename* RENAME TO *new_tablename*;**

For example, this renames the table **customers** to **clients**:

```
ALTER TABLE customers RENAME TO clients;
```

When you rename a table, Hive or Impala changes the table's name in the metastore, and if the table is internally managed, it also renames the table's directory in HDFS.

## Moving a Table to a Different Database

To move a table to a different database, you also use **RENAME TO**, and you specify the fully qualified names of the old (existing) and new tables, including the database names:

```
ALTER TABLE old_database.tablename RENAME TO new_database.tablename;
```

For example, this moves the existing table named **clients** from the **default** database to the **dig** database:

```
ALTER TABLE default.clients RENAME TO dig.clients;
```

When you move a table to a different database, Hive or Impala changes the associated metadata in the metastore, and if the table is managed, it also moves the table's directory in HDFS into the subdirectory for the different database.

## Changing Column Name or Data Type

To change the name or data type of a column, use

```
ALTER TABLE tablename CHANGE old_colname new_colname type;
```

If you are not changing the data type, you still need to supply the type. If you are not changing the column name—only the data type—then repeat the column name.

For example, the following changes the **first_name** column (of type **STRING**) in the **employees** table to **given_name** (but keeps it a **STRING** column).

```
ALTER TABLE employees CHANGE first_name given_name STRING;
```

The following example changes **salary** from **INT** to **BIGINT** without changing the column name:

**ALTER TABLE employees CHANGE salary salary BIGINT;**

---

## CHANGING COLUMN ORDER *(HIVE ONLY)*

When you create a table for existing data, your columns need to be provided in the same order that they appear in the data files. If you have made a mistake and put them in a different order in the **CREATE TABLE** statement, you can fix it with the **ALTER TABLE** statement.

To change where a column goes using Hive, use the **CHANGE** keyword just as if you were changing the column name and add either **AFTER *column*** or **FIRST** at the end.

For example, the **employees** table in the VM has columns in this order: **empl_id**, **first_name**, **last_name**, **salary**, **office_id**. Suppose you discover that the file data actually lists the office ID before the employee's salary. The following moves the **salary** after the **office_id** column:

**ALTER TABLE employees CHANGE salary salary INT AFTER office_id;**

If the column to move needs to be the first (leftmost) column, then the statement would be

**ALTER TABLE *tablename* CHANGE *col_name col_name col_type* FIRST;**

Notes

This feature is available in Hive but not Impala. For Impala, see “Replacing All Columns” for an alternative method.

You always need to give the “old” and “new” names of the column you're moving, along with its data type, ***even if those details are not changing.***

This does not change the data files. If you change the order of columns to something ***different*** from the order in the files, you'll need to recreate the data files using the new order.

## ADDING OR REMOVING COLUMNS

You can add one or more columns to the end of the column list using **ADD COLUMNS**, or (with Impala only) you can delete columns using **DROP COLUMN**. The general syntax is

**ALTER TABLE *tablename* ADD COLUMNS (*col1 TYPE1,col2 TYPE2,…* );**

**ALTER TABLE *tablename* DROP COLUMN *colname*;**

For example, you can add a **bonus** integer column to the **employees** table:

**ALTER TABLE employees ADD COLUMNS (bonus INT);**

Or you can drop the **office_id** column from the **employees** table:

**ALTER TABLE employees DROP COLUMN office_id;**

Notes

**DROP COLUMN** is not available in Hive, only in Impala. However, see "Replacing All Columns" below.

You can only drop one column at a time. To drop multiple columns, use multiple statements or use the method to replace columns (see below).

You cannot add a column in the middle of the list rather than at the end. You can, however, add the column then change the order (see above) or use the method to replace columns (see below).

As with changing the column order, these do not change the data files.

- If the table definition agrees with the data files before you drop any column other than the last one, you will need to recreate the data files without the dropped column's values.

- If you drop the last column, the data will still exist but it will be ignored when a query is issued.
- If you add columns for which no data exists, those columns will be **NULL** in each row.

## Replacing All Columns

You can also completely replace all the columns with a new column list. This is helpful for dropping multiple columns, or if you need to add columns in the middle of the list. The general syntax is

**ALTER TABLE *tablename* REPLACE COLUMNS (*col1 TYPE1,col2 TYPE2,…* );**

This completely removes the existing list of columns and replaces it with the new list. Only the columns you specify in the **ALTER TABLE** statement will exist, and they will be in the order you provide.

Note

Again, this does not change the data files, only the metadata for the table, so you'll either want the new list to match the data files or need to recreate the data files to match the new list.

## Changing to an Unmanaged (External) Table

If you have created a table as a managed table (without the **EXTERNAL** keyword) and later realize you want it unmanaged, you can use **ALTER TABLE** with **TBLPROPERTIES** to make it unmanaged. This is particularly helpful if you want to drop a table without losing the data.

The general syntax is

**ALTER TABLE *tablename* SET TBLPROPERTIES('EXTERNAL'='TRUE');**

Notes

Both **EXTERNAL** and **TRUE** are in quotes, and they must be uppercase, here.

You can also use **SET TBLPROPERTIES** with other properties that were not set at creation.

Try It!

Test these out with the **investors** table.

The first thing you'll do is verify that the table is unmanaged (external) so you can drop the table without losing the data, in case you make a mistake. You can then recreate the table using the exercise from "The ROW FORMAT Clause" reading in "The CREATE TABLE Statement" lesson, where you first created the **investors** table. (You can leave this alteration, no need to change it back.) If it's ***not*** unmanaged, then you should change it.

1. In Hive or Impala, execute the following on the **customers** table so you can see what a ***managed*** table looks like. Be sure the **default** database is your active database.

```
DESCRIBE FORMATTED customers;
```

Look down the results for **Table Type**; the value should be **MANAGED_TABLE**. This means it was created without the **EXTERNAL** keyword.

2. Now run the same command on the **investors**table, and note what the value is for **Table Type**. If it's also **MANAGED_TABLE**, execute the following to change it to an unmanaged table, then run the **DESCRIBE FORMATTED** statement again and check the value for **Table Type**. (If the value is not **MANAGED_TABLE**, you don't need to run this command—though it does no harm if you do.)

```
ALTER TABLE investors SET TBLPROPERTIES('EXTERNAL'='TRUE');
```

Next, try changing the name.

3. Execute the following statement:

```
ALTER TABLE investors RENAME TO companies;
```

4. Verify that the table name changed in the data source panel (refresh the display if necessary), or by running **SELECT * FROM companies;**

5. Change the name back to **investors** and verify the change.

Now move it to the **dig** database.

6. Execute the following statement:

```
ALTER TABLE default.investors RENAME TO dig.investors;
```

7. Verify that the table is no longer in the **default** database, and that it ***is*** in the **dig** database.

8. Check the Hive warehouse directory. The **investors**subdirectory has not moved to the **dig.db** directory—it is still in the **default** database. Do you know why? (See the "Postscript" section below for the answer!)

9 .Change the directory back to **default** and verify the change.

Change the column **amount** to **holdings**.

10. Execute the following statement:

```
ALTER TABLE investors CHANGE amount holdings INT;
```

11. Verify that the column's name has changed, using the data source panel or by running **DESCRIBE investors;**

12. Change the column name back to **amount**and verify the change.

Change the column **amount**from **INT**to **BIGINT**.

13. Execute the following statement:

```
ALTER TABLE investors CHANGE amount amount BIGINT;
```

14. Verify that the column's type has changed, using the data source panel or by running **DESCRIBE investors;**

15. Change the column type back to **INT** and verify the change.

Use Hive ***(not Impala)***to put the **amount**column at the end instead of the middle, and see the effect.

16. First take a look at the data by running **SELECT * FROM investors;**

17. Execute the following statement:

**ALTER TABLE investors CHANGE name name STRING AFTER amount;**

18. Verify that the columns have been reordered in the metadata ***but not in the data itself*** by running

**SELECT * FROM investors;**

Notice that **amount** is **NULL** in each row—this is because the data being loaded is the non-numeric character data that used to go into **name**. Since **amount** is an**INT** column, the data isn't valid. The integer data meant for **amount** is valid for the **STRING** column **name**, so those values are not **NULL**. (The point here is that the data itself did not change.)

19. Change the column order back by executing

**ALTER TABLE investors CHANGE name name STRING FIRST;**

20. Verify that the columns are back to normal using the **SELECT *** query.

Use Impala to drop a column and then add it back. (You can add with Hive or Impala, but Hive doesn't recognize the **DROP COLUMN** keyword, so to make things easier, use Impala for both.)

21. First drop the **share** column:

**ALTER TABLE investors DROP COLUMN share;**

22. Run the **SELECT *** query to verify the column has been dropped. The values in the other columns has not changed.

23. Add the **share** column back:

**ALTER TABLE investors ADD COLUMNS (share DECIMAL(4,3));**

24. Verify that the column has been restored.

Finally, change the columns completely!

25. Execute the following statement:

**ALTER TABLE investors REPLACE COLUMNS (company STRING, holdings BIGINT, share DOUBLE);**

26. Verify the table columns have changed using **DESCRIBE investors;**

27. Restore the original columns:

**ALTER TABLE investors REPLACE COLUMNS**

**(name STRING, amount INT, share DECIMAL(4,3));**

28. Verify that the table has changed back.

Postscript

The **investors** table directory didn't move when moved to a different database because it's an unmanaged table. Just as dropping the table won't delete the data, changing the database will not move the data.

Try moving **customers**, which is managed, to the**dig** database,and confirm that the table directory moved in that case. (That is, **/user/hive/warehouse/** should no longer have the **customers/** subdirectory, but **/user/hive/warehouse/dig.db/** should have it instead.) You might need to refresh the display. Be sure to move the table back to the **default** database when you're done.

---

## APACHE HIVE AND APACHE IMPALA INTEROPERABILITY

Hive and Impala Interoperability

In general, Hive and Impala work well together. If you create a table using Hive, you can query it in both Hive and Impala. If you create a table on Impala, you can also use both engines to query it. That's in general. By now you should have read

about some techniques that work in Hive, but not Impala. Like the ALTER TABLE command for changing the order of columns. There are also some commands that work in Impala, but not Hive. Like the command for dropping a column. There are also a few parts of the CREATE TABLE syntax, that one or the other might not support. For example, in Impala, you can use the LIKE PARQUET clause to create a table using this schema information that's embedded inside a parquet file. You'll learn more about this in next week's materials.

But Hive does not support this LIKE PARQUET clause. It's also possible that a table created with one of these engines cannot be queried by the other. For example, if you use some SERDE like this one in a CREATE TABLE statement in Hive, then Impala will not be able to query that table. So while you mostly can rely on tables being available in both engines, remember that there are some things that require a specific engine. If you get error messages from one engine, check whether that engine supports what you're trying to do.

Remember that Hive is typically slower than Impala, but Hive is more general than Impala in the types of file formats it supports for tables. Impala is designed to be much faster and so it specializes in the use of the best file formats for fast running queries, formats like Apache Parquet. When you're on the job, you will often find yourself using Hive and its wide variety of available SERDEs to read data in many different formats. Then Hive can put that data into new tables in a format like Parquet for fast querying using Impala.

## IMPALA METADATA REFRESH

While you can usually use either Hive or Impala to create end-query tables, there's an important difference between how Hive and Impala access the metastore. Hive retrieves metadata from the metastore every time it builds a query, but Impala does not. Impala caches metadata in memory to reduce query latency. This Impala cache metadata consists of the structure and locations of tables retrieved from the metastore, and also additional information about table data files retrieved from the data storage system like HTFS or S3.

Impala's metadata cache helps it return query results as quickly as possible. But the cached metadata can get out of sync with the metadata in the metastore, and with the stored data files. This happens when changes are made outside of

Impala. For example, when new tables are created using hive, when table data is imported using Hughes table browser, or when table data is added using HTFS command. When changes like this occur outside of Impala, it's necessary to refresh Impala's metadata cache.

There are different ways to do this depending on what changes were made outside of the Impala. The refresh command updates the information that Impala caches about the schema of a particular table, and the locations and files for that table in the data storage system. Use this command if you have altered a table schema such as renaming a column or added data to the table. The syntax is refresh and the table name. But if you've added a new table to the database from outside of Impala, then you'll need to use a different command to update Impala's metadata cache. The command is invalidate metadata and the table name. This command causes Impala to add all the information about this new table to its metadata cache.

Finally ,you can also use invalidate metadata without specifying a table. But be careful when you use this, it will mark the metadata for all tables as tail, and will reload all the metadata when a new query is issued. For a large production environment with many tables, this can be a very expensive operation and it can take a long time. Note that when Impala itself modifies the metastore or any stored files, it can automatically update the cache metadata. So it's only changes from outside of Impala that require you to use any of these commands.

---

### DATA TYPES AND FILE TYPES

Learning Objectives

- List the common data types and contrast their properties
- Choose appropriate data types for existing data
- Describe common file types and their advantages and disadvantages
- Choose appropriate file types for storing data

---

## DATA TYPES

Welcome to week three of managing big data in clusters and cloud storage. In this week of the course, you'll learn about data types and file type. At this point you

already have some knowledge of data types. Remember that every column in a table has a specific data type. You've used a few data types already like strings and int in create table statements earlier in the course. In this week of the course, we'll describe more details about these and about the other data types that you can use with Hive and Impala. You'll learn about the limits of the different data types, what they're intended for, and how to choose the right ones when you create a table. Then in the second half of this week, you'll learn about file types. These are the different file formats that you can use to store the data in Hive and Impala tables. You've already learned a little bit about file formats and you've seen some examples of files in delimited text format. So in this week, we'll quickly cover and review the text file format and then go into more detail about three other popular file formats, Avro, Hark, and NRC. You'll learn what's different about these file formats, when you choose one of them versus text files and how to efficiently create tables with files in different formats. We'll also give a quick review of some other file formats that are less often used.

## OVERVIEW OF DATA TYPES

Every column in a table has a data type. When you create a table with Hive or Impala, you specify those data types along with the names of the columns. In this lesson, you'll learn about the data types that Hive and Impala support. These break down into three main categories: integer data types, which represent whole numbers, decimal data types, which can represent numbers with fractional parts, and character string data types which represent text.

Hive and Impala also support some various other data types that do not fall into these three categories. You'll learn about those as well. You should already have a basic working understanding of some of these data types because we've used some of them like string and int in creating table statements earlier in this course. In this lesson, you'll learn the specific details of these and other data types, and what limitations they have.

## INTEGER DATA TYPES

Hive and Impala support four integer data types: **TINYINT**, **SMALLINT**, **INT**, and **BIGINT**. These represent whole numbers with no fractional parts.

Larger integer types allow you to represent larger ranges of numbers, as shown in the table below, but processing them requires more memory, so you should generally use the smallest integer type that accommodates the full range of values in your data.

| Integer Type | Range |
|---|---|
| **TINYINT** | -128 to 127 |
| **SMALLINT** | -32,768 to 32,767 |
| **INT** | -2,147,483,648 to 2,147,483,647 (approximately 2.1 billion) |
| **BIGINT** | -9,223,372,036,854,775,808 to 9,223,372,036,854,775,807 (approximately 9.2 quintillion) |

Unlike in some relational databases, all integer types in Hive and Impala are signed; that is, they can represent positive or negative integer values. There are no unsigned integer types.

## DECIMAL DATA TYPES

Hive and Impala support three decimal data types: **FLOAT**, **DOUBLE**, and **DECIMAL**. These types represent numbers that can include fractional parts.

The **FLOAT** and **DOUBLE** types represent floating-point numbers, which do not have a predetermined number of digits after the decimal point. **DOUBLE** offers greater range and precision than **FLOAT**, but processing **DOUBLE**s requires more memory than processing **FLOAT**s, so in general, choose **FLOAT** unless you need the range or precision that **DOUBLE** offers.

Because of the binary system used to store numbers, both **FLOAT** and **DOUBLE** data types can produce unexpected inaccuracies, even with

seemingly simple arithmetic like 0.1 + 0.2. (See The Floating Point Guide for more about this.)

**DOUBLE** is more accurate because **DOUBLE** uses 64 bits to store each number, while **FLOAT** uses only 32 bits. (So **DOUBLE** has ***double*** the number of bits.) This means **FLOAT** is typically accurate up to 7 digits, while **DOUBLE** is accurate up to 15 or maybe 16 digits.

The **DECIMAL** type represents numbers with fixed precision and scale. When you create a **DECIMAL** column, you specify the precision, ***p***, and scale, ***s***. Precision is the total number of digits, regardless of the location of the decimal point. Scale is the number of digits after the decimal place. To represent the number 8.54 without a loss of precision, you would need a **DECIMAL** type with precision of at least 3, and scale of at least 2.

The table below illustrates the difference in precision using results for ***π.*** Note that the **DECIMAL(17,16)** type means there is a total of 17 digits, with 16 of them after the decimal point.

| Data Type | Result for $\pi$ (**bold** are accurate) |
|---|---|
| **FLOAT** | **3.141592**7410125732 |
| **DOUBLE** | **3.1415926535897931** |
| **DECIMAL(17,16)** | **3.1415926535897932** |

Using the **DECIMAL** type, it is possible to represent numbers with greater precision than the **FLOAT** or **DOUBLE** types can represent. The maximum allowed precision and scale of the **DECIMAL** type are both 38. (Hive can allow larger values of precision and scale, but Impala does not support them.)

The table below describes the range of **DOUBLE**, **FLOAT**, and **DECIMAL(38,0)**. The ranges described below are the largest negative and largest positive number that each data type can represent.

| Data Type | Range |
|---|---|
| **FLOAT** | -3.40282346638528860 * 10^38 to 3.40282346638528860 * 10^38 |

| Data Type | Range |
|---|---|
| DOUBLE | -1.79769313486231570 x 10^308 to 1.79769313486231570 x 10^308 |
| DECIMAL(38,0) | -10^38 + 1 to 10^38 - 1 |

For representing currency, you should use **DECIMAL** instead of **FLOAT** or **DOUBLE**; this prevents loss of precision, which is typically of paramount importance with financial data. Another choice for currency is to use an integer type to represent, for example, the number of cents, instead of storing dollars with fractional parts.

## CHARACTER STRING DATA TYPES

Hive and Impala support three character data types: **STRING**, **CHAR**, and **VARCHAR**. These types represent alphanumeric text values.

The **STRING** data type represents a sequence of characters with no specified length constraint.*

If you're familiar with relational databases, you are probably more accustomed to the character types **CHAR** and **VARCHAR**, which have a specified length.

The **CHAR** type represents fixed-length character sequences, with a precise specified length. Values longer than the specified length are truncated. Values shorter than the specified length are padded with spaces. If you assign the 13-character value **Impala rules!** to a **CHAR** column with length 16, then Hive and Impala will pad that value with three spaces to make it 16 characters long: **Impala rules!**␣␣␣ (The three symbols shown in this example represent spaces.)

The **VARCHAR** type represents character sequences with a maximum specified length.

Values longer than the maximum are truncated, but values shorter than the maximum are not padded with spaces. If you attempt to assign the 13-character value **Impala rules!** in a **VARCHAR** column with a maximum length of 10, then Hive and Impala will truncate that value to 10 characters, discarding the last three characters: **Impala rul**. However, if the maximum length is 13 or more, the stored

value will be exactly **Impala rules!** (with no extra spaces as you would get with the **CHAR** type).

The table here summarizes these examples.

| Data Type | Description | Value (attempting Impala rules!) |
|---|---|---|
| STRING | Any number of characters | Impala rules! |
| CHAR(10) | Exactly 10 characters | Impala rul |
| CHAR(16) | Exactly 16 characters | Impala rules! ␣ ␣ ␣ |
| VARCHAR(10) | At most 10 characters | Impala rul |
| VARCHAR(16) | At most 16 characters | Impala rules! |

With **CHAR** types, trailing spaces are ignored in comparisons.
With **VARCHAR** and **STRING** values, any trailing spaces are considered in comparisons. (This makes sense, since neither is automatically padded—trailing spaces are not considered to be "padding" in these cases.)

You should generally choose **STRING** over **CHAR** or **VARCHAR**. **STRING** offers greater flexibility and ease of use, and in some cases Hive and Impala have better performance and compatibility when using **STRING** columns. But if you have a particular need for string values with precise lengths or with maximum lengths, then you could use **CHAR** or **VARCHAR**.

### *Footnote: Actual String Limits

There actually are practical limits to the length of strings, though in most real-world applications, it's unlikely you'll ever come up against them. For example, in Impala, these are the considerations for lengths of strings (taken from STRING Data Type in Cloudera's Impala documentation):

- The hard limit on the size of a **STRING** and the total size of a row is 2GB.
- If a query tries to process or create a string larger than this limit, it will return an error to the user.
- The limit is 1GB on **STRING** when writing to Parquet files.
- Queries operating on strings with 32KB or less will work reliably and will not hit significant performance or memory problems (unless you have very complex queries, very many columns, and so on.)

- Performance and memory consumption may degrade with strings larger than 32KB.

This varies somewhat according to which version of Impala you are using, so if you are working with exceptionally large strings, check the documentation.

## OTHER DATA TYPES

Besides the integer, decimal, and character data types, Hive and Impala support several other simple data types.

The **BOOLEAN** type represents a Boolean value, that is, a **true** or **false** value.

The **TIMESTAMP** type represents an instant in time. **TIMESTAMP**s can represent values with up to nanosecond precision. They are interpreted as being in UTC, or Coordinated Universal Time, but Hive and Impala provide functions for conversion to local timezones.

Hive (but not Impala) provides a **DATE** type, representing a particular day in the form **YYYY-MM-DD**, without a time of day. With Hive, a **TIMESTAMP**can be stripped of the time of day by casting to a **DATE** type.

There's also the **BINARY** type, which can represent any sequence of raw bytes, and also is supported only by Hive, not by Impala. This is analogous to the **VARBINARY** type in some relational databases.

The table below summarizes these types.

| Data Type | Description | Example Value |
|---|---|---|
| **BOOLEAN** | True or false | **true** |
| **TIMESTAMP** | Instant in time | 2019-02-25 16:51:05 |
| **DATE** (Hive only) | Date without time of day | 2019-02-25 |

| Data Type | Description | Example Value |
| --- | --- | --- |
| **BINARY** (Hive only) | Raw bytes | N/A |

Hive and Impala also support complex types (**ARRAY**, **MAP**, and **STRUCT**), but they are an advanced topic for this course. If you complete the Honors lessons (Week 5), you will learn about complex types then.

## WORKING WITH DATA TYPES

Choosing the Right Data Types

At this point in the course, you should know what datatypes Hive and Impala support, and you should have a sense of how to choose the appropriate data types for the columns in your data. A few important points to remember are, Impala does not currently support the date type but Hive does. With integers, choosing a larger integer type allows you to represent a larger range of numbers, but storing and processing the larger integer types uses more resources, so you should generally choose the smallest integer type that accommodates the full range of values in each integer column.

With character strings, you should generally choose the STRING type, not CHAR or VARCHAR unless you have some specific reason to be using those. The STRING type offers greater flexibility and ease of use, and in some cases better performance. To avoid loss of precision, you should not use the FLOAT or DOUBLE type to represent currency or other quantities that need to be exact to a specific number of places after the decimal point. For those, you should use decimal and specify the appropriate values of precision and scale.

So when will you need to apply this knowledge about datatypes? Well, there are really only two situations when you'll need to manually specify the names and data types of the columns in a Hive and Impala table. One when you're creating a table to query data that's stored in delimited text files. Delimited text files do not contain any information about the data types of the columns in them. They sometimes do have a header row but that gives only the column names, not their datatypes. In this situation, you are constrained, you need to choose column

names and data types that match the columns in the delimited text files. If you mess up and for example, you choose a numeric datatype for a column that contains character strings, then when you query that table, you'll get unexpected null values in your query results.

Situation two is when you're creating a new empty table that you're going to fill with data. In this situation, you have freedom, you can choose the names and data types you want, and the constraints come later when you're filling in the table, then you'll need to ensure that the data you're putting in matches the table schema. Remember that Hive and Impala will generally not prevent you from creating or filling a table with columns whose datatypes are mismatched where the schema does not match the data files.

Typically you'll only find out about a mismatch later when you query the table. So those are the two situations when you will need to manually specify the column names and data types. There are some other situations when you will not necessarily need to do this. One is when you're creating a new table by cloning the structure of an existing Hive and Impala table. Recall that in that situation, you can use the like keyword in the create table statement to points to the table who schema you want to clone. The other is when the table data is stored in existing Avro or Parquet files. As you'll see in a later lesson, Hive or Impala have ways of automatically determining the schema in those situations. For the remainder of this lesson, I'll provide a few more tips for working with data types.

## EXAMINING DATA TYPES

If you're unsure what data types are assigned to a column—perhaps it's an existing table that someone else created, you think you might have made a mistake, or you just forgot how you defined it—there are different ways to get this information.

## EXAMINE THE TABLE SCHEMA

Of course, you can use Hue's Table Browser, or the data source panel on the left in Hue, to view the table's schema. You can view the names of the columns along

with their data types. As you learned in the Week 2 reading, "Examining Table Structure," you can also use the **DESCRIBE** or **DESCRIBE FORMATTED** commands. Both show what columns are in the table, with their data types and sometimes comments. **DESCRIBE FORMATTED** provides a bit more information, including the format and location of the table's data files.

The **SHOW CREATE TABLE** command can also be used to see a table's definition. You can read the resulting **CREATE TABLE** command to see what the columns are, including what their data types are.

See "Examining Table Structure" in Week 2 if you need a refresher on those commands.

---

### THE TYPEOF FUNCTION IN IMPALA

In Impala (but not Hive), you can also use the **typeof** function in a **SELECT** statement to get the data type:

**SELECT typeof(*colname*) FROM *tablename* LIMIT 1*;***

If you don't use **LIMIT 1** it will return one row for each row in the table.

This method also is useful to see what data type an expression returns. Directly examining a table, whether using Hue's graphical interface or by using a command, will not provide this information. Instead of ***colname*** in the query above, use the expression you're interested in. If the expression doesn't involve a column reference, Impala will allow you to leave off the **FROM** clause.

Follow the steps below to practice using the **typeof** function with Impala. If you want to practice using **DESCRIBE** or **SHOW CREATE TABLE**, see "Examining Table Structure" in Week 2.

1. Use the **typeof** function to find the data type of the **list_price** column in the **fun.games** table. (Use the **SELECT** statement above, replacing ***colname*** and *tablename*.)
2. Some governments add a sales tax to purchases of items such as games. The amount of the tax is a percentage of the price paid. For example, a 7% sales tax would make the tax of a game **0.07 * list_price**, and the final cost

would be **1.07 * list_price**. Find the data type of this expression, for the games in the **fun.games** table.

3. Impala will allow you to omit the **FROM** clause when there is no column reference. Use the **typeof** function in Impala to find the data type of the following expressions. (All you need is **SELECT *expression*;**)

a. **0.6**

b. **cos(0.6)**

c. **ceil(cos(0.6))**

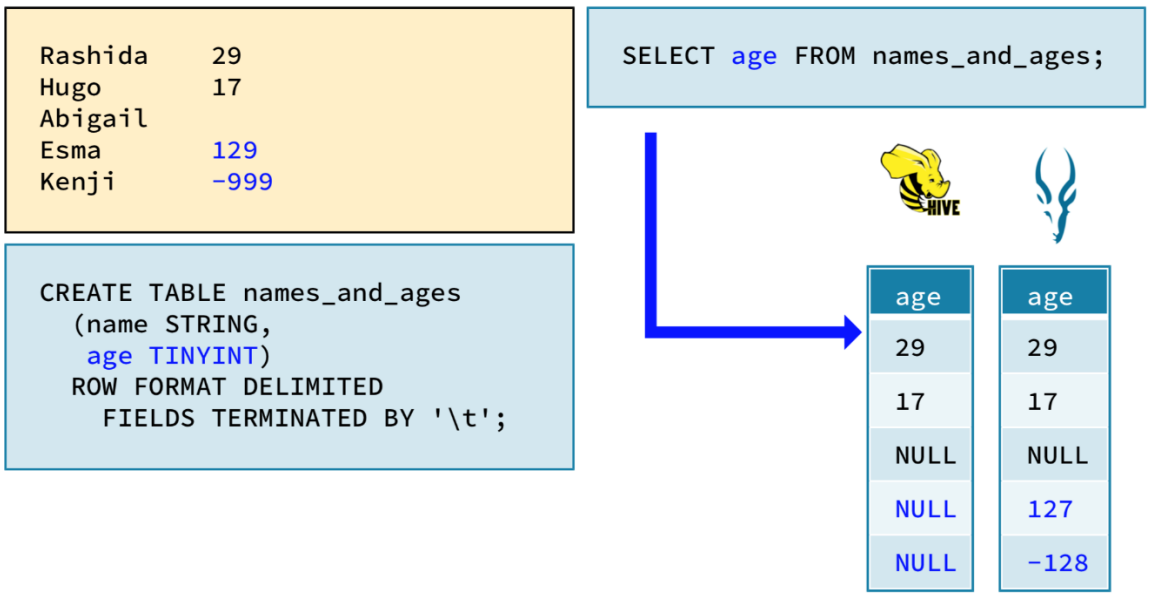
d. **ceil(cos(0.6))/3**

---

## OUT-OF-RANGE VALUES

Both Hive and Impala return **NULL** for **DECIMAL** values that are out of range for the target data type (such as **23.63** in a **DECIMAL(3,2)** column). They also both return **-Infinity** or **Infinity** for **FLOAT** and **DOUBLE** types when the value is too large, and zero when the value is close to zero but too small for the data type's range. (These zero values may be rendered in slightly different ways, for example: **0**, **0.0**, **-0**, or **-0.0**.)

However, there's an important difference between how Hive and Impala handle out-of-range values in integer columns: Hive returns **NULL** for out-of-range integer values, whereas Impala returns the minimum or maximum value for the column's specific integer data type. The example shown below illustrates this.




In this example, there is a table that includes a column named **age** with data type **TINYINT**. The **TINYINT** data type can represent integer values from **-128** to **+127**. But the data file for this table (represented by the yellow box) contains two erroneous age values that are outside this range. The age value for Esma is **129**, which is above the maximum value for the **TINYINT** type. The age value for Kenji is **-999**, which is below the minimum value for the **TINYINT** type.

When you query this table with Hive, it returns **NULL** for these two out-of-range values, but Impala behaves differently. Impala returns **127** for Esma's age (**127** is the maximum value for the **TINYINT** type). And Impala returns **-128** for Kenji's age (**-128** is the minimum value for the **TINYINT** type).

Why does Impala behave differently than Hive in this situation? Notice in the example shown here that Abigail's age is missing in the data file in HDFS. The Impala query results allow you to distinguish between the missing value of Abigail's age (**NULL**) and the out-of-range values of Esma's and Kenji's ages, whereas the Hive query results do not allow you to distinguish between these.

However, one implication of this is that you should choose a data type whose largest and smallest values *do not occur* in your data. For example, if your data contains integers ranging from **-127** to **+126**, then it would be fine to use the **TINYINT** data type. In that case, a value of **-128** or **+127** in the query result will unambiguously indicate an out-of-range value. But if your data contains integers

ranging from **-128** to **+127**, then you should choose **SMALLINT** to avoid ambiguity between the *actual* values **-128** and **+127** and out-of-range values.

## FILE TYPES

Overview of File Types

Recall that the data in Hive and Impala tables is stored in files in a directory in HDFS or S3. There is not just one fixed file format for this table data. Hive and Impala support a number of different file formats. When you create a table with Hive and Impala, you specify which format to use in the stored as Clause of the create table statement. In this lesson, I'll describe some different file formats that Hive and Impala support. I'll focus on the four that are used most often; Textfiles, Avro files, Parquet files and ORC files. I'll describe the advantages and disadvantages of using each of these. Mostly in terms of trade-offs, between human readability and performance and also interoperability with other tools you might work with. Different patterns for storage, ingest or querying can make one format a better choice than another format. With big data, the differences can have significant impact on performance. The choice of format depends partly on which query engine you're using.

Textfiles, Avro files, and Parquet files are compatible with both Hive and Impala. But ORC files are compatible only with Hive. I'll also give a quick overview of some other file formats that are used less often. The SequenceFile and RCFile formats. Sometimes the choice of which file format to use when you create a table, is based simply on which file format the data is already in. For example, if you have an HDFS directory or an S3 bucket with Parquet files already in it, then you can just create a Parquet base table to query that existing data with Hive and Impala without moving it or copying it.

But in other situations you'll have the flexibility to choose which file format to use and you'll convert data into that format as you load it into the table storage directory. You'll learn more about that in the next week of the course. For now, just remember that it is possible to convert data between these different file

formats. So you won't get trapped in a dead end by your choice of file format. However, making the right choice when you first create a table can save you from needing to convert the files later, which can be an expensive operation if the data has grown very large.

## TEXT FILES

Text files are the most basic file type. Virtually any programming language can read and write data in text files, and many applications that data analysts use can work with comma- and tab-delimited text files. They are also human-readable. Values are represented as plain text strings, so you can simply open a text editor and view the data. This is useful when you're investigating a problem.

However, there are downsides to using text files. When used to store large amounts of data, text files are inefficient. Representing numeric values as strings wastes a great deal of storage space. It's difficult to represent binary data like images in a text file; this requires using techniques like Base64 encoding. Converting data between text representations and their native data types requires serialization and deserialization, which slows performance.

Overall, text files offer excellent interoperability but poor performance.

## AVRO FILES

Apache Avro is an efficient data serialization framework. Avro defines a file format that uses optimized binary encoding to store data efficiently. (For an explanation of binary encoding, see "What Is Binary Encoding" from wisegeek.com.) The Avro format is widely supported by big data tools, and it's also designed to work across different programming languages and tools outside the typical big data system. Avro files are suitable for long-term data storage.

An Avro data file includes an embedded schema definition, which makes the file ***self-describing*** —the file itself provides information about what's in the file. Avro is also built to handle ***schema evolution***. This means that it's possible to

add, remove, or modify columns in a Hive or Impala table without needing to make changes to the existing data stored within Avro files. The Avro framework will accommodate these schema changes, so the table and the existing data files will continue to be compatible, even though their schemas do not perfectly match. (For more about schema evolution, see "How Schema Evolution Works" within the linked page. Note that most of the details in that article pertain to Avro schemas in general, but some details are specific to the use of Avro within the Oracle NoSQL Database and so are not applicable to the big data systems presented in this course.)

Overall, Avro offers excellent interoperability and excellent performance, making it a popular choice for general-purpose big data storage.

## PARQUET FILES

Apache Parquet is a *columnar* file format originally developed by engineers at Cloudera and Twitter. Parquet was inspired by a project at Google called Dremel.

Columnar, or column-oriented, file formats organize data by column, rather than by row. This makes them more efficient when you need to process only one or a few columns. For example, see Figure 1 below. The rows (1, 2, 3) and columns (A, B, C) of the data are shown in a tabular structure on the left side. The images on the right side represent how this data could be stored in a row-oriented file format (top) and a column-oriented file format (bottom).

When the data is stored in a row-oriented format, the file organizes the data sequentially first by row (Row 1 consists of A1, B1, C1). When the data is stored like this, Hive and Impala must read each full row even if the query requires processing only one column.

When the data is stored in a column-oriented format, the file organizes the data sequentially first by column (Column A has values A1, A2, A3). When the data is

stored like this, Hive and Impala can skip over the columns that are not part of the query, and quickly read only the values for the columns that are part of the query. This improves query performance, especially for tables with dozens or hundreds of columns.

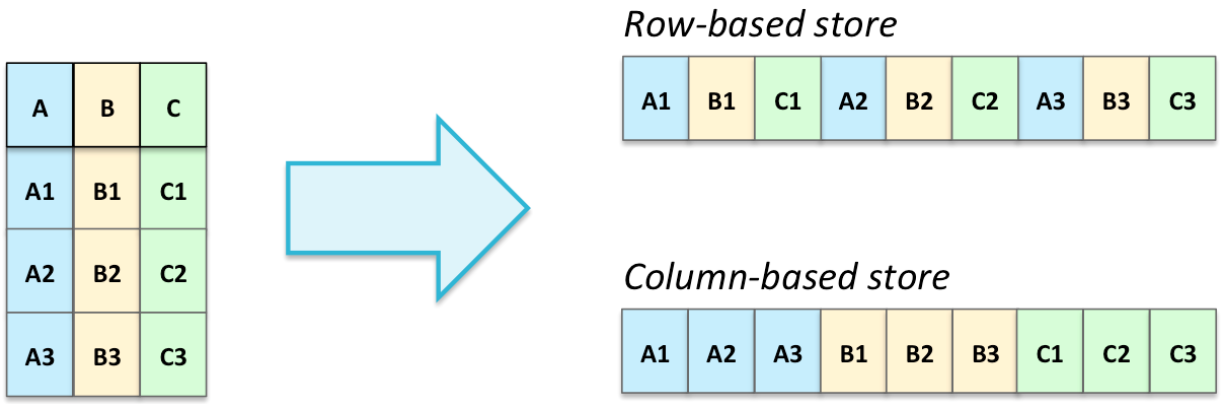



**Figure 1**

Parquet is widely supported by big data tools, including Hive and Impala, and it's designed to work across different programming languages. Like Avro, Parquet embeds a schema definition in the file, and it supports schema evolution.

Parquet uses advanced optimizations to store data more compactly and to speed up queries. It is most efficient when data is loaded all at once or in large batches; this enables Parquet to take advantage of repeated patterns in the data to store it more efficiently. (See Figure 2 below.)

| id | name | city |
|---|---|---|
| 1 | Alice | Palo Alto |
| 2 | Bob | Sunnyvale |
| 3 | Bob | Palo Alto |
| 4 | Bob | Berkeley |
| 5 | Carol | Berkeley |


**Figure 2**

Parquet also uses *compression*. Compression encodes data in a way to take up less storage than an uncompressed file will, but there is a time cost when you read or write the file. That is, encoding for less storage means more time needed to compress (encode) the file before writing it, or to decompress (decode) the file after reading it.

Overall, Parquet offers excellent interoperability and excellent performance, making it a popular choice for columnar data storage.

---

## ORC FILES

Apache ORC (Optimized Record Columnar) is a file format originally developed by engineers at Hortonworks and Facebook. (Hortonworks has since merged with Cloudera.)

ORC is very similar to Parquet. ORC and Parquet were designed to meet many of the same needs, and internally they use many of the same techniques to achieve excellent performance.

ORC is often used with Hive. To take advantage of certain features of Hive, you *must* store table data in the ORC file format. (These features are not covered by this course or in the other courses in this specialization.) However, ORC is not widely supported by other tools. Impala cannot query tables whose data is stored in ORC files.

Overall, ORC offers excellent performance but limited interoperability. We recommend choosing ORC when you are using features of Hive that require it. Keep in mind that the Hive tables that use ORC files can not be queried using Impala.

## Other File Types

Big data systems sometimes use other file formats in addition to text, Avro, Parquet, and ORC. Two common options are ***Sequence-Files*** and ***RC-Files***. We generally do not recommend using these file formats, but they are briefly described below so that you will be aware of them.

## Sequence Files

The SequenceFile format was developed for big data systems as an alternative to text files. SequenceFiles store key-value pairs in a binary container format. They store data more efficiently than text files, and they can store binary data like images.

However, the SequenceFile format is closely associated with the Java programming language, and it is not widely supported outside the Hadoop ecosystem.

Overall, SequenceFiles offer good performance but poor interoperability.

## RCFiles

RCFile, which stands for Record Columnar File, is a columnar file format that was developed for use with Hive. RCFile is also supported by some other tools, including Impala, but this support is limited. The RCFile format stores all data as strings, which is inefficient.

Overall, RCFile offers poor performance and limited interoperability.

The ORC file format (described in the previous reading) is an improved version of RCFile with superior performance.

## WORKING WITH FILE TYPES

With so many different file formats, you might wonder which one will work best for your data and use case. There are several factors to consider when choosing a file format. For example, what's the ingest pattern? In other words, how is the data loaded? Will the data set be loaded all at once, or will new records be added continually? If the data is loaded all at once, a columnar file format like Parquet might be able to take advantage of repeated patterns in the data, to store it more efficiently.

But if it's being loaded in very small batches, Parquet can be less efficient than other file formats. Another factor is what tools you'll use to work with the data. This defines how much interoperability you'll need. For example, will you be using MapReduce, Hive, Impala, Spark or any other tools? What formats do those tools support? In general, text files and Parquet files offer the best interoperability. When you use these file formats, you'll have more flexibility in how you can process the data.

However, if you're using only Hive, not Impala, then the ORC file format might be a better choice because there are some features of Hive that require the data to be in ORC files. Another question is, what's the expected lifetime of the data? Is it temporary, such as an input to a job that will be deleted after the job completes? Or will you need to retain the data and be able to read it years from now? If the data is being stored for a long time then you should choose a file format with good interoperability. Because years from now you might not be using the same tools that you're using today. Also for long-term data storage, consider choosing a file format that stores the schema information in the data files.

That way you'll always be able to determine the names and data types of the columns in the data files. And consider choosing a file format that supports schema evolution so you can add, remove or modify columns in a Hive or Impala table without needing to make changes to the data files themselves. Generally, Avro and Parquet are good choices for long-term data storage. Somewhat related is the question, what are your requirements for data size and query performance? If you're planning to store a huge amount of data and you need to correlate efficiently, then you should choose a compressed file format that's optimized for query performances.

Parquet files are typically a good choice for this. The bottom line is there was no single file format that is best in all cases. The best format depends on your data and what you're doing with it. Use these considerations along with how each format works to inform your decision.

## CHOOSING THE RIGHT FILE TYPES

With so many different file formats, you might wonder which one will work best for your data and use case. There are several factors to consider when choosing a file format. For example, what's the ingest pattern? In other words, how is the data loaded? Will the data set be loaded all at once, or will new records be added continually? If the data is loaded all at once, a columnar file format like Parquet might be able to take advantage of repeated patterns in the data, to store it more efficiently. But if it's being loaded in very small batches, Parquet can be less efficient than other file formats.

Another factor is what tools you'll use to work with the data. This defines how much interoperability you'll need. For example, will you be using MapReduce, Hive, Impala, Spark or any other tools? What formats do those tools support? In general, text files and Parquet files offer the best interoperability. When you use these file formats, you'll have more flexibility in how you can process the data. However, if you're using only Hive, not Impala, then the ORC file format might be a better choice because there are some features of Hive that require the data to be in ORC files. Another question is, what's the expected lifetime of the data? Is it temporary, such as an input to a job that will be deleted after the job completes?

Or will you need to retain the data and be able to read it years from now? If the data is being stored for a long time then you should choose a file format with good interoperability. Because years from now you might not be using the same tools that you're using today. Also for long-term data storage, consider choosing a file format that stores the schema information in the data files. That way you'll always be able to determine the names and data types of the columns in the data files.

And consider choosing a file format that supports schema evolution so you can add, remove or modify columns in a Hive or Impala table without needing to

make changes to the data files themselves. Generally, Avro and Parquet are good choices for long-term data storage. Somewhat related is the question, what are your requirements for data size and query performance? If you're planning to store a huge amount of data and you need to correlate efficiently, then you should choose a compressed file format that's optimized for query performances. Parquet files are typically a good choice for this. The bottom line is there was no single file format that is best in all cases. The best format depends on your data and what you're doing with it. Use these considerations along with how each format works to inform your decision.

## CREATING TABLES WITH AVRO AND PARQUET FILES

When you create tables using files in Apache Avro or Apache Parquet formats, there are additional options specific to those file formats.

## APACHE AVRO

To create an Avro table (that is, a table that will use the Avro format) specify **STORED AS AVRO**.

Avro embeds a schema definition (column names and their data types) in the file itself, but you still must provide a schema definition in your **CREATE TABLE** statement. There are three ways to do this:

1. The usual way of specifying columns and their data types
2. Specifying an Avro schema file
3. Providing the schema as a JSON literal string

To create an Avro table in the usual way, provide the name and data type of each column, in parentheses, with the name-type pairs separated by commas:

```
CREATE TABLE company_email_avro (id INT, name STRING, email STRING)

  STORED AS AVRO;
```

You can avoid listing the full schema in the **CREATE TABLE** statement by storing the table schema in a separate Avro schema file and linking to it as a table property. In your **CREATE TABLE** statement, specify **TBLPROPERTIES**

('avro.schema.url'='/*path*/*to*/*file*.avsc'); where *file*.avsc is a JSON file containing an Avro schema.

For example, you can create the same table as above using order_details.avsc with these contents:

```
{"type":"record",
        "name":"company_email_avro",
        "fields":[
                {"name":"id","type":["null","int"]},
                {"name":"name","type":["null","string"]},
                {"name":"email","type":["null","string"]}
        ]}
```

(The instances of "null" preceding the names of the data types in this schema indicate that the columns can contain NULL values.)

Then the CREATE TABLE statement could look like this:

```
CREATE TABLE company_email_avro
  STORED AS AVRO
  TBLPROPERTIES ('avro.schema.url'='/path/to/company_email.avsc');
```

The URL path can point to an Avro schema file in HDFS, as shown above. You can also point to other locations, such as an HTTP server using the http:// protocol or an S3 bucket using the appropriate protocol for your configuration, such as s3a://.

Instead of using the Avro schema file, you can put the JSON for the schema into the CREATE TABLE statement as a table property using 'avro.schema.literal'. The following will also create the same table as the previous examples:

```
CREATE TABLE company_email_avro
  STORED AS AVRO
  TBLPROPERTIES ('avro.schema.literal'=
   '{"type":"record",
     "name":"company_email_avro",
      "fields":[
                {"name":"id","type":["null","int"]},
                {"name":"name","type":["null","string"]},
                {"name":"email","type":["null","string"]}
     ]}');
```

If the JSON schema of an Avro data file is not available to you, you can extract it from the Avro data file by using the **avro-tools** command-line utility. The syntax is:

**avro-tools getschema** ***/path/to/avro_data_file.avro***

This command extracts the schema from the specified Avro data file and prints the JSON representation of the schema to the screen. You could then copy and paste this JSON into the **TBLPROPERTIES** clause as described above, or save this JSON to a plain text file using the **.avsc** extension then point to that file in the **TBLPROPERTIES** clause, as previously described. This **avro-tools** utility is installed on the course VM.

## Apache Parquet

Like with Avro tables, you can create Parquet tables by specifying **STORED AS PARQUET**. You saw this in the Week 2 reading about the **STORED AS** clause.

Parquet files also have the schema embedded in the file, but the table definition also needs to include the schema definition, for example:

```
CREATE TABLE investors_parquet

    (name STRING, amount INT, share DECIMAL(4,3))

  STORED AS PARQUET;
```

With Parquet files, though, Impala ***(not Hive)***provides a shortcut: **LIKE PARQUET '*/path/to/file.parq*'**. By using this shortcut, you can take the schema directly from a Parquet file—no need for a separate schema file as Avro requires:

```
CREATE TABLE investors_parquet

    LIKE PARQUET '/user/hive/warehouse/investors_parquet/investors.parq'

  STORED AS PARQUET;
```

Note that in this case, the table being created will use the specified file as one of its data files, because the table's file location will be **/user/hive/warehouse/investors_parquet** and the referenced Parquet file is in that location. This does ***not*** need to be the case; you can use a file that exists elsewhere than where the table's files will be stored.

Note also that this is not the same as cloning a table using **LIKE *tablename***.(See the "CREATE TABLE Shortcuts" reading from Week 2.)
Using **LIKE *tablename*** points to an existing table and copies the schema information for that table from the metastore; using **LIKE PARQUET '*/path/to/file.parq*'** points to the path of a specific Parquet file and copies the schema information from that file. You ***may*** use the Parquet file for an existing table, but in that case, it's probably better to use **LIKE *tablename***.

Cloudera's documentation for **CREATE TABLE**includes the following notes about using **LIKE PARQUET**:

- Each column in the new table has a comment stating that the SQL column type was inferred from a Parquet file.

- The file format of the new table defaults to text, as with other kinds of **CREATE TABLE** statements. To make the new table also use Parquet format, include the clause **STORED AS PARQUET** in the **CREATE TABLE LIKE PARQUET** statement.
- If the Parquet data file comes from an existing Impala table, currently, any **TINYINT** or **SMALLINT** columns are turned into **INT** columns in the new table. Internally, Parquet stores such values as 32-bit integers.

Try creating the Avro tables in all three ways using the following commands, dropping the table each time before recreating it a different way. ***(Be sure you use the EXTERNAL keyword so you don't delete the file when you drop the table.)***

1. Use the usual way:

```
CREATE EXTERNAL TABLE company_email_avro

    (id INT, name STRING, email STRING)

  STORED AS AVRO

  LOCATION 's3a://training-coursera2/company_email_avro/';
```

Use the **DESCRIBE** command to inspect the schema, then check sample data using your favorite method to do that (such as the **Sample** from the data source panel, or running a **SELECT *** query). Drop the table when you're done.

2. Use the JSON schema file at **s3a://training-coursera2/company_email_avro.avsc**. First take a look at the contents of that file (using a **hdfs dfs -cat** or **aws s3 cp** command for S3 content—you can't use Hue for this), then run this command:

```
CREATE EXTERNAL TABLE company_email_avro

  STORED AS AVRO

  LOCATION 's3a://training-coursera2/company_email_avro/'

  TBLPROPERTIES ('avro.schema.url='
```

```
s3a://training-coursera2/company_email_avro.avsc');
```

Use the **DESCRIBE** command to inspect the schema, then check sample data using your favorite method to do that (such as the **Sample** from the data source panel, or running a **SELECT *** query). Drop the table when you're done.

3. Finally, use the JSON literal:

```
CREATE EXTERNAL TABLE company_email_avro

  STORED AS AVRO

  LOCATION 's3a://training-coursera2/company_email_avro/'

  TBLPROPERTIES ('avro.schema.literal'=

'{"type":"record",

  "name":"company_email_avro",

   "fields":[

     {"name":"id","type":["null","int"]},

     {"name":"name","type":["null","string"]},

     {"name":"email","type":["null","string"]}

   ]}');
```

Use the **DESCRIBE** command to inspect the schema, then check the sample data using your favorite method to do that (such as the **Sample** from the data source panel, or running a **SELECT *** query). Drop the table when you're done.

4. You probably created a Parquet table already, the **investors_parquet** table in the reading “The STORED AS Clause” in Week 2. If you dropped that table, or didn't create it, go back and do so now. (It was the last step.) Then use **LIKE PARQUET** as directed above to create a new table called **investors_parquet_archive**. (Note that you need to specify the Parquet

data file, not the existing table.) Use **DESCRIBE** to check the schema, then check that this cloned table has no data. Keep the table, you will use it again later.

# WEEK 4

## MANAGING DATASETS IN CLUSTERS AND CLOUD STORAGE

Learning Objectives

- Manage data in HDFS, with Hue and with the command line
- Manage data in AWS S3 with the command line
- Use Sqoop to transfer data between RDBMSs and big data systems
- Load data into tables using Hive and Impala SQL statements

## INTRODUCTION

### Week 4 Introduction

Welcome to week 4 of managing big data in clusters and Cloud storage. This week of the course is all about managing data sets. Remember that in Hadoop-based clusters are platforms, table data is stored separately from table metadata. The metadata for a table, which includes the table structure or schema is stored in the metastore. You can create and manage the metadata part of a table by using SQL statements like CREATE TABLE and ALTER TABLE. You learned all about those statements in the previous two weeks of this course. The data for a table is stored separately in a file system like HDFS or S3. In most of the examples in the earlier weeks of this course, the data was already loaded into the file system for you and the task at hand was to define the metadata so that you could query those data files with Hive and Impala. But in the real world, the data will not always be preloaded for you. So in this week of the course, you'll learn how to load data into HDFS and S3.

There is more than one way to do this. So the content in this week is broken into four main lessons. First, we'll show how to load data files into HDFS, the Hadoop Distributed File System. HDSF is where Hive and Impala store table data by

default. You can load data into HDFS using Hue or using the command line. We'll show you how to do both. Next, we'll show how to load data files into Amazon S3, the most popular cloud storage platform. You can also do this using Hue but we'll focus on the command line. We'll show two different ways to do it from the command line. Sometimes the source data you want to load is not in files, instead it's in a table in a relational database system like MySQL or PostgreSQL. There is a tool called Sqoop.

They can move data between relational database systems and HDFS or S3, we'll show you how to use that. Finally, we'll show how Hive and Impala themselves can be used to move data into tables by running SQL commands. We'll introduce several SQL statements that you can use to do this and we'll describe how Hive and Impala typically require a different approach to loading data than relational database systems. After this week of the course, you'll have all the fundamental skills you need to create Hive and Impala tables, load data into them, and manage those tables and data making changes as needed.

---

## REFRESH IMPALA'S METADATA CACHE AFTER LOADING DATA

Recall that Impala's fast query performance comes in part from using a Metadata Cache in computer memory. So it doesn't have to go to the Metastore to look up the metadata for a table every time a query is executed. Impala automatically updates or refreshes its Metadata Cache when you make changes with Impala. But you have to tell it to refresh the metadata when you make changes outside of Impala. For example, if you want to alter the order of columns using an Alter Table statement with after or first, you have to do this using Hive because Impala does not support these after our first key word. If you immediately try to sub-query the table in Impala, it will still use the old schema. So in this case, you need to tell Impala to update its Metadata Cache. To do this, run a refreshed command with Impala specifying the name of the table you've changed. Then, Impala's Metadata Cache will be updated and it will use the new table schema.

Now in this week's materials, you won't be changing a table's column information. You will be loading data into the tables, which involves working with the files that hold the data. Impala's Metadata Cache, also stores information about these files, such as where they are located and how many they are. There are different ways to load the data into a table directory and you don't have to use Impala to

do it.  If you load data into a table directory from outside of Impala, Impala will not be aware of this new data.

You might query a table and get a result with no data or with incomplete data.  To avoid this, you need to run this same refreshed command in Impala, whenever you load new data into a table from outside of Impala. That way, Impala's Metadata Cache will know that the new data is there.

## LOADING FILES INTO HDFS

Recall from the earlier weeks of this course that Hue includes a table browser that you can use to browse the tables defined in the Metastore, and a file browser that you can use to browse the directories and files in HDFS. You can also use these two interfaces to load data files into a table storage directory in HDFS. In this video, I'll demonstrate how to do this using Hue's table browser. To get to the table browser, click the menu icon in the upper left corner, then under browsers, click tables. Recall that the table browser allows you to create a new table, you start this process by clicking this plus icon.

In this type drop-down menu, if you select the manually option, then in the next step, Hue gives you the option to create an empty table with no data in it yet or to create an externally managed to table to query some existing data that's already in HDFS. In those cases, no data is loaded into HDFS. But if you select the file option, then Hue will create a table and move data into it all in one operation. When I click this path selector, Hue opens this choose a file dialog. Here you have two options. One, you can choose an existing file from somewhere in HDFS.

If you do that, you will move that file out of its current location and into the storage directory for this new table. Two, you can upload a file into this new table storage directory from your local file system. I'll demonstrate this upload option. I'll click upload a file. In my local file system, I'll navigate to slash training, slash training underscore materials, slash analyst, slash data. Here I'll select the file castles.csv, and click open. Hue then upload this file to a temporary location in HDFS.

In the format section below, you can specify the field separator. In other words, the delimiter. The record separator, that's the character that marks the end of a

record, and to the quote character. It's common to need to change the field separator. For the other two, you can usually keep the default values. This file castles.csv does have a header row giving the column names. So I'll check this has header checkbox. You can see a preview of the data as it will look when you query this table. You can use this preview to confirm that the settings in the format section are correct.

It looks correct to me, so I'll click next. In this step, I can give a name to the table and specify what database it should be in. This name field is already pre-populated with default.castles, meaning that the table will be named castles, and it will be in the default database. I'll keep this as is. Next, I can specify some more properties. If the data was in some other format like Parquet or Avro, I could use this format to drop down to specify that. I'll keep it set to text. I'll keep this checkbox selected. That way, this table storage directory will be a subdirectory named castles in the Hive warehouse directory, slash users, slash Hive, slash warehouse. The uploaded data will go into that directory in HDFS. I'll leave these other properties untouched.

In the fields section below, you can specify the name and the data type for each column in this table. These fields are pre-populated using the names from the header row in the file I uploaded, and Hue's best guesses about the data types of the columns. You should take a careful look at these to make sure Hue has guessed correctly. Take into account what you know about the data, and what you learned about the different datatypes in the previous week of this course. In this case, it looks good.

So I'll click submit. Hue then creates an entry for this table in the Metastore, creates the storage directory for the table in HDFS, and moves the file that I uploaded into this storage directory. You can see this file in the storage directory by clicking this location link. Here in the HDFS directory, slash users, slash Hive, slash warehouses, slash castles is the uploaded file. In the query editor, I can update Impala's metadata cache.

In this case, I need to use invalidate metadata castles rather than refreshed castles because castles is a new table. Then, I can query this table. So you can use the table browser in Hue to load data files into the HDFS storage directory of a new table at the time when you create the new table. You can also use the table

browser to load data files into the storage directory of an existing table. I'll demonstrate that now. In the table browser, I'll browse to an existing table. I'll go into the fun database, and in that, the games table. This table is used extensively in the second course in this specialization, it has five rows describing five popular board games.

I'll click the location link to browse this table's storage directory in HDFS. You can see there's just one file there, games.csv containing the data for these five games. I'll click the back button several times to return to the table browser. There is another file in the local file system on the VM named ancient_games.csv. I'd like to add the data in that file to this table without losing the data that's already in this table. To do that, I'll click this import data button. Hue opens this import data dialog.

Here like in the previous example, you have two options. One, you can choose an existing file or directory from somewhere in HDFS. If you do that, you will move the file or directory out of its current location and into the storage directory for this table. Two, you can upload a file into this table storage directory from your local file system. I will again demonstrate this upload option. This time, I'll upload the file ancient_games.csv. After uploading this file, I'll select it here. I will not check this overwrite existing data checkbox.

Checking that box would cause Hue to remove everything in this table directory deleting the existing file games.csv, and that's not what I want to do in this case. I'll click submit and wait for the task to finish. When it's finished, I'll click the location link to browse again to this table storage directory in HDFS. Now, you can see there are two files there; games.csv which was there before, and ancient_games.csv which is new. This file ancient_games.csv contains two records, representing checkers and chess. The columns are exactly the same as in the file games.csv, and notice there are two instances of \N indicating missing values in the third column. In the query editor, I can select the funded database, then update Impala's metadata for the game's table.

The refresh command is sufficient in this case, because the game's table is not a new table but there is new data in it. So this command will cause Impala to notice that new data. Then, I can query this table. In the query results, you can see two new rows representing checkers and chess along with the five old rows. Notice

the null values in the third column, the inventor column, for checkers and chess. No one knows who invented these games. So using Hue's table browser, you can load data files into the storage directory for a new table or an existing table. In the next video, I'll show how you can also load files in HDFS through Hue's file browser.

---

## LOADING FILES INTO HDFS WITH HUE'S TABLE BROWSER

Another way to load files into HDFS is through Hue's File Browser. The file browser lets you load files into any directory in HDFS, not just the storage directories for tables. When you use the file browser, it's up to you to make sure you're loading data into the right place in HDFS. For example, if you want to load some additional data into a table, then you must find out what the storage directory for the table is and browse into that directory and then load the data there.

I'll demonstrate how to do this. I'll start first in the table browser since I want to work with the directory for a particular table. Say, I'd like to add some additional data to the airlines table in the fly database. I'll browse to that table, then click the location link to go to the storage directory for that table. After clicking that link, now I'm in the file browser. You can see the storage directory is /fly/airlines. The data for the tables in the fly database is not stored under the Hive warehouse directory.

To load the data file into this directory, I'll click the Upload button, then choose the files option. I'll click Select files then in the directory /training /training_materials/analyst/data, I'll select the file defunct_airlines.CSV and click open. Then this file is immediately uploaded into this HDFS directory, you can see it here. You can also use the file browser to move or copy files from one directory to another within HDFS. I'll demonstrate this, but first I'll delete this copy of defunct_airlines.CSV.

I'll check the checkbox next to it then click move to trash and click Yes to confirm. I know there is already a copy of this same file in another HDFS directory, it's in /old/airlines. I'll navigate there and here's the file, I'll check the box next to it then under Actions, I'll click Copy, I'll select the destination folder /fly/airlines and click the Copy button. Now, if I go back to /fly/airlines, I can see that the file has been

copied here. Let's see how the addition of this file affects the results of queries on the airlines table.

In the query editor, I'll select the fly database, update in Impala's metadata for the airlines table, then query the table. But there's a problem, it appears that the file we added had a header line containing the column names because a row containing these column names is showing up in the query results. The existing data file that was already in this table storage directory does not have a header line and this table is not configured to skip header lines. I'll use the Assist panel on the left side to navigate to the storage directory for this table and open the file defunct_airlines.CSV. Sure enough, it has a header line. Using Hue, I can edit the file to remove this, I'll save the edited file then I'll return to the query editor, refresh the metadata again and query the table again and now, the problem is resolved.

The Assist panel on the left side which is also called the Data Source panel, can also be used to load files into HDFS. Notice the plus icon in the upper right, you can click this to upload a file into this directory in HDFS. There are fewer options available here, so for most tasks you'll want to use the file browser instead. So to summarize this video and the one before it, there are four ways to use Hue to load data into HDFS. One, you can use the table browser to load data into the storage directory of a new table at the time when you create the new table. Two, you can also use the table browser to load data into the storage directory of an existing table.

Three, you can use the file browser to load data into any HDFS directory and four, you can also load data into any HDFS directory through the Assist panel on the left side but your options there are more limited. Of these four options, the third one, using the file browser, is the most flexible. But it's up to you to make sure you're loading data into the right directory.

Also, whenever you're loading data into a table storage directory, you must ensure that the data files match the table metadata. They should have the same number of columns with values that match the tables datatypes and the files should use the correct delimiter, the correct file format so on. Differences will not necessarily cause Hive or Impala to throw errors but they will cause unexpected and probably unwanted results. Additionally, within one table storage directory,

you generally must ensure that all the files are uniform. They should all have the same file type, they should either all have header rows or none of them should and they should all have the same number of columns separated by the same delimiters.

## LOADING FILES INTO HDFS WITH HUE'S FILE BROWSER

Another way to load files into HDFS is through Hue's File Browser. The file browser lets you load files into any directory in HDFS, not just the storage directories for tables. When you use the file browser, it's up to you to make sure you're loading data into the right place in HDFS. For example, if you want to load some additional data into a table, then you must find out what the storage directory for the table is and browse into that directory and then load the data there. I'll demonstrate how to do this.

I'll start first in the table browser since I want to work with the directory for a particular table. Say, I'd like to add some additional data to the airlines table in the fly database. I'll browse to that table, then click the location link to go to the storage directory for that table. After clicking that link, now I'm in the file browser. You can see the storage directory is /fly/airlines. The data for the tables in the fly database is not stored under the Hive warehouse directory. To load the data file into this directory, I'll click the Upload button, then choose the files option.
I'll click Select files then in the directory /training /training_materials/analyst/data, I'll select the file defunct_airlines.CSV and click open. Then this file is immediately uploaded into this HDFS directory, you can see it here. You can also use the file browser to move or copy files from one directory to another within HDFS. I'll demonstrate this, but first I'll delete this copy of defunct_airlines.CSV. I'll check the checkbox next to it then click move to trash and click Yes to confirm.

I know there is already a copy of this same file in another HDFS directory, it's in /old/airlines. I'll navigate there and here's the file, I'll check the box next to it then under Actions, I'll click Copy, I'll select the destination folder /fly/airlines and click the Copy button. Now, if I go back to /fly/airlines, I can see that the file has been copied here. Let's see how the addition of this file affects the results of queries on

the airlines table. In the query editor, I'll select the fly database, update in Impala's metadata for the airlines table, then query the table.

But there's a problem, it appears that the file we added had a header line containing the column names because a row containing these column names is showing up in the query results. The existing data file that was already in this table storage directory does not have a header line and this table is not configured to skip header lines. I'll use the Assist panel on the left side to navigate to the storage directory for this table and open the file defunct_airlines.CSV. Sure enough, it has a header line.

Using Hue, I can edit the file to remove this, I'll save the edited file then I'll return to the query editor, refresh the metadata again and query the table again and now, the problem is resolved. The Assist panel on the left side which is also called the Data Source panel, can also be used to load files into HDFS. Notice the plus icon in the upper right, you can click this to upload a file into this directory in HDFS. There are fewer options available here, so for most tasks you'll want to use the file browser instead.

So to summarize this video and the one before it, there are four ways to use Hue to load data into HDFS. One, you can use the table browser to load data into the storage directory of a new table at the time when you create the new table. Two, you can also use the table browser to load data into the storage directory of an existing table. Three, you can use the file browser to load data into any HDFS directory and four, you can also load data into any HDFS directory through the Assist panel on the left side but your options there are more limited. Of these four options, the third one, using the file browser, is the most flexible.

But it's up to you to make sure you're loading data into the right directory. Also, whenever you're loading data into a table storage directory, you must ensure that the data files match the table metadata. They should have the same number of columns with values that match the tables datatypes and the files should use the correct delimiter, the correct file format so on. Differences will not necessarily cause Hive or Impala to throw errors but they will cause unexpected and probably unwanted results. Additionally, within one table storage directory, you generally must ensure that all the files are uniform. They should all have the same file type,

they should either all have header rows or none of them should and they should all have the same number of columns separated by the same delimiters.

## Loading Files into HDFS from the Command Line

In the previous two videos, you learned how to use Hue to load files into HDFS. In this video, you'll learn how you can also load files into HDFS from the command line. Recall from earlier in this course, some of the reasons why you might want to use the command line instead of Hue. Entering commands is a more systematic way of accomplishing a task. If you can perform a task on the command line, that means that you can also script it, automate it, schedule it, and more.

By saving your commands in a file, you can effectively document the steps you performed and make a task reproducible. In the first week of this course, you learned that to interact with HDFS on the command line, you used the command hdfs dfs followed by various commands, and options, and arguments. These are called HDFS shell commands or Hadoop file system shell commands. You learned how to use a few of these commands to browse files in HDFS.

There's ls to list the contents of a directory in HDFS, cat, to print the contents of a file to the screen, and get, to download a file from HDFS to your local file system. However, we did not yet introduced the commands to load files into HDFS or to manage files in HDFS. I'll demonstrate those now. Recall that in the first video in this lesson, I used Hue to add a data file named ancient_games.csv to the storage directory for the games table in the fun database. Let's use an HDFS shell command to list the contents of that table storage directory. I'll enter the command hdfs dfs dash ls, followed by the path to the storage directory for the games table in the fun database. That directory is in the usual place under the Hive warehouse directory.

So the path is slash user, slash hive, slash warehouse, slash fun.db, slash games. From the results, you can see that the file ancient_games.csv is still there, and so is the file games.csv that was originally there. I want to demonstrate how to load the file ancient_games.csv there again, this time using an HDFS shell command. But I can't do that if there's already a copy there. So I'll first run a command to delete this file. The delete command is hdfs dfs dash rm, for remove, followed by

the path to the file to delete. In this case, that's slash user, slash hive, slash warehouse, slash fun.db, slash games, slash ancient_games.csv. After I press enter, HDFS deletes that file.

Just to check this, I'll press the up arrow key twice and press enter to run the ls command again, and the listing shows that the file is no longer there. So now I can run a command to load this file here again. There is actually a copy of this file ancient_games.csv already in HDFS in a different directory. So I'll first show you how you can copy a file from one directory to another in HDFS. The command to do this is hdfs dfs dash cp for copy. You first specify the source path then the destination path.

I know there's a copy of this file in slash old, slash games. So for the source, I'll specify slash old, slash games, slash ancient_games.csv. For the destination, I'll specify the directory slash user, slash hive, slash warehouse, slash fun.db, slash games. The directory path is sufficient, there's no need to specify the destination file name unless you want it to be different than the source file name. After I press enter, HDFS copies these file from the source to the destination. I'll run the ls command again to see it.

If you want to move the file instead of copying it, you can use mv instead of cp. When you run an mv command, the file will no longer exist in the source directory after the command is executed. These three HDFS shell commands rm, cp, and mv are named after the analogous Linux shell commands. Like their Linux analogs, these commands support wildcard characters in paths, which enables them to operate on multiple files at once. For example, I can delete all the files in the storage directory for the games table by running an hdfs dfs dash rm command with slash star.

That's an asterisk, after the directory path. Now, all the files in that directory are deleted. I'll run an ls command again to see this. The last command I'll describe in this video is the put command, which uploads files from your local file system to HDFS. The syntax is hdfs dfs dash put, then the source path and the destination path. You can use wildcard characters in the source path with this command too.

For example, I know that in the folder slash home, slash training, slash training_materials, slash analyst, slash data, there are copies of both of the files

containing games data, games.csv and ancient_games.csv. So I'll use the filename star games.csv. In this case, the star or the asterisk represents zero or more of any character. So this will match both of those filenames. For the destination path, I'll specify the storage directory for the games table. When I run this command, both of these files are uploaded to HDFS.

Once again, I'll run an ls command and you can see them both in the listing. Remember that anytime you change the files in a table storage directory, you should refresh Impala's metadata cache. I'll do that now in Hue. In the Impala query editor, I'll run the command refresh games, then I can run queries on the games table. So to review, in this video you learned about four HDFS shell commands that you can use to manage files; rm, to delete files in HDFS, put to upload files from your local file system to HDFS, cp to copy files from one directory to another within HDFS, and mv to move files from one directory to another within HDFS. With these commands plus knowledge of which HDFS directories contain the data files for which tables, you'll be able to manage the data in Hive and Impala tables from the command line.

---

## More about HDFS Shell Commands

This reading supplies some additional information about HDFS and the Hadoop File System Shell that you should know.

---

## The -mkdir Command

One useful command that was not mentioned in the previous video is **hdfs dfs -mkdir**, which can be used to create one or more directories in HDFS. This command expects that the parent directory of the directory you are creating already exists, so if you want to use this command to create nested directories, start first by creating the highest-level parent directory, then create the next-level child subdirectory, and so on. Alternatively, you can use the **-p** option (short for ***parent***) to automatically create any necessary parent directories of the specified directory if they do not already exist.

## More HDFS Command Options

Some of the commands that you did learn about in this course can take additional command-line options that were not described. For example, when using the when using **hdfs dfs -rm** command, you can specify the **-r** option (short for ***recursive***) to delete the specified directory and all files and subdirectories under it. The syntax is:

**hdfs dfs -rm -r *[/path/to/directory/]***

Be extremely careful when using this **-r** option! It is easy to inadvertently delete a huge amount of data by misspecifying the directory path

## Full List of HDFS Shell Commands, Options, and Arguments

You can see a complete list of the Hadoop File System Shell commands and the options and arguments they accept on this web page: https://hadoop.apache.org/docs/stable/hadoop-project-dist/hadoop-common/FileSystemShell.html. Many of these you will never need to use, but this is a good resource to find ways to do things you do need. As you look through that page, remember that **hdfs dfs** is the same as **hadoop fs**—you can use either.

Some of the commands on that page relate to permissions and file ownership, which is beyond the scope of this course, but see the "HDFS Permissions" reading if you're interested to learn more about that.

## HDFS Paths

In this course, most of the HDFS paths you will see in **hdfs dfs** commands are in the form ***/path/to/directory/*** (for a directory), or ***/path/to/directory/file.ext*** (for a file). This is known as an ***unqualified path*** because it does not specify what ***protocol*** to use and it does not specify what specific instance of HDFS to use. In most cases, it is sufficient to use unqualified HDFS paths, because the configuration of your big data environment will specify what protocol and what HDFS instance to use.

However, in some cases, depending on the configuration of your big data environment, it might be necessary to ***fully qualify*** your HDFS paths by specifying the **hdfs://** protocol and a hostname indicating what instance of HDFS to use. To do this, use the form: **hdfs://*hostname/path/to/directory/file.ext*** or **hdfs://*hostname.domain/path/to/directory/file.ext***. Ask your system administrator what hostname and what domain (if any) to use.

You can also specify the **hdfs://** protocol without specifying a hostname, using the form: **hdfs:///*path/to/directory/file.ext***. Notice the ***three*** slashes after **hdfs:** The first two slashes are part of the protocol, and the third slash is the start of the path.

## HDFS TRASH

Note that HDFS has a "trash" directory for recently deleted files, similar to the Trash on iOS or Windows systems. This trash is not always enabled, so you should check your system to see if it's enabled before assuming that you can recover any deleted files!

The **hdfs dfs -rm** command has a **-skipTrash** option that you can use to bypass the trash (if it's enabled) and delete the file immediately. When deleting large files to free up space in HDFS, consider using this option so that you do not need to perform the additional step of emptying the trash. But remember that when you use this option, the files you delete will not be recoverable.

## CHAINING AND SCRIPTING WITH HDFS COMMANDS

When you work a lot with HDFS commands, there are two techniques that you will likely find very useful: chaining and scripting.

### CHAINING HDFS COMMANDS

If you've worked much with the command line, you might be familiar with using pipes (**|**) to chain commands, or using redirection to push results to a different

location (such as to a file instead of to the screen). You can do this with HDFS shell commands as well.

You can achieve many practical tasks by combining HDFS shell commands with piping or redirection. Two examples are described below.

**Viewing the First Few Lines of a File**

To view only the first few lines of a text file stored in HDFS, you can pipe the output of the **hdfs dfs -cat** command to the **head** command:

```
$ hdfs dfs -cat /path/to/file.txt | head
```

You can ignore the message that says "Unable to write to the output stream." This happens because the **hdfs dfs -cat** command outputs more data than the head command inputs.

## Loading Data Without the Header Line

Instead of using the **skip.header.line.count** table property to ignore the header line in text files, you can copy everything ***except*** the header line when you put a file into HDFS:

```
$ tail -n +2 source_file.txt | hdfs dfs -put - /path/to/destination_file.txt
```

In the above command, the hyphen or dash character (**-**) after **-put** tells the HDFS shell application to take the output of the **tail** command before the pipe and load that into HDFS.

To remove the header line when copying a text file in the ***opposite*** direction (from HDFS to the local file system) you would run a different command, this time using the output redirection operator (**>**) to store the output of the **tail** command in a file:

```
$ hdfs dfs -cat /path/to/source_file.txt | tail -n +2 > destination_file.txt
```

## Using Commands in Scripts

You can also use commands like this in scripts. This is particularly helpful when you automate tasks that involve managing files in HDFS. In shell scripts, you can use **hdfs dfs** commands just as you normally would other shell commands. Scripts for other languages, such as Python and R, typically can invoke shell commands using commands specific to the language. For example, in Python you can use the **os** or **subprocess** modules and use a call such as **subprocess.call()**. In R, you can use **system()**.

## HDFS PERMISSIONS

Security is an important consideration for any computer system, and big data systems are no exception. In the VM for this course, you have all the permissions you need to do the exercises and examples in this course, but at your company, you may not be able to do everything you're learning here. For example, the ability to create databases, or even tables, is often restricted so a company's system doesn't get overloaded with redundant or otherwise unnecessary items. Similarly, the ability to drop tables, or to delete or move files, may be restricted. Larger companies will only allow their data engineers, database administrators, or system administrators to do this work.

A deep look at permission issues is beyond the scope of this course, but the following resources can provide additional information, if you're interested:

1. DWGeek's Hadoop Security – Hadoop HDFS File Permissions is a quick overview of file permissions. This is a good place to start, and then you can decide if you want to dig deeper.
2. HDFS Commands, HDFS Permissions and HDFS Storage is a chapter from a book, ***Expert Hadoop Administration: Managing, Tuning, and Securing Spark, YARN, and HDFS***. The section "HDFS Users and Superusers" may be helpful if you are unfamiliar with the concept of a superuser.
3. The documentation on Hadoop includes the HDFS Permissions Guide, which likely has more detail than you'll ever want, but it's a great resource if there's a particular question you're looking to answer.

## LOADING FILES INTO S3 FROM THE COMMAND LINE

In the previous lesson, you learned how to load files into HDFS. In this lesson, you'll learn how to load files into Amazon S3. The most popular cloud storage platform. Of course, loading files into an S3 bucket requires having right access to that bucket. Unfortunately, we cannot give all Coursera learners right access to the S3 buckets we're using for this course. We can only imagine what some of you might do if we gave you right access.

So although this is an important topic, you will not be able to practice it on the Course VM. One way to load files into S3 buckets is through Hue. If you have right access to an S3 bucket, then you can use the file browser and other interfaces in Hue to interact with S3, both reading and writing. You can load files in the same way you would load them into HDFS, but you would choose the S3 bucket as the destination rather than an HDFS directory. Another way is to use the command line, that's what I'll describe in the rest of this video.

Recall from earlier in the course that many of the Hadoop file system shell commands that you can use to interact with HDFS also work for S3 if you specify the path as an S3 URL using the s3a protocol. We demonstrated that you can use commands including ls, cat, and the get with S3 just the same way you can with HDFS. The same thing is true of the HDFS shell commands for loading and managing files. Here are some examples. You should recall all of these from the previous lesson, the only difference here is that the paths are S3 paths specified with the s3a protocol.

To delete a file in S3, use hdfs dfs -rm followed by the path to the file to delete. This and the other commands I'll describe work with any type of file, I'm using the file extension.ext just as a generic placeholder. It could be a text file, a parquet file, or any other type of file. To copy a file from one location to another within S3, use hdfs dfs -cp followed by the source path then the destination path. You must include the s3a protocol at the start of both paths.

You can use this command to copy files within the same bucket or from one bucket to a different bucket. If you omit the filename from the destination path, then the copied file will have the same name as the source file. You can also use this cp command to copy a file from HDFS to S3 or from S3 to HDFS. To do this, use the HDFS protocol in one path and the s3a protocol in the other. The example shown here copies of file from HDFS to S3.

Notice the three slashes after hdfs:. The first two slashes are part of the protocol, hdfs://. The third slash is the start of the HDFS path. If you want to move a file instead of copying it, you can use mv. This works exactly like cp except the file will no longer exist in the source directory after the command is executed. As with cp, you can use this to move a file within S3 or between HDFS and S3 in either direction. To upload a file from your local file system to S3, you can use hdfs dfs -put. For example, the command shown here.

For the source file, you can specify an absolute local file path as in this example or a relative path relative to your current working directory in the local file system. Once again, if you omit the file name from the destination path, then the uploaded file will have the same name as the source file. With all these commands, it's possible to use wildcard characters like the asterisk in the source paths to operate on multiple files at once.

The last HDFS shell command I'll describe is mkdir, you can use this to create a subdirectory within an S3 bucket. You can optionally specify two or more directory paths separated with spaces to create multiple directories with one command. To run these HDFS shell commands, you need the HDFS shell application to be installed on the computer you're using. That's true regardless of whether you're interacting with files in HDFS or in S3, but it can be impractical to install and configure the HDFS shell application. So there is another way. Recall from earlier in the course that Amazon provides a tool called the AWS Command Line Interface or the AWS CLI.

As we demonstrated earlier, you can use this tool to list the contents of an S3 bucket, download a file from S3, and so on. You can also use this tool to load files into S3 and to manage files in S3. To you use this tool to interact with S3, you run commands that begin with aws s3. For most of the hdfs dfs commands I described earlier in this video, there is an equivalent aws s3 command. To delete a file in S3,

use aws s3 rm followed by the path to the file to delete. There is no dash before rm. Notice that the protocol before the bucketname is s3://. With these aws s3 commands, the protocol is S3 not s3a.

To copy a file from one location to another within S3, use aws s3 cp followed by the source path then the destination path. You must include the S3 protocol at the start of both paths, and this works to copy files within a bucket or between two buckets. To upload a file from your local file system to S3, you also use this cp command. You specify a local file path for the source and an S3 path for the destination. This is different from the HDFS shell which requires a separate command "put" to upload files.

You might recall from an earlier video that you can also use this aws s3 cp command to download files from S3 to your local file system instead of using the HDFS shells "get" command. Finally, you can use mv instead of cp to move a file, so the file will no longer exist in the source directory after the command is executed. This works for moving files within S3 and between S3 and your local file system.

Note that there is no AWS CLI command that's equivalent to the mkdir command in the HDFS shell, so it's not directly possible to create a subdirectory in an S3 bucket using the AWS CLI. However, when you use the cp or mv commands, any necessary subdirectories will be created for you. You simply run the command to copy or move files to the S3 destination path you want and all the necessary subdirectories are automatically created. The technical reason for this is that in an S3 bucket, there really are no subdirectories. Everything in a bucket is stored in a flat file structure and subdirectories are simulated by using slashes in the filenames.

Some tools like the HDFS shell are designed to hide this technical detail from you. Others like the AWS CLI, do not try to hide it. It is important to note that these AWS CLI commands cannot move files between HDFS and S3. It does not provide access to HDFS only to S3 and to your local file system. Remember that these commands will not work on the Course VM because the VM is not configured to provide right access to any S3 buckets, but these commands will work in a real-world environment where you do have right access to S3 buckets. There are some other AWS S3 commands that I did not introduce in this video but that you might

find useful. To see a full listing of all the available commands, run the command aws s3 help. When you're done, press "Q" to exit the help interface.

## Other Ways to Load Files into S3

There are other ways to connect to S3 and load data, besides what we mentioned so far. The method you use will depend on your company's policy and your own preferences. Feel free to read more about the examples provided through the links here:

- Other command line interfaces (CLIs), such as **s3cmd** (see https://s3tools.org/s3cmd)
- Graphical user interfaces (GUIs), such as CloudBerry Explorer or CyberDuck
- AWS Management Console, a direct interface from Amazon Web Services

## S3 Permissions

As with HDFS, managing permissions is vital with S3—even more so, because as a cloud service, S3 can in many cases be more easily accessed by hackers and other bad actors. There are many stories about S3 buckets being made public and having unwanted consequences:

- There's a Hole in 1,951 Amazon S3 Buckets
- Verizon and Viacom
- Don't panic, Chicago, but an AWS S3 config blunder exposed 1.8 million voter records
- Leaky S3 bucket sloshes deets of thousands with US security clearance

If you use S3, be sure you understand the permissions issues and what is allowed. AWS provides a guide on Setting Bucket and Object Access Permissions.

## MISSING VALUES

**NOTE: Although this reading is in the Cloud Storage lesson, it applies to working with data in HDFS as well.**

When working with character string columns, it is important to remember that an ***empty string***, also called a ***zero-length string***, is ***not*** the same thing as a ***missing value*** (**NULL**).

However, when data is stored in delimited text files by processes external to Hive and Impala, it is common for missing values to be represented as empty strings. For example, your organization might have a data collection process in which first and last names are stored in a CSV file, and some individuals might leave their last names blank. So the rows of the resulting CSV file might look like this:

```
fname,lname
Bert,
Big,Bird
Count,von Count
Ernie,
Two-Headed,Monster
```

Notice the trailing commas in the rows where individuals did not provide a last name.

If you load this CSV file into the storage directory for a Hive and Impala table, the absent values in the **lname** column will be interpreted as empty strings (''), ***not*** as missing values (**NULL**).

This might cause some confusion. For example, an analyst might use Hive or Impala run a query on this table, such as

```
SELECT * FROM names WHERE lname IS NOT NULL;
```

This query would return ***all*** the rows. This might defy the analyst's expectation that it would only return the rows for Big Bird and Count von Count (because the other rows appear to have no last names). But Hive and Impala do not treat these empty strings as **NULL** by default.

However, there is a table property you can set to make Hive and Impala interpret empty strings in table data files as missing values:
the **serialization.null.format** property.

You can set this property in a **CREATE TABLE** by using this **TBLPROPERTIES** clause:

```
TBLPROPERTIES ('serialization.null.format' = '')
```

Notice how the value of this property is set to **''**(an empty or zero-length string).

You can also set this property on an existing table by using an **ALTER TABLE** statement. For example:

```
ALTER TABLE names SET TBLPROPERTIES ('serialization.null.format' = '');
```

If you set this table property to an empty string (**''**), then Hive and Impala queries on that table will treat empty strings in character string columns as if they are true missing values (**NULL**).

The default value of this **serialization.null.format** is **'\N'**. This is a two-character-long literal string consisting of a backslash followed by the capital letter N; it is not a special character.

The **serialization.null.format**property also determines how **NULL** values are represented in the table data files when data is ***inserted*** into the table using Hive or Impala. It affects how missing values are stored for columns of ***all*** data types (not only character string types).

For a particular table, you should decide whether missing values should be represented as **\N** or as an empty string in the files for that table. You should never mix different representations in the files for a single table.

As a best practice, we recommend processing text-based data to ensure that the values you want interpreted as **NULL** by Hive and Impala are represented as **\N** in

the data files ***before*** you store the files in the table directory. But in situations where this is not practical, the method described above is an effective workaround.

The above information may make more sense when you see it in action, so try the following steps using Hue.

1. To see an example of a data file that contains this value **\N**, look at the contents of the file **/user/hive/warehouse/offices/offices.txt** in HDFS. This is the file containing the data in the **offices** table in the **default** database. Notice the **\N** in the fourth row and third column of this file, which represents a missing value of **state_province** for the office located in Singapore. Query the **offices** table with Hive or Impala, and notice that this value appears as **NULL** in the query result.

For the rest of this section, you will create a table and see the effect of setting the **serialization.null.format** property.

2. Using either Hive or Impala, create a table named **names** by running the following **CREATE TABLE** statement:

```
CREATE TABLE names (fname STRING, lname STRING)

  ROW FORMAT DELIMITED

    FIELDS TERMINATED BY ',';
```

3. In the table's directory in HDFS (**/user/hive/warehouse/names/**), create a text file and store the following five lines in it:

```
Bert,

Big,Bird

Count,von Count

Ernie,

Two-Headed,Monster
```

You can do this with Hue's File Browser or with an **hdfs dfs** command.

4. If you're using Impala, run the following command to force Impala to refresh its file metadata for this table:

**REFRESH names;**

5. Run the following SQL query on this table:

**SELECT * FROM names WHERE lname IS NOT NULL;**

Notice how the result set includes ***all*** the rows, including the rows in which **lname** is empty.

6. Now run the following statement to tell Hive and Impala to treat empty strings as missing values for this table:

**ALTER TABLE names SET TBLPROPERTIES ('serialization.null.format' = '');**

If you're using Impala, refresh the metadata for this table again.

7. Run the SQL query again to show rows for which **lname IS NOT NULL**. This time, notice how the rows with empty last names are ***not*** returned in the result set.

8. Finally, drop this **names** table, because you won't need it again.

---

## CHARACTER SETS

**NOTE: Although this reading is in the Cloud Storage lesson, it applies to working with data in HDFS as well.**

As you might know, computer systems use defined ***character sets,*** which are collections of characters, for what are allowable characters. Examples include Unicode, ASCII, Extended ASCII, and various ISO sets.

Compatibility between the character sets your data files use and the character set your system uses is an important consideration. Incompatible characters may appear oddly, or they might even cause the system to throw errors on reading.

If you're familiar with SQL on RDBMSs, you might have seen the **CHARACTER SET**clause in the **LOAD DATA** statement, which lets you specify the character set when you load the data. Hive and Impala do not have such a clause in their **LOAD DATA** statements.

Cloudera's documentation for Impala says this about the character sets used by Impala:

> For full support in all Impala subsystems, restrict string values to the ASCII character set. Although some UTF-8 character data can be stored in Impala and retrieved through queries, UTF-8 strings containing non-ASCII characters are not guaranteed to work properly in combination with many SQL aspects....
>
> For any national language aspects such as collation order or interpreting extended ASCII variants such as ISO-8859-1 or ISO-8859-2 encodings, Impala does not include such metadata with the table definition. If you need to sort, manipulate, or display data depending on those national language characteristics of string data, use logic on the application side.

Hive does have full support for the UTF-8 character set, but no others. From the Hive User FAQ:

> You can use Unicode string on data/comments, but cannot use for database/table/column name.
>
> You can use UTF-8 encoding for Hive data. However, other encodings are not supported (HIVE-7142 introduce encoding for LazySimpleSerDe, however, the implementation is not complete and [does] not address all cases).

Both Hive and Impala also include string functions (such as **encode()** and **decode()** in Hive, and **base64encode()** and **base64decode()** in Impala) that might help for transmitting data in characters other than UTF-8 or ASCII.

Apache Sqoop is an open source tool that was originally created at Cloudera. Its name comes from the contraction of “SQL to Hadoop”; it moves data between a relational database management system (RDBMS) and HDFS. For example, it can import all the tables from a database, just one table, or just part of a table, such as specific columns or specific rows. It can also export data from HDFS to a relational database. You see more about some of these options in the next couple of readings.

Sqoop is a command-line tool offering several commands. The Sqoop **import** command is used to import the data in a single table in an RDBMS to a directory in HDFS. The following example will import all the columns from the **customers** table in the **company** database in MySQL. In the example below, the **$** character represents the operating system shell (terminal) prompt, and the **** (backslash) character is used to continue the command on multiple lines.

```
$sqoop import \
   --connect jdbc:mysql://localhost/company \
   --username jdoe \
   --password bigsecret \
   --table customers
```

This command creates a subdirectory named **customers** in the user’s home directory in HDFS, and populates that subdirectory with files containing all the data from the **customers** table in the RDBMS. By default, Sqoop stores the data in plain text files, where each line of the file is one record from the table and the fields are separated by commas. These defaults can be changed by adding options, which are described in the next reading.

By default, Sqoop uses JDBC to connect to the database. However, depending on the database, there may be a faster, database-specific connector available, which you can use by using the **--direct** option.

If the table whose data you're importing does not have a primary key, then you should specify one using **--split-by *column*.** If you're going to split by a string column, or if the primary key for a table is a string column and you're using some newer versions of Sqoop (as of 1.4.7), include the setting

```
-Dorg.apache.sqoop.splitter.allow_text_splitter=true
```

immediately after the **import** command.

To import ***all*** the tables from a database into HDFS, use the Sqoop **import-all-tables** command. This example brings all the tables from the **company** database into HDFS.

```
$ sqoop import-all-tables \
    --connect jdbc:mysql://localhost/company \
    --username jdoe \
    --password bigsecret
```

The VM has tables in a MySQL database. Although these are already imported into the Hive metastore, do the following to re-import one, just to give you some practice using the Sqoop **import** command.

1. Open a terminal window, if you don't have one already, and execute the following command. The **card_rank** table you're importing doesn't have a primary key, so you need to specify one. The most reasonable one is **rank**, which is a text column, so include the

```
-Dorg.apache.sqoop.splitter.allow_text_splitter=true
```

setting as well.

```
$sqoop import \
    -Dorg.apache.sqoop.splitter.allow_text_splitter=true \
    --connect jdbc:mysql://localhost/mydb \
```

```
--username training \

--password training \

--table card_rank \

--split-by rank
```

2. Check that HDFS now has **/user/training/card_rank**, with at least one file in it.

3. Review one of the files to see how the fields are delimited. (You'll need that when you create the table.)

4. Although you've imported the data, you don't yet have a table for it. (The existing **card_rank** table is in the **fun** database, and its data is in **/user/hive/warehouse/fun.db/card_rank**.) Run a **CREATE TABLE** statement to create a table in the **default** database, **default.card_rank**, with the following columns. Be sure to use **ROW FORMAT** to specify the delimiter, and use a **LOCATION** clause to specify the location of the data as **'/user/training/card_rank/'**.

| name | type |
|---|---|
| rank | STRING |
| value | TINYINT |

5. Run a query on your new **default.card_rank** table or check the sample data from the data source panel in Hue, to see that your table does have the data you imported. ***Note:*** If you created the table in Impala, you should not need to refresh the metadata, because the data was there when you created the table.

6. You can now drop the **default.card_rank** table. (Be careful ***not*** to drop the ***other*** **card_rank** table.)

There are many options for importing tables using Sqoop. A few of the most commonly used ones are given below; for more, see the documentation through https://sqoop.apache.org. Be sure to use the documentation for the version you are using. To see what version of Sqoop you are using, run the command **sqoop version**.

**--target-dir** specifies the HDFS directory **/mydata/customers** as the location the data will be saved to:

```
$sqoop import \
    --connect jdbc:mysql://localhost/company \
    --username jdoe \
    --password bigsecret \
    --table customers \
    --target-dir /mydata/customers
```

**--warehouse-dir** specifies the HDFS parent directory to use for the import; Sqoop will create a sub-directory matching the table's name in the specified parent directory, so this example will also end up creating **/mydata/customers** in HDFS:

```
$sqoop import \
    --connect jdbc:mysql://localhost/company \
    --username jdoe \
    --password bigsecret \
    --table customers \
    --warehouse-dir /mydata
```

**--fields-terminated-by** specifies the delimiter between columns; in this example, **\t** (tab) is the delimiter:

```
$sqoop import \
    --connect jdbc:mysql://localhost/company \
    --username jdoe \
    --password bigsecret \
    --table customers \
    --fields-terminated-by '\t'
```

**--columns** specifies particular columns to import, rather than all of them; here, **--columns** is used to specify the **prod_id**, **name**, and **price** columns of the **products** table, so only these three columns will be imported into HDFS:

```
$sqoop import \
    --connect jdbc:mysql://localhost/company \
    --username jdoe \
    --password bigsecret \
    --table products \
    --columns "prod_id,name,price"
```

**--where** provides a filter to import only specific rows from the table; the following example uses a simple example in which the value in the **price** column must be greater than or equal to 1000, but you may use more complex expressions, including those with **and** and **or**:

```
$sqoop import \
    --connect jdbc:mysql://localhost/company \
    --username jdoe \
    --password bigsecret \
```

```
--table products \
--columns "prod_id,name,price" \
--where "price >= 1000"
```

## MORE SQOOP IMPORT OPTIONS

You not only can import data from an RDBMS into Hadoop, but you can send data the other way as well, using the Sqoop **export** command.

For example, suppose some new product recommendations have been generated after some processing on the Hadoop cluster. These recommendations need to be exported to the web site's back-end database. This can be done with the following command:

```
$ sqoop export \
    --connect jdbc:mysql://localhost/company \
    --username jdoe \
    --password bigsecret \
    --table product_recommendations \
    --export-dir /mydata/recommender_output
```

The **--export-dir** argument specifies where the data to be exported is located in HDFS; in this case, it's in the **/mydata/recommender_output** directory. The destination table in the RDBMS is identified in the **--table** option; in this case, the table will be **product_recommendations**. Note that this only ***exports the data***, it

doesn't ***create the table*** in the RDBMS—the destination table must already exist.

While Sqoop will let you import all tables into HDFS using a single command, it does not have a command to export more than one table. Exporting must be done one table directory at a time.

For more options, see the documentation through https://sqoop.apache.org. Be sure to use the documentation for the version you are using. To see what version of Sqoop you are using, run the command **sqoop version**.

## USING SQOOP TO EXPORT DATA

### USING HIVE AND IMPALA TO LOAD DATA INTO TABLES

If you're familiar with traditional relational Database Systems, this week of the course might have been disorienting for you so far. Because in a traditional RDBMS, the way you load data into a table is usually by running SQL statements, in particular Insert Statements, and Load Data Statements. But in this week so far, we have not used any SQL statements to load data. Instead we've used Hue, and we've used some Shell Commands. To understand why, remember, that in a traditional RDBMS, or a Traditional Data Warehouse, Data Storage is encapsulated by the database software.

This means that the only way to access the data in the tables, is by going through the database software. So if you want to get data out of a table, the only way to do that is by running an SQL select statement, and if you want to put data into a table, the only way to do that is also by running an SQL statement, for statement like Insert, or Load Data.

But with a Modern Data Warehouse, using an engine like Hive or Impala, there are a variety of ways to access the data. Hive and Impala do not encapsulate the data. They don't even store the data. It's stored in a separate system like HDFS, or S3. So to load data into a table, you do not need to go through Hive or Impala. Anytime you'd like you can load the data files directly into the HDFS directory, or

the S3 bucket for that table. With a Hive and Impala you often do not need to load the data into a table at all.

If the data is already somewhere in HDFS, or S3, or one of the other supported file systems, you can create a table to query the data from there. No data needs to move. Even so, sometimes it is useful to have Hive or Impala load the data into a table. Maybe you're familiar with the traditional SQL syntax, for Insert, and Load Data Statements, and it's just easier for you to load data that way, or maybe you're building a data process where every other step is implemented as an SQL statement.

So it's just easier to use an SQL statement to load the data too. Or maybe you want Hive, or Impala to perform some processing on the data as it's loading it into the table storage directory. Well, the good news is that Hive and Impala do support the Insert Statement, and the Load Data Statement. However, there are some important differences in how you use these statements with Hive and Impala, compared to traditional RDBMS's. In this week of the course, we'll show you how to use these, and some other related statements, and we'll describe what's different about them when you're using Hive and Impala.

---

## SQL LOAD DATA STATEMENTS

One way to load data into a table is by running a **LOAD DATA INPATH** statement with Hive or Impala. This moves the specified data files into the table's storage directory in the file system (HDFS or S3).

The example shown below moves the file **sales.txt** from the HDFS directory **/incoming/etl** to the directory for the table named **sales**. Notice that this statement specifies the destination as a table name, not a directory; Hive or Impala uses metadata from the metastore to determine the table's storage directory, and moves the file there.

```
LOAD DATA INPATH '/incoming/etl/sales.txt' INTO TABLE sales;
```

The source path can refer either to a file, as in this example, or to a directory, in which case Hive or Impala will move all the files within that directory into the table.

The **LOAD DATA INPATH** statement *adds* the source files to any existing files that are already in the table's directory—that is, it does not remove existing files in the table's directory, and if there are any filename collisions, then it automatically renames the new files so that no existing files are overwritten. In some cases, you may want to delete all existing data files in the table's directory before loading new data files. To do this, use the **OVERWRITE** keyword, as shown in the example below. This option is useful when you need to do a complete reload of all the data in a table.

```
LOAD DATA INPATH '/incoming/etl/sales.txt' OVERWRITE INTO TABLE sales;
```

The **LOAD DATA INPATH** statement assumes that the files are already somewhere in a file system that is accessible to your instance of Hive or Impala (like HDFS or S3). If they are not, then you first need to load files from your local filesystem into HDFS or S3, for example by running an **hdfs dfs -put** command, or by using the Hue File Browser. Also, this method *moves* the files rather than copying them; the files will no longer exist in the source directory after the statement is executed.

This probably sounds a lot like using **hdfs dfs -mv**—and it is, but there are a couple of advantages to using **LOAD DATA INPATH**.

One is that, if you're running the **LOAD DATA INPATH** statement with Impala, the metadata cache is automatically updated. You do not need to execute a **REFRESH** command; your next query will include the new data.

The other advantage is that since the statement renames any files that are the same as files that already exist within the directory, you don't need to worry about what your files are named. If you use **hdfs dfs -mv** and there is an existing file with the same name, the command will fail and no changes will be made—you'll have to rename one of the files yourself, first.

## SQL INSERT STATEMENTS

SQL **INSERT** statements, common for adding new rows of data to tables in RDBMSs, can be used with Hive and Impala. There are essentially two versions, **INSERT INTO** (which adds files without changing or deleting existing files) and **INSERT OVERWRITE** (which replaces existing files with new files).

If you did the "Try It!" sections in Week 2, you have used them to add a single row of data to a table you created. They can also be used to add multiple rows:

```
INSERT INTO tablename
  VALUES
    (row1col1value,row1col2value, … ),
    (row2col1value,row2col2value, … ),
    … ;
```

This can be helpful for testing a table, such as examining how it stores the data, especially with an unusual or edge case. However, as those exercises noted, in general this is an ***antipattern*** with Hive and Impala, and using it for data ingest in a production environment is a bad practice.

In general, files in HDFS are ***immutable***—they cannot be directly modified. (A file in HDFS can be deleted, and it can be overwritten with a new version of the file, but in general a file in HDFS cannot be directly modified.) So, each time you run an **INSERT** statement, Hive or Impala creates a new file in the table's storage directory to store the new data values given in the statement. So inserting data in small batches like this causes Hive or Impala to create many small files in the table's storage directory. This is a problem.

The ***small files problem*** is a common problem in big data systems. Most of the tools and frameworks that run on a Hadoop cluster are designed to work with ***large*** files, not lots of small files. So if you have a lot of small files (anything less than about 64MB is considered small), operations such as queries in Hive and Impala become inefficient, and query performance is negatively affected. Each **INSERT** command creates a new file, and they will be quite small (for any amount of data that is reasonable to add using an **INSERT** command).

One corrective solution for this, once you find you have a lot of small files, is to rewrite the whole dataset using this command:

```
INSERT OVERWRITE tablename SELECT * FROM tablename;
```

You'll learn more about **INSERT … SELECT** statements in the next reading.

Note that the small files problem can arise from other uses as well. For example, images are typically separate files, and there is no easy way to combine them into one file, so a large number of images will most likely be stored as a large number of files, potentially small ones depending on the resolution and size of the image. The corrective solution described above doesn't really help in that case; there are other ways to work around that problem, but this is beyond the scope of this course. If you're interested in reading more about the small files problem, see the Cloudera blog post, The Small Files Problem.

Note: You might see the syntax **INSERT INTO TABLE** or **INSERT OVERWRITE TABLE**. The **TABLE** keyword is optional. You can include it or not, as you like, but it's helpful to know that both syntaxes are valid, in case you see the one you decide not to use.

You can read more about using **INSERT** in Hive and Impala using these links:

- Hive LanguageManual UDF, Inserting data into Hive Tables from queries
- Cloudera's Impala documentation, INSERT Statement

## SQL INSERT ... SELECT AND CTAS STATEMENTS

As you should know by now, the **SELECT** statement in SQL returns a result set. Typically, you either view this result set, or else store it in a file or in memory on your local computer where you might use it to generate a report or data visualization. However, you can do more with the **SELECT** statement than just view the results or store them locally. The result set from a **SELECT** statement has the same basic structure as a table, so you can use a result set to materialize a new table. As you'll learn in this reading, it is possible to do this with a single command. This command allows you to save the result of a query to a table, so that you can later run another query to analyze or retrieve that result.

The way to do this is by combining together an **INSERT** and a **SELECT** into a single command. Using an **INSERT** statement, you can specify the name of the destination table, followed by a **SELECT** statement. This compound type of statement is known as an **INSERT ... SELECT** statement. Hive or Impala executes the **SELECT** statement then saves the results into the specified table. Note that the destination table must already exist. If you want to replace any existing data, use **INSERT OVERWRITE**; if you want to retain any existing data, use **INSERT INTO**.

For example, you might want to create a new table, **chicago_employees**,that contains only the rows of the **employees** table for employees in the Chicago office (**office_id = 'b'**). After you create this table (perhaps using **LIKE employees** in the **CREATE TABLE** command), the following command will populate the table with the desired rows. Any previously existing records in the **chicago_employees** table are deleted.

```
INSERT OVERWRITE chicago_employees

  SELECT * FROM employees WHERE office_id='b';
```

The line breaks and indentation used in these examples are all optional. The indentation serves to show that the **SELECT** statement is actually part of the **INSERT** statement.

As mentioned above, an **INSERT … SELECT** statement requires that the destination table already exists. But there is a different statement that does not require this: the **CREATE TABLE AS SELECT** (CTAS) statement. A CTAS statement creates a table and populates it with the result of a query, all in one command.

For example, the following CTAS statement creates the **chicago_employees** table and loads the same data as the **INSERT … SELECT** command above, but it does it all in one command.

```
CREATE TABLE chicago_employees AS

  SELECT * FROM employees WHERE office_id='b';
```

CTAS effectively combines a **CREATE TABLE** operation and an **INSERT … SELECT** operation into a single step. The column names and data types for the

newly created table are determined based on the names and types of the columns queried in the **SELECT** statement.

However, the other attributes of the newly created table, like the delimiter and storage format, are ***not*** based on the format of the table you're querying; you must specify these attributes, or else the newly created table will use the defaults. So in the example here, since there is no **ROW FORMAT** clause, Hive or Impala will use the default field delimiter, which is the ASCII Control+A character. If you wanted comma-delimited files, you would need to include a **ROW FORMAT** clause. You must put this (and any other clauses that specify properties of the new destination table) ***before*** the **AS** keyword, as shown in the example below.

```
CREATE TABLE chicago_employees
   ROW FORMAT DELIMITED FIELDS TERMINATED BY ','
AS
   SELECT * FROM employees WHERE office_id='b';
```

With both CTAS and **INSERT … SELECT**, you could select only some columns to include in the new table by listing only the desired columns in the **SELECT** list. For example, since all the records are in the same office, the **office_id** column isn't really needed. You could use:

```
CREATE TABLE chicago_employees AS
   SELECT empl_id, first_name, last_name, salary
      FROM employees
      WHERE office_id='b';
```

Note that if the data in the **employees** table is updated ***after*** the **chicago_employees** table is created, then the data in the **chicago_employees** table would be stale—any new employees in Chicago are ***not*** automatically added to the **chicago_employees** table. So if you are using a CTAS or **INSERT … SELECT** statement to create a derivative table that you want

kept up to date with the original table, then you need to set up some process to keep it up to date. For example, you could schedule a job that runs an **INSERT OVERWRITE … SELECT** statement to repopulate the derivative table nightly, so the data there is never more than 24 hours stale.

If you complete the optional honors lessons in Week 6 of this course, you will learn another way to take a query and turn it into something that can itself be queried like a table: using a ***view***. The main difference is that a view in Hive and Impala does not materialize (store) the results of the query like an **INSERT … SELECT** statement does. See Week 6 for more about this.

1. Use the information above to create the **chicago_employees** table in two steps—create the table using **LIKE employees** (so the **office_id** column will be included) and populate it with the **INSERT … SELECT** statement above. Verify that the new table has only two rows, for Virginia and Luzja.
2. Drop the **chicago_employees** table (deleting the data—delete it manually if you created the table as an **EXTERNAL** table), and then recreate it ***without*** the **office_id** column, using a CTAS statement. Again verify that the new table has the two rows.