MODULE 1 – INTRODUCTION TO HIVE

History of Hive

- Initially developed at Facebook in 2007 to handle massive amounts of growth.
  - Dataset growth from 15TB to 700TB over a few years
  - RDBMS Data Warehouse was taking too long to process daily jobs
  - Moved their data into scalable open-source Hadoop environment
  - Using Hadoop / creating MapReduce programs was not easy for many users
  - Vision: Bring familiar database concepts to the unstructured world of Hadoop, while still maintaining Hadoop's extensibility and flexibility
  - Hive was open sourced in August 2008
  - Currently used at Facebook for reporting dashboards and ad-hoc analysis

What is Hive?

- Data Warehouse system built on top of Hadoop
  - Takes advantage of Hadoop distributed processing power
- Facilitates easy data summarization, ad-hoc queries, analysis of large datasets stored in Hadoop
- Hive provides a SQL interface (HQL) for data stored in Hadoop
  - Familiar, Widely known syntax
  - Data Definition Language and Data Manipulation Language
- HQL queries implicitly translated to one or more Hadoop MapReduce job(s) for execution
  - Saves you from having to write the MapReduce programs!
  - Clear separation of defining the *what* (you want) vs. the *how* (to get it)
- Hive provides mechanism to project structure onto Hadoop datasets
  - Catalog ("metastore") maps file structure to a tabular form

What Hive is not…

- **Hive is not a full database - but it fits alongside your RDBMS.**
- Is not a real-time processing system
  - Best for heavy analytics and large aggregations – Think Data Warehousing.
  - Latencies are often much higher than RDBMS
  - Schema on Read
    - Fast loads and flexibility – at the cost of query time
    - Use RDBMS for fast run queries.
- Not SQL-92 compliant
  - Does not provide row level inserts, updates or deletes
  - Doesn't support transactions
  - Limited subquery support
- Query optimization still a work in progress
- See HBase for rapid queries and row-level updates and transactions

Hive versus Java and Pig

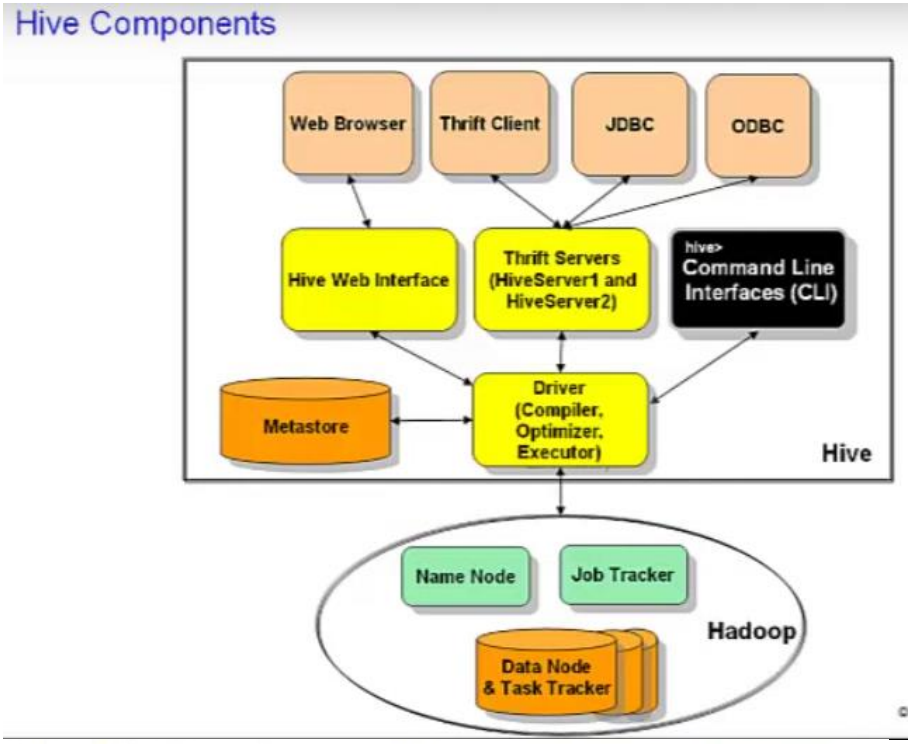
Java

- Word Count MapReduce example
  - Lists words and number of occurrences in a document
  - Takes 63 lines of Java code to write this.
  - Hive solution only takes 7 easy lines of code!

Pig

- High level programming language ("data flow language")
  - Higher learning curve for SQL programmers
- Good for ETL, not as good for ad-hoc querying
- Powerful transformation capabilities
- Often used in combination with Hive

Hive Components



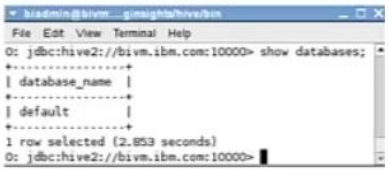
Hive Directory Structure

- Lib directory
  - $HIVE_HOME/lib
  - Location of Hive JAR files
  - Contain the actual Java code that implement the Hive functionality
- Bin directory
  - $HIVE_HOME/bin
  - Location of Hive Scripts/Services
- Conf directory
  - HIVE_HOME/conf
  - Location of configuration files

CLI (Command Line Interface)

- Most common way to interact with Hive
- From the shell you can
  - Perform queries, DML, and DDL
  - View and manipulate table metadata
  - Retrieve query explain plans (execution strategy)
- The Hive Beeline shell and original CLI are located in **$HIVE_HOME**/bin/hive

Beeline CLI



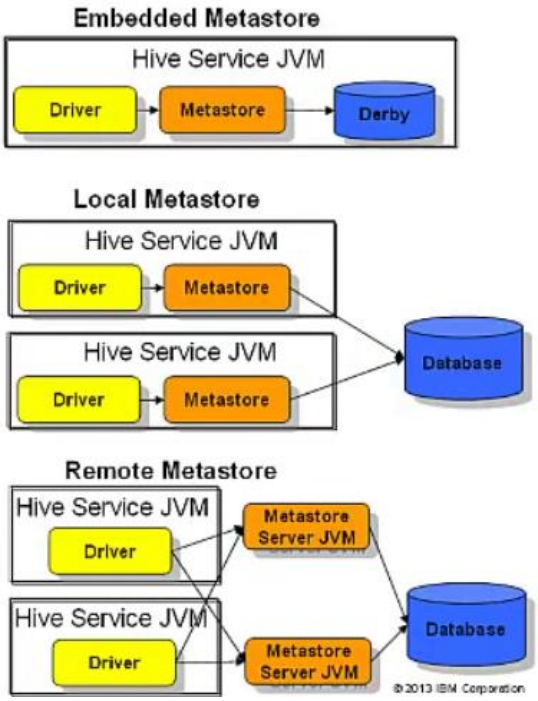
Original Hive CLI

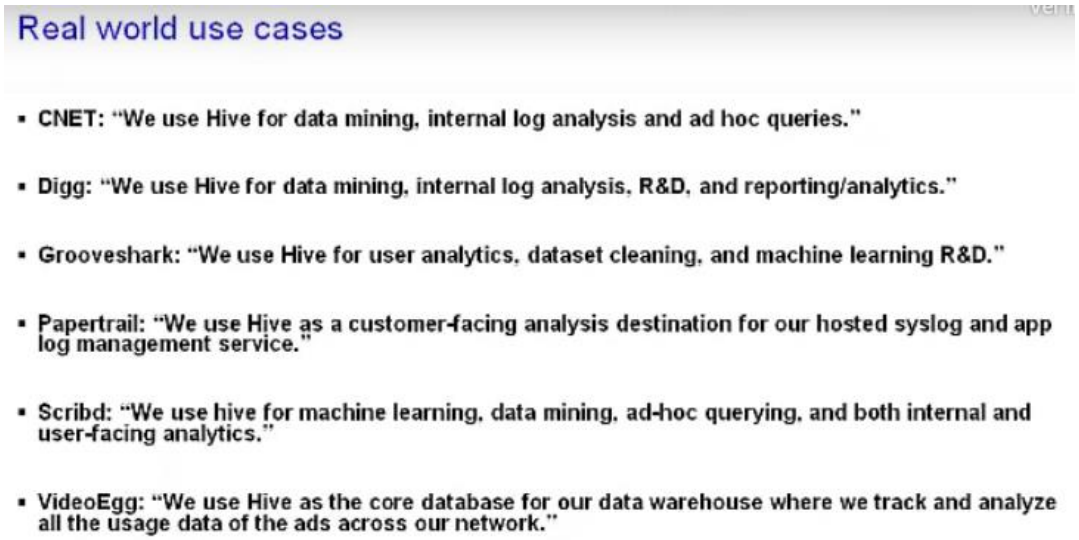
```
$ $HIVE_HOME/bin/hive
2013-01-14 23:36:52.153 GMT : Connection obtained for host: master-
Logging initialized using configuration in file:/opt/ibm/biginsight
Hive history file=/var/ibm/biginsights/hive/query/biadmin/hive_job

hive> show tables;
mytab1
mytab2
mytab3
OK
Time taken: 2.987 seconds
hive> quit;
```

Information Management

Metastore

- 2 pieces – Service & Datastore
- Stores Hive Metadata in 1 of 3 configs:
  - **Embedded:** in-process metastore, in-process database
  - **Local:** in-process metastore, out-of-process database
  - **Remote:** out-of-process metastore, out-of-process database
  - If metastore not configured - Derby database is used
    - Derby metastore allows only one user at a time
  - Can be configured to use a wide variety of storage options (DB2, MySQL, Oracle, XML files, etc.) for more robust metastore





Skip to a navigable version of this video's transcript.
9:29 / 9:43
Maximum Volume.
Skip to end of transcript.

Hi. Welcome to Accessing Hadoop Data Using Hive. This lesson will provide you with an Introduction to Hive.
After completing this lesson, you should be able to:
Describe what Hive is, what it's used for and how it compares to other similar technologies. Describe the Hive architecture.
Describe the main components of Hive. And list interesting ways others are using Hive.
Hive was initially developed by Facebook in 2007 to help the company handle massive amounts of new data. At the time Hive was created, Facebook had a 15TB dataset they needed to work with. A few short years later, that data had grown to 700TB.
Their RDBMS data warehouse was taking too long to process daily jobs so the company decided to move their data into the scalable open-source Hadoop.
The company found that creating MapReduce programs was not easy and was time consuming for many users. When they created Hive, their vision was to bring familiar database concepts to Hadoop, making it easier for all users to work with.
In 2008 Hive was open sourced. Facebook has since used Hive for reporting dashboards and ad-hoc analysis.
So what exactly is Hive? Hive is a data warehouse system built on top of Hadoop. Hive facilitates easy data summarization, ad-hoc queries, and the analysis of very large datasets that are stored in Hadoop. Hive provides a SQL interface, better known as HiveQL or HQL for short, which allows for easy querying of data in Hadoop. HQL has its own Data Definition and Data Manipulation languages which are very similar to the DML and DDL many of us already have experience with.
In Hive, the HQL queries are implicitly translated into one or more MapReduce jobs, shielding the user from much more advanced and time consuming programming.
Hive provides a mechanism to project structure (like tables and partitions) onto the data in Hadoop and uses a metastore to map file structure to tabular form.
Hive is not a full database. However Hive can fit right alongside your RDBMS. There are a variety of things that Hive lacks when compared to an RDBMS. Hive is not a real-time processing system

and is best suited for batch jobs and huge datasets. Think heavy analytics and large aggregations. Latencies are often much higher than in a traditional database system. Hive is schema on read which provides for fast loads and flexibility, at the sacrifice of query time. Hive lacks full SQL support and does not provide row level inserts, updates or deletes. Hive does not support transactions and has limited subquery support. Query optimization is still a work in progress too.

If you are interested in a distributed and scalable data store that supports row-level updates, rapid queries and row-level transaction, then HBase is also worth investigating.

Let's compare Hive to a couple of common alternatives. An example often used is that of the Word Count program. The Word Count program is meant to read in documents on Hadoop and return a listing of all the words read in along with the number of occurrences of those words. Writing a custom MapReduce program to do this takes 63 lines of code. Having Hive perform the same task only takes 7 easy lines of code!

Another Hive alternative is Apache Pig. Pig is a high level programming language, best described as a "data flow language" and not a query language. Being a custom language means there is a higher learning curve for SQL programmers to become comfortable with the Pig language. Pig has powerful data transformation capabilities and is great for ETL. It is not so good for ad-hoc querying. Pig is a nice complement for Hive and the two are often used in tandem in a Hadoop environment.

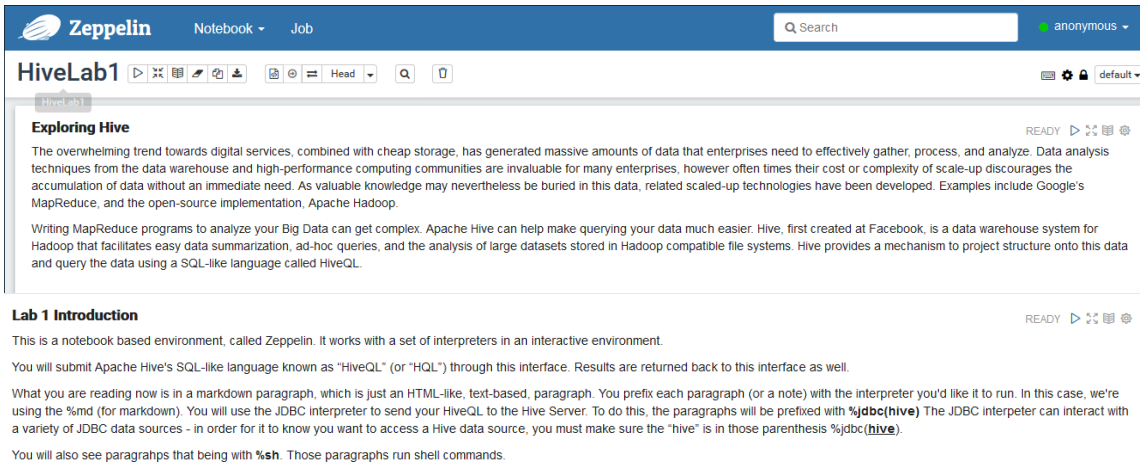
Now let's take a look at Hive's architecture. There are a variety of different ways that you can interface with Hive. You can use a web browser to access Hive via the Hive Web Interface. You could also access Hive using an application over JDBC, ODBC, or the Thrift API, each made possible by Hive's Thrift Server referred to as HiveServer. HiveServer2 was released in Hive 0.11 and serves as a replacement for HiveServer1, though you still have the choice of which HiveServer to run, or can even run them concurrently. HiveServer2 brings many enhancements including the ability to handle concurrent clients and more. Hive also comes with some powerful Command Line interfaces (often referred to as the "CLI"). The introduction of HiveServer2 brings with it a new Hive CLI called Beeline, which can be run in embedded mode or thin client mode. In thin client mode, the Beeline CLI connects to Hive via JDBC and HiveServer2. The original CLI is also included with Hive and can run in embedded mode or as a client to the HiveServer1. Hive comes with a catalog known as the Metastore. The Metastore stores the system catalog and metadata about tables, columns, partitions and so on. The metastore makes mapping file structure to a tabular form possible in Hive. A newer component of Hive is called HCatalog.

HCatalog is built on top of the Hive metastore and incorporates Hive's DDL. HCatalog provides read and write interfaces for Pig and MapReduce and uses Hive's command line interface for issuing data definition and metadata exploration commands. Essentially, HCatalog makes it easier for users of Pig, MapReduce, and Hive, to read and write data on the grid. The Hive Driver, Compiler, Optimizer, and Executor work together to turn a query into a set of Hadoop jobs. The Driver piece manages the lifecycle of a HiveQL statement as it moves through Hive. It maintains a session handle and any session statistics. The Query Compiler compiles HiveQL queries into a DAG of MapReduce tasks. The Execution Engine executes the tasks produced by the compiler in proper dependency order. The Execution Engine interacts with the underlying Hadoop instance, working with the Name Node, Job Tracker and so on.

A typical Hive installation has the following directory structure. First there is a "lib" folder in the Hive installation. The lib folder contains a variety of JAR files. These JAR files contain the Java code that collectively make up the functionality of Hive.

Then there is the "bin" directory. This is the location of a variety of Hive scripts that launch various Hive services.
Finally there is the "conf" directory. This directory contains Hive's configuration files.
Now let's take a slightly deeper look at the Hive CLI. The CLI or "Command Line Interface" is the most common way to interact with the Hive system. From the CLI shell you can perform queries, DML, and DDL. You can view and manipulate table metadata, retrieve query explain plans, and more. Hive currently comes with two command line interfaces : the original CLI and the newer Beeline CLI. These two CLI's are located in Hive's bin directory. There are differences in the original CLI and Beeline architecture, however running commands within the two is a very similar procedure.
The metastore stores the Hive metadata. It consists of two pieces : the service and the datastore. There's three configurations you can choose for your metastore. The first is embedded, which runs the metastore code in the same process with your Hive program and the database that backs the metastore is in the same process as well. The embedded metstore is likely to be used only in a test environment.
The second configuration option is to run the metastore as local, which keeps the metastore code running in process, but moves the database into a separate process that the metastore code communicates with.
The last option is to setup a remote metastore. This option moves the metastore itself out of the process as well. The remote metastore can be useful if you wish to share the metastore with other users. The remote metastore is the configuration you are most likely to use in a production environment, as it provides some additional security benefits on top of what's possible with a local metastore.
A minimum Hive configuration identifies where the metastore is located. If there are no configuration details provided by the user then an embedded Derby database is used. A Derby metastore only allows one user at a time, so it may be advantageous to setup Hive to use a more robust database option, such as DB2, MySQL or another JDBC-compliant database.
A good introduction to Hive has to include some cool real world examples of how companies are using Hive. Here's a very small sampling of some Hive real world use cases. As you can see, Hive is used for data mining,
data analysis, analytics, customer facing business intelligence and a multitude of other uses.
You have now completed this topic. Thank you for watching

>> Lab:



Tasks READY

You are to go through this notebook and follow the instructions in either the markdown paragraphs, or the actual Hive paragraphs. You will run the paragraphs by clicking the "Play" button to the right of each paragraph. Running a markdown paragraph will just print the text out onto the results pane.

```
%md
Play
```

FINISHED

Play

Took 2 sec. Last updated by anonymous at April 28 2020, 4:36:27 PM.

Exploring the Hive environment READY

In this cloud environment, Hive is running in a remote container. Let's navigate to the Hive home directory on the Linux file system within this Hive container and investigate the directories that Hive is comprised of.

```
%sh
ssh root@hive "ls -l /opt/hive"
```

ERROR

```
bash: warning: setlocale: LC_ALL: cannot change locale (en_US.UTF-8)
ls: cannot access /opt/hive: No such file or directory

ExitValue: 2
```

In the above file and directory listing, you will notice the following directories of interest:

- bin – executables to start/stop/configure/check status of hive, various scripts
- conf – Hive environment, metastore, security, and log configuration files
- examples – Hive examples
- hcatalog – Hcatalog files
- jdbc – Contains the hive-jdbc-2.1.0-standalone.jar file
- lib – server's JAR files
- scripts – scripts for upgrading derby and MySQL metastores from one version of Hive to the next

In the next paragraph you will list out the contents of Hive's /bin directory. Notice the entry named **beeline**.
If you were working on the command line and wanted to interact with Hive, Beeline would be a tool you would likely use. However, you are in the cloud, so you wont need to play with that!

FINISHED

Took 0 sec. Last updated by anonymous at April 28 2020, 4:38:09 PM.

```
%sh
ssh root@hive "ls -l /opt/hive/bin"
```

ERROR

```
bash: warning: setlocale: LC_ALL: cannot change locale (en_US.UTF-8)
ls: cannot access /opt/hive/bin: No such file or directory

ExitValue: 2
```

Took 0 sec. Last updated by anonymous at April 28 2020, 4:38:13 PM.

Now let's take a quick look at the Hadoop filesystem (HDFS) that is connected to our Hive container, to see what directories are currently on there.

FINISHED

Took 0 sec. Last updated by anonymous at April 28 2020, 4:38:17 PM.

```
%sh
ssh root@hive 'su - root bash -c "hdfs dfs -ls /"'
```

FINISHED

```
bash: warning: setlocale: LC_ALL: cannot change locale (en_US.UTF-8)
stdin: is not a tty
20/04/28 14:38:36 WARN util.NativeCodeLoader: Unable to load native-hadoop library for your platform... using builtin-java classes where applicable
Found 1 items
drwx-wx-wx   - root supergroup          0 2020-04-28 14:30 /tmp
```

Took 18 sec. Last updated by anonymous at April 28 2020, 4:38:37 PM.

Excellent - HDFS looks quite clean. It wont be quite as clean, when we start adding Hive tables!

FINISHED

On HDFS there is a /user/hive/warehouse directory. If you'd like to confirm this, modify the above hdfs list command to the following:
ssh root@hive 'su - root bash -c "hdfs dfs -ls /user/hive"'

Next, let's run some actual HiveQL, just to prove it works. In the next paragraph, you will tell the JDBC interpeter to send the **SHOW DATABASES** command to your Hive instance.

Took 0 sec. Last updated by anonymous at April 28 2020, 4:38:23 PM.

```
%jdbc(hive)
SHOW DATABASES
```

ERROR

```
java.lang.ClassNotFoundException: org.apache.hive.jdbc.HiveDriver
        at java.net.URLClassLoader.findClass(URLClassLoader.java:382)
        at java.lang.ClassLoader.loadClass(ClassLoader.java:424)
        at sun.misc.Launcher$AppClassLoader.loadClass(Launcher.java:349)
        at java.lang.ClassLoader.loadClass(ClassLoader.java:357)
        at java.lang.Class.forName0(Native Method)
        at java.lang.Class.forName(Class.java:264)
        at org.apache.zeppelin.jdbc.JDBCInterpreter.createConnectionPool(JDBCInterpreter.java:411)
        at org.apache.zeppelin.jdbc.JDBCInterpreter.getConnectionFromPool(JDBCInterpreter.java:422)
        at org.apache.zeppelin.jdbc.JDBCInterpreter.getConnection(JDBCInterpreter.java:442)
        at org.apache.zeppelin.jdbc.JDBCInterpreter.executeSql(JDBCInterpreter.java:691)
```

Notice that the only database returned in the above command, is the **default** database. FINISHED

The default database is the database that automatically comes with Hive when it is first installed. You will of course be creating new databases in the upcoming labs!

Took 0 sec. Last updated by anonymous at April 28 2020, 4:40:26 PM.

Finally, you can take a look at all of the Hive configs that are currently set, but running the **set** command. Run the command below and take a little time to explo FINISHED that are returned.

Took 0 sec. Last updated by anonymous at April 28 2020, 4:40:33 PM.

```
%jdbc(hive)
set
```

ERROR

```
java.lang.ClassNotFoundException: org.apache.hive.jdbc.HiveDriver
        at java.net.URLClassLoader.findClass(URLClassLoader.java:382)
        at java.lang.ClassLoader.loadClass(ClassLoader.java:424)
        at sun.misc.Launcher$AppClassLoader.loadClass(Launcher.java:349)
        at java.lang.ClassLoader.loadClass(ClassLoader.java:357)
        at java.lang.Class.forName0(Native Method)
        at java.lang.Class.forName(Class.java:264)
```

MODULE 2 – HIVE DDL

Hive Data Units

- Organization of Hive data (order of granularity)

```
Database
  ->Table
    ->Partition
      ->Buckets
```

– **Databases**: Namespaces that separate tables and other data units from naming confliction.

– **Tables**: Homogeneous units of data which have the same schema.

– **Partitions**: A virtual column which defines how data is stored on the file system based on its values. Each table can have one or more partitions (and one or more levels of partition).

– **Buckets** (or **Clusters**): In each partition, data can be divided into buckets based on the hash value of a column in the table (useful for sampling, join optimization).

– Note that it is not necessary for tables to be partitioned or bucketed, but these abstractions allow the system to prune large quantities of data during query processing, resulting in faster query execution.

Physical Layout – Data in Hive

Data files are just regular HDFS files
- Variety of storage and record formats can be used
- Internal format can vary table-to-table (delimited, sequence, etc.)

- Warehouse directory in HDFS
  - Specified by "*hive.metastore.warehouse.dir*" in hive-site.xml (can be overridden)
  - E.g. /user/hive/warehouse
- One can think tables, partitions and buckets as directories, subdirectories and files respectively

Hive Entity	Sample	Sample location *In this example $WH is a variable that holds warehouse path*
database	testdb	$WH/testdb.db
table	T	$WH/testdb.db/T
partition	date='01012013'	$WH/testdb.db/T/date=01012013

We could also Bucket...

Bucketing column	userid	$WH/testdb.db/T/date=01012013/000000_0 ... ... $WH/testdb.db/T/date=01012013/000032_0

Database Show/Describe

- **List the Database(s) in the Hive system**

```
hive> SHOW DATABASES;
```

- **Show some basic information about a Database**

```
hive> DESCRIBE DATABASE mydatabase;
```

- **Show more detailed information about a Database**

```
hive> DESCRIBE DATABASE EXTENDED mydatabase;
```

Database Use/Drop/Alter

- **Hive "default" database is used if database is not specified**

- **Tell Hive that your following statements will use a specific database**

```
hive> USE mydatabase;
```

- **Delete a database**

```
hive> DROP DATABASE IF EXISTS mydatabase;
```

  – Note: Database directory is also deleted

- **Alter a database**

```
hive> ALTER DATABASE mydatabase SET DBPROPERTIES ('createdby' =
'bigdatauser2');
```

  – Note: Only possible to update the DBPROPRTIES metadata

Primitive Data Types

- Types are associated with the columns in the tables.
- Primitive types in Hive (subset of typical RDBMS types):
  - **Integers**
    - TINYINT - 1 byte integer
    - SMALLINT - 2 byte integer
    - INT - 4 byte integer
    - BIGINT - 8 byte integer
  - **Boolean type**
    - BOOLEAN - TRUE/FALSE
  - **Floating point numbers**
    - FLOAT - single precision
    - DOUBLE - Double precision
    - DECIMAL - provide precise values and greater range than Double
  - **String types**
    - STRING - sequence of characters in a specified character set
    - VARCHAR – specify length of character string (between 1 and 65,355)
  - **Date/Time types**
    - TIMESTAMP - YYYY-MM-DD HH:MM:SS.fffffffff
    - DATE - YYYY-MM-DD
  - **Binary type**
    - Binary - array of bytes (similar to VARBINARY in RDBMS)

  – On comparison, Hive does some implicit casting
    - Any integer to larger of two integer types
    - FLOAT to DOUBLE
    - Any integer to DOUBLE

Complex Data Types

- Complex Types can be built up from primitive types and other composite types.
- **Arrays** - Indexable lists containing elements of the same type.
  - Format: ARRAY<data_type>
  - Literal syntax example: array('user1', 'user2')
  - Accessing Elements: [n] notation where n is an index into the array. E.g. *arrayname[0]*
- **Structs** - Collection of elements of different types.
  - Format: STRUCT<col_name : data_type, ...>
  - Literal syntax example: struct('Jake', '213')
  - Accessing Elements: DOT (.) notation. E.g. *structname.firstname*
- **Maps** – Collection of key-value tuples.
  - Format: MAP<primitive_type, data_type>
  - Literal syntax example: map('business_title', 'CEO', 'rank', '1')
  - Accessing Elements: ['element name'] notation. E.g. *mapname['business_title']*
- **Union** – at any one point can hold exactly one of their specified data types.
  - Format: UNIONTYPE<data_type, data_type, ...>
  - Accessing Elements: Use the create union UDF (see Hive docs for more info)

Creating Databases

- **Create a database named "mydatabase"**

```
hive> CREATE DATABASE mydatabase;
```

- **Create a database named "mydatabase" and override the Hive warehouse configured location**

```
hive> CREATE DATABASE mydatabase
    > LOCATION '/myfavorite/folder/';
```

- **Create a database and add a descriptive comment**

```
hive> CREATE DATABASE mydatabase
    > COMMENT 'This is my database';
```

- **Create a database and some descriptive properties**

```
hive> CREATE DATABASE mydatabase
    > WITH DBPROPERTIES ('createdby' = 'bigdatauser', 'date' =
'2014-01-01');
```

Hi. Welcome to Accessing Hadoop Data Using Hive. In this lesson we will discuss Hive DDL.

After completing this lesson, you should be able to:

Create databases and tables in Hive, while using a variety of different Data Types.

Run a variety of different DDL commands. Use Partitioning to improve performance of Hive queries. And create Managed and External tables in

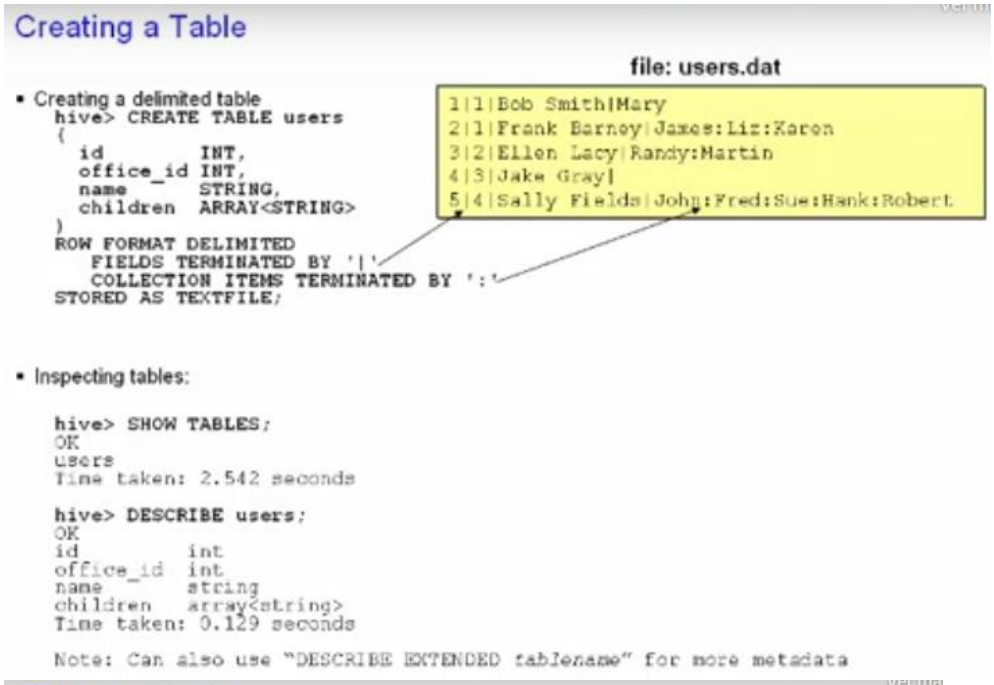
Hive.

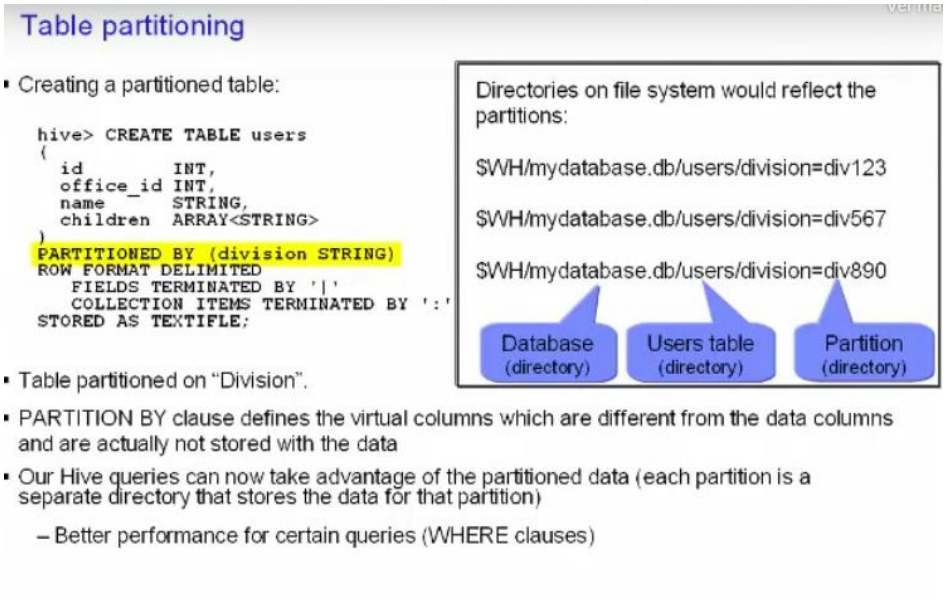
In Hive data is organized into the following units.

First data is organized into Databases which are namespaces that separate tables and other data units from naming conflicts. Next data is organized into tables which are homogenous units of data that have the same schema.

Data can then be organized into partitions, though this is not a requirement. A Partition in Hive is a virtual column that defines how
data is stored on the file system based on its
values. A table can have zero or more partitions. Finally, in each partition, data can be organized
into Buckets based on the hash value of a column in the table.
Again, note that it isn't necessary for tables to be partitioned or bucketed in Hive, however these abstractions allow the system to prune large quantities of data during query processing, resulting in faster query execution and reduced latencies.
Now that you've seen that Hive logically breaks data out into Databases, Tables, and Partitions, let's take a look at how this data is physically stored.
Data files in Hive are stored right in the Hadoop file system. A variety of different storage and record formats can be used for your data. The internal format of your data can really vary from table-to-table depending on how you want to set things up.
In Hive your data is stored in the warehouse directory as specified in your configuration file unless you override this location. Listed on this slide is an example of how Hive stores databases, tables, partitions, and
buckets. You can see that a directory is created for a database and inside that is a subdirectory for a table, that itself contains a subdirectory for a partition. The actual file that holds the data would be stored inside
that partition folder in this example. If we chose to bucket the table in this example, then our data would be divided up into multiple bucket files and stored inside of the partition directory. In this example our data
is stored in 32 bucketed data files inside of the partition directory. When creating this table, we would specify the number of buckets
- it is not a randomly assigned number.
Creating a database in Hive is easy. It can be as simple as typing three words - "create database" followed by the name you'd like the database to be called. There are additional options you can add to your basic Create statement, such as specifying the location you'd like the database to be created - and adding a comment to the metadata for that database.
To view a listing of the databases in Hive, simply run the statement "Show Databases" in the Hive shell. You can show information about
an individual database by using the "Describe" statement. By adding the word "Extended" to the Describe statement, you can get additional information about the database.
A typical Hive installation comes installed with a "default" database. This default database is used if you run statements without first specifying which database you'd like to work with. To tell Hive that you'd like to work with
a certain database, simply enter the USE statement in the Hive shell.
To delete a database in Hive, you use the Drop Database syntax. It is also possible to alter the DBPROPERTIES of a database by using the
Alter command.
Before we look at Tables in Hive, it is prudent that we review the different Data Types available for use. Data types are associated with the columns in a table.
Hive has a variety of primitive data types available - though not as many as the typical RDBMS has. There are multiple Integer types, including TINYINT, SMALLINT, INT, and BIGINT. There is a BOOLEAN type which can be True or False. There are three floating point number types, specifically FLOAT, DOUBLE, and DECIMAL. Hive has two STRING types - String and VARCHAR. There are two Date/Time formats - TIMESTAMP and DATE. Hive also has a Binary type.
When comparing different data types in a Hive query, it's important to note that casting will take place. If two different types of integers are compared, then the smaller integer type will be cast to the larger type and so on.

Complex data types can be built up from the primitive types we just reviewed. If you've worked with different programming languages then these may look familiar.
The ARRAY data type is an indexable list containing elements of the same type. Note that arrays in Hive are zero-based. The STRUCT data type in Hive represents a collection of elements of different types. Dot
notation is used to access the elements in a Struct.
The MAP data type is a collection of key-value tuples. The keys must be primitive typed, but the values can be any type. Hive also has a Union data type which can
at any one point hold exactly one of their specified data types.





Managed Vs External Tables

Managed Tables

- By default Hive tables are "Managed".
  - Hive controls the metadata AND the lifecycle of the data
  - Data stored in subdirectories within the hive.metastore.warehouse.dir location
  - Dropping a managed table deletes the data in the table

External Tables

- Store table in a directory outside of Hive
- Useful if sharing your data with other tools
- Hive does not assume it owns the data in the table
  - Dropping table deletes the tables metadata - NOT the actual data
- Must add the EXTERNAL and LOCATION keywords to CREATE statement

```
CREATE EXTERNAL TABLE users
...
LOCATION '/path/to/your/data';
```

Drop/Alter Table

- **Delete a table**

```
hive> DROP TABLE IF EXISTS users;
```

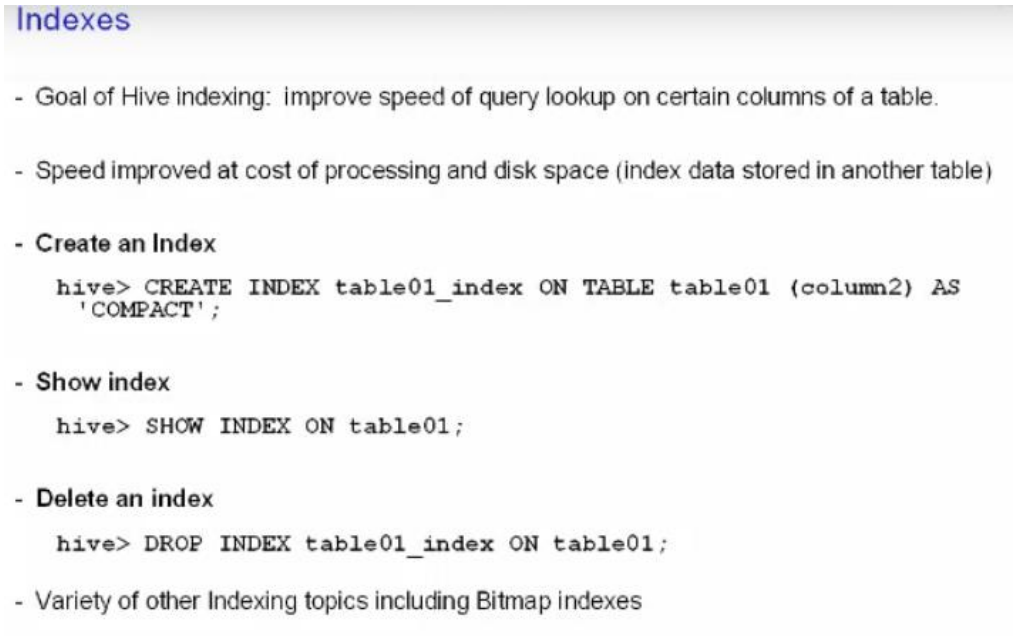
- **Change name of table**

```
hive> ALTER TABLE users RENAME TO employees;
```

- **Add two columns to end of table**

```
hive> ALTER TABLE users ADD COLUMNS (
    location  STRING,
    age       INT);
```

- **Variety of other Alter statements**
  - Delete columns
  - Alter table properties
  - Alter storage properties
  - Alter partitions



Now that we know what data types can be used in Hive, let's look at an example of a simple Hive table being created with HQL. In this example a table named "users" is being created with 4 columns. There is an ID column which is of the INT data type, along with an office_id column of the same type. There also is a name column that is of the String data type, and finally an array column called "children" which can hold String values. We tell Hive that the fields in our rows are delimited by the pipe character. We then tell Hive that the collection items in the array field are terminated by the colon character. Finally, we explicitly specify that our data file is a plain text file. Once our table is created, we can then run the "show tables" statement to see a listing of all the tables in our database. We can also use the "DESCRIBE" statement to view some metadata for a specific table. Further, we can use "DESCRIBE EXTENDED" to view additional table metadata.

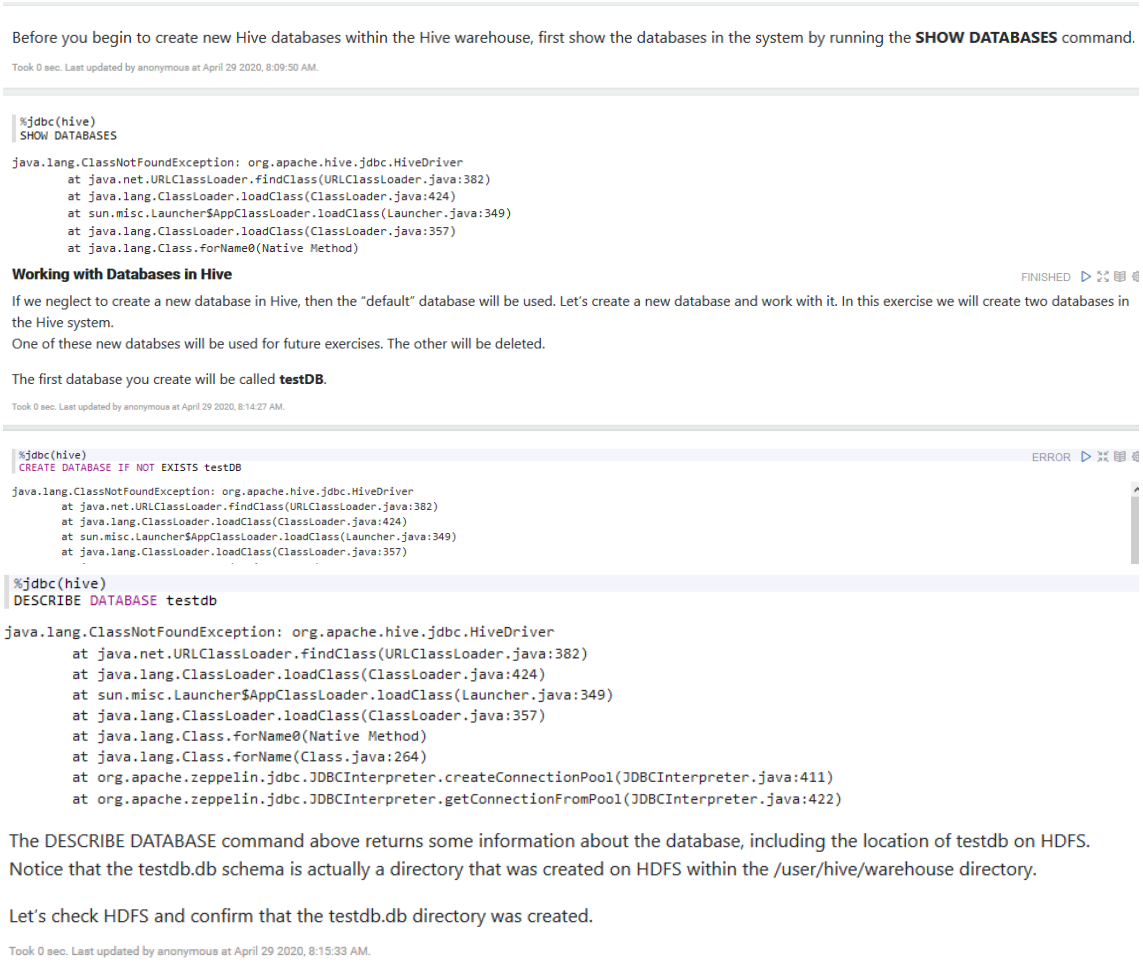
In this example, our new users table will now be partitioned on "division", which we specify is of the String data type. We accomplished this by adding the PARTITION BY clause. Division will now be a virtual column - the division is not actually stored in the data file with the rest of the data. Now that we have a partitioned table, once data is loaded in, our Hive queries will be able to take advantage of the fact that each partition is a separate directory that contains just that one partitions' data on the file system. This will help us achieve better performance especially when running queries with WHERE clauses. The less data we need to read and process for a given query, the better, as we gain the benefit of reduced latency.

Inside the box on this screen you can get a feel for what the physical directory structure would look like for a table with 3 partitions. Down the Warehouse path we can see the directory for the database, the table, and the partition.

Tables in Hive can either be Managed or External. Tables by default are Managed by Hive - Hive controls the metadata for that table AND the lifecycle of the actual data in the table. The managed table data is stored in subdirectories within the configured warehouse directory. Dropping a managed table will delete the actual table data in addition to the metadata Hive stored for that table. Let's contrast this behavior to Hive's External Tables. An external table's data file(s) are stored in a location outside of Hive. Hive

does not assume it owns the data in the table -
dropping an external table deletes just the table metadata and leaves the actual data untouched. External tables can be useful if you are sharing your data with other tools outside of Hive. Creating an external table simply requires the addition of the EXTERNAL and LOCATION keywords in a create table statement.
Dropping a table in Hive is done using the Drop Table syntax. There are a variety of ways to Alter a Hive table. We can change the name of a table, add or remove columns, alter table properties and more.
In Hive there's a variety of advanced topics, one of them being Indexing. The goal of Hive indexing is to improve the speed of query lookups for certain columns of a table. It's important to note that the speed improvement comes at the cost of processing and disk space as the index data is stored in another table on the file system.
Here's a few simple statements that demonstrate how to create, show and delete an index. There's also a variety of other indexing topics including bitmap indexes and more.
You have now completed this topic. Thank you for watching.

>>Lab:



```
%sh
ssh  root@hive 'su - root bash -c "hdfs dfs -ls /user/hive/warehouse/"'
```

```
bash: warning: setlocale: LC_ALL: cannot change locale (en_US.UTF-8)
stdin: is not a tty
20/04/29 06:15:55 WARN util.NativeCodeLoader: Unable to load native-hadoop library for your platform... using builtin-java classes where applicable
ls: `/user/hive/warehouse/': No such file or directory

ExitValue: 1
```

Took 19 sec. Last updated by anonymous at April 29 2020, 8:15:56 AM.

Add some information to the DBPROPERTIES metadata for the testdb database. You can do this by using the **ALTER DATABASE** syntax.

Took 0 sec. Last updated by anonymous at April 29 2020, 8:15:43 AM.

```
%jdbc(hive)
ALTER DATABASE testdb SET DBPROPERTIES ('creator' = 'bigdatarockstar')
```

```
java.lang.ClassNotFoundException: org.apache.hive.jdbc.HiveDriver
        at java.net.URLClassLoader.findClass(URLClassLoader.java:382)
        at java.lang.ClassLoader.loadClass(ClassLoader.java:424)
```

Let's view the extended details of our testdb database.

Took 0 sec. Last updated by anonymous at April 29 2020, 8:16:25 AM.

```
%jdbc(hive)
DESCRIBE DATABASE EXTENDED testdb
```

```
%jdbc(hive)
DROP DATABASE testdb CASCADE
```

```
%jdbc(hive)
CREATE DATABASE IF NOT EXISTS computersalesdb
```

Exploring the Sample Datasets

READY

Before we begin creating tables in our new database it is important to understand what data is in our sample files and how that data is structured.

First you will create a directory, chmod it to 777, and import the sample data files. Then you will analyze the files.

To import the data files, use the **wget** command to download them. Place the files in a newly created directory in /tmp/hive_data on the hive container's Linux filesystem. There are three csv data files that you will import below.

```
%sh
ssh root@hive 'su - root bash -c "mkdir /tmp/hive_data"'
```

FINISHED

```
bash: warning: setlocale: LC_ALL: cannot change locale (en_US.UTF-8)
stdin: is not a tty
```

Took 15 sec. Last updated by anonymous at April 29 2020, 8:18:31 AM.

Use the **ls** command to check the /tmp directory on the hive container to ensure the /tmp/hive_data directory was created.

FINISHED

Took 0 sec. Last updated by anonymous at April 29 2020, 8:18:18 AM.

```
%sh
ssh root@hive "ls -l /tmp"
```

READY

Use the **chmod** command to set the permission for /tmp/hive_data to 777 (full read/write permission).

READY

```
%sh
ssh root@hive 'su - root bash -c "chmod -R 777 /tmp/hive_data"'
```

FINISHED

```
bash: warning: setlocale: LC_ALL: cannot change locale (en_US.UTF-8)
stdin: is not a tty
```

Took 15 sec. Last updated by anonymous at April 29 2020, 8:21:57 AM.

Use **wget** to download the three csv files to /tmp/hive_data on the hive container.

FINISHED

Took 0 sec. Last updated by anonymous at April 29 2020, 8:21:41 AM.

```
%sh
ssh root@hive 'su - root bash -c "wget --quiet --output-document /tmp/hive_data/Customer.csv https://ibm.box.com/shared/static/fjcm8worqj0dki7n0g36nml5t7v8o8...."'
ssh root@hive 'su - root bash -c "wget --quiet --output-document /tmp/hive_data/Sales.csv https://ibm.box.com/shared/static/xzebk7ulqfep31zuc57jsc2vboemlatb.csv"'
ssh root@hive 'su - root bash -c "wget --quiet --output-document /tmp/hive_data/Product.csv https://ibm.box.com/shared/static/kjcvfgcgnnvcoya7e8i7gdp0yircxodc.csv"'
```

RUNNING 0%

Sample Data Descriptions

Now you will analyze each data file. Before we do that, here is the metadata for each file.

Our sample data is from a fictitious computer retailer. The company sells computer parts and generally serves a single State in the country.

Customer.csv:

```
Purpose: Hold customer records.

Columns:
FNAME                       Customer's First Name
LNAME                       Customer's Last Name
STATUS                      Active or Inactive status
TELNO                       Telephone # Customer's unique ID
CUSTOMER_ID CITY|ZIP        City and Zip code separated by the "|" character.
```

Product.csv:

```
Purpose: Hold product records.
```

To import the data files, use the **wget** command to download them. Place the files in a newly created directory in /tmp/hive_data on the hive container's Linux filesystem. There are three csv data files that you will import below.

```
%sh
ssh root@hive 'su - root bash -c "mkdir /tmp/hive_data"'
```

FINISHED

```
bash: warning: setlocale: LC_ALL: cannot change locale (en_US.UTF-8)
stdin: is not a tty
```

Took 15 sec. Last updated by anonymous at April 29 2020, 8:18:31 AM.

Use the **ls** command to check the /tmp directory on the hive container to ensure the /tmp/hive_data directory was created.

Took 0 sec. Last updated by anonymous at April 29 2020, 8:18:18 AM.

```
%sh
ssh root@hive "ls -l /tmp"
```

Use the **chmod** command to set the permission for /tmp/hive_data to 777 (full read/write permission).

```
%sh
ssh root@hive 'su - root bash -c "chmod -R 777 /tmp/hive_data"'
```

```
bash: warning: setlocale: LC_ALL: cannot change locale (en_US.UTF-8)
stdin: is not a tty
```

Took 15 sec. Last updated by anonymous at April 29 2020, 8:21:57 AM.

Use **wget** to download the three csv files to /tmp/hive_data on the hive container.

Took 0 sec. Last updated by anonymous at April 29 2020, 8:21:41 AM.

```
%sh
ssh root@hive 'su - root bash -c "wget --quiet --output-document /tmp/hive_data/Customer.csv https://ibm.
ssh root@hive 'su - root bash -c "wget --quiet --output-document /tmp/hive_data/Sales.csv https://ibm.box
ssh root@hive 'su - root bash -c "wget --quiet --output-document /tmp/hive_data/Product.csv https://ibm.b
```

Managed Non-Partitioned Tables

FINISHED

The first table we will create in Hive is the products table. This table will be fully managed by Hive and will not contain any partitions.

Note the data types we have assigned to the different columns. The packaged_with column is of special interest – it is designated as an Array of Strings. The array will hold data that is separated by the colon ":" character - e.g. satacable:manual. We also tell Hive that the columns in our rows are delimited by commas ",". The last line tells Hive that our data file is a plain text file.

Took 0 sec. Last updated by anonymous at April 29 2020, 8:23:49 AM.

```
%jdbc(hive)
CREATE TABLE computersalesdb.products
(
        prod_name       STRING,
        description     STRING,
        category        STRING,
        qty_on_hand     INT,
        prod_num        STRING,
        packaged_with   ARRAY<STRING>
)
ROW FORMAT DELIMITED
        FIELDS TERMINATED BY ','
        COLLECTION ITEMS TERMINATED BY ':'
STORED AS TEXTFILE
```

READY

```
%jdbc(hive)
SHOW TABLES IN computersalesdb
```

You can see that only one table exists in our database and it is the new products table we just created.

Now add a note to the TBLPROPERTIES for our new products table.

+ Add Paragraph

```
%jdbc(hive)
ALTER TABLE computersalesdb.products SET TBLPROPERTIES ('details' = 'This table holds products')
```

List the extended details of the products table.

```
%jdbc(hive)
DESCRIBE EXTENDED computersalesdb.products
```

```
%jdbc(hive)
DESCRIBE EXTENDED computersalesdb.products
```

That is a lot of details! Notice there is some interesting info including the location of this table within HDFS: /user/hive/warehouse/computersalesdb.db/products

Let's verify that the products directory was created on HDFS in the location listed above. Run the HDFS ls command from within a Linux console. First list the contents of the database d contents of the products table directory.

+ Add Paragraph

```
%sh
ssh  root@hive 'su - root bash -c "hdfs dfs -ls /user/hive/warehouse/computersalesdb.db"'
```

```
%sh
ssh  root@hive 'su - root bash -c "hdfs dfs -ls /user/hive/warehouse/computersalesdb.db/products"'
```

The first **hdfs ls** command above confirms that there is in fact a products table directory on HDFS. The second command shows that there are no files within the products directory yet This directory will be empty until we load data into the products table in a later exercise.

Imagine that our fictitious computer company adds sales data to a "sales_staging" table at the end of each month. READY
From this sales_staging table they then move the data they want to analyze into a partitioned "sales" table. The partitioned sales table is the one they actually use for their analysis.

Now that you know how to create tables, you will create one more managed non-partitioned table called "sales_staging". This table will hold ALL of the sales data from the sales.csv file. In later exercises you will actually split this sales_staging data into a partitioned table called "sales".

You will now create the new sales_staging table in Hive.

READY

```
%jdbc(hive)
CREATE TABLE computersalesdb.sales_staging
 (
        cust_id     STRING,
        prod_num    STRING,
        qty         INT,
        sale_date   DATE,
        sales_id    STRING
 )
 COMMENT 'Staging table for sales data'
 ROW FORMAT DELIMITED
        FIELDS TERMINATED BY ','
 STORED AS TEXTFILE
 TBLPROPERTIES("skip.header.line.count"="1")
```

You can now assume that the new sales_staging table directory is on HDFS in the following folder: /user/hive/warehouse/computersalesdb.db/sales_staging. Quickly confirm this. READY

You can now assume that the new sales_staging table directory is on HDFS in the following folder: /user/hive/warehouse/computersalesdb.db/sales_staging. Quickly confirm this.

```
%sh
ssh  root@hive 'su - root bash -c "hdfs dfs -ls /user/hive/warehouse/computersalesdb.db"'
```

Sure enough, the sales_staging directory was created and is now being managed by Hive.

Ask Hive to show you the tables in the computersalesdb database. Confirm your new sales_staging table is in the Hive catalog.

```
%jdbc(hive)
SHOW TABLES in computersalesdb
```

Managed Partitioned Tables

Now you will create a partitioned table. This table will be a managed table – Hive will manage the metadata and lifecycle of this table, just like the tables we previously created.

Now, create the sales table. This table will be partitioned on the sales date.

```
%jdbc(hive)
CREATE TABLE computersalesdb.sales
 (
        cust_id    STRING,
        prod_num   STRING,
        qty        INT,
        sales_id   STRING
 )
 COMMENT 'Table for analysis of sales data'
 PARTITIONED BY (sales_date STRING)
 ROW FORMAT DELIMITED
        FIELDS TERMINATED BY ','
 STORED AS TEXTFILE
```

Notice that you listed the sales_date in the PARTITIONED BY clause instead of listing it in the data column metadata.
Since you are partitioning on sales_date, Hive will keep track of the dates for you outside of the actual data.

Let's view the extended details of our new table.

```
%jdbc(hive)
DESCRIBE EXTENDED computersalesdb.sales
```

You can see a directory was created on HDFS at: /user/hive/warehouse/computersalesdb.db/sales.

When you later put data into this table, a new directory will be created inside the sales directory for EACH partition.

The following line in the details shows us how our table is partitioned: partitionKeys: [FieldSchema(name:sales_date, type:string, comment:null)]

External Table

READY

Another department in our fictitious computer company would like to be able to analyze the customer data. It therefore makes sense that we setup the customer table as EXTERNAL so they can use their tools on the data and we can use ours (Hive). We will place a copy of the Customer.csv file in HDFS and then create a new table in Hive that points to this data.

First you need to create a new directory – let's call it "shared_hive_data" - on HDFS that can house our Customer.csv data file. Let's put this in the /tmp directory on HDFS. Run the command to make the new directory on HDFS.

```
%sh
ssh  root@hive 'su - root bash -c "hdfs dfs -mkdir /tmp/shared_hive_data/"'
```

READY

```
%sh
ssh  root@hive 'su - root bash -c "hdfs dfs -ls /tmp"'
```

READY

```
%sh
ssh root@hive 'su - root bash -c "hdfs dfs -chmod 777 /tmp/shared_hive_data"'
```

READY

Run the "cat" command to verify the data is in the Customer.csv file on HDFS.

```
%sh
ssh  root@hive 'su - root bash -c "hdfs dfs -cat /tmp/shared_hive_data/Customer.csv"'
```

Now you simply need to define the external customer table.

```
%jdbc(hive)
CREATE EXTERNAL TABLE IF NOT EXISTS computersalesdb.customer
(
        fname        STRING,
        lname        STRING,
        status       STRING,
        telno        STRING,
        customer_id  STRING,
        city_zip     STRUCT<city:STRING, zip:STRING>
)
COMMENT 'External table for customer data'
ROW FORMAT DELIMITED
        FIELDS TERMINATED BY ','
        COLLECTION ITEMS TERMINATED BY '|'
LOCATION '/tmp/shared_hive_data'
TBLPROPERTIES("skip.header.line.count"="1")
```

Let's view the extended details of the new external customer table.

```
%jdbc(hive)
DESCRIBE EXTENDED computersalesdb.customer
```

You can see above that the location points to the /tmp/shared_hive_data directory that you designated on HDFS. Also notice towards the end of the output that tableType:EXTERNAL_

Ask Hive to show you the tables in the metastore. Then check the Hive warehouse on HDFS to prove to yourself that Hive did NOT create a customer directory within it's default wareh

Hive will manage the metadata for the external table, but it wont manage the actual data.

```
%jdbc(hive)
show tables in computersalesdb
```

```
%sh
ssh  root@hive 'su - root hash -c "hdfs dfs -ls /user/hive/warehouse/computersalesdb.db"'
```

Since **customer** is an external table and Hive already knows where the data is sitting, you can already begin to run queries on this table.

Reward yourself for completing this lab by running a simple select query as proof.

```
%jdbc(hive)
SELECT * FROM computersalesdb.customer
```

This Hive query didn't run any MapReduce jobs. Hive is able to read the data file and return the results without using MapReduce, since this is a simple SELECT and LIMIT statement.

MODULE 3 – HIVE DML

Loading Data into Hive – From a File

- **Loading data from input file (Schema on Read)**

```
hive> LOAD DATA LOCAL INPATH '/tmp/data/users.dat'
      OVERWRITE INTO TABLE users;
```

  - The "LOCAL" indicates the source data is on the local filesystem
  - Local data is **copied** to final location
  - Otherwise file is assumed to be on HDFS and is **moved** to final location
  - Hive does not do any transformation while loading data into tables

- **Loading data into a partition requires PARTITION clause**

```
hive> LOAD DATA LOCAL INPATH '/tmp/data/usersny.dat'
      OVERWRITE INTO TABLE users;
      PARTITION (country = 'US', state = 'NY')
```

  - HDFS directory is created:
    /user/hive/warehouse/mydb.db/users/country=US/state=NY
    - usersny.dat file is copied to this HDFS directory

Loading Data from a Directory

- **Load data from an HDFS directory instead of single file**

```
hive> LOAD DATA INPATH '/user/biadmin/userdatafiles'
      OVERWRITE INTO TABLE users;
```

  - Lack of "LOCAL" keyword means source data is on the HDFS file system
  - Data is **moved** to final location
  - All of the files in the /user/biadmin/userdatafiles directory are copied over into Hive
  - OVERWRITE keyword causes contents of target table to be deleted and replaced.
    - Leaving out the OVERWRITE means files will be added to the table
    - If target has file name collision then new file will overwrite existing Hive file

Exporting Data out of Hive

- If data files are already format you like, can just copy them out of HDFS
- Query results can be inserted into file system directories (local or HDFS)
- if LOCAL keyword is used, Hive will write data to the directory on the local file system.

```
INSERT OVERWRITE LOCAL DIRECTORY '/mydirectory/dataexports'
SELECT sale_id, product, date
FROM sales
WHERE date='2014-01-01';
```

- Data written to the file system is by default serialized as text with columns separated by ^A and rows separated by newlines.
  - Non-primitive columns serialized to JSON format.
  - Delimiters and file format may be specified.

- INSERT OVERWRITE statements to HDFS is good way to extract large amounts of data from Hive. Hive can write to HDFS directories in parallel from within a MapReduce job.

- **Warning: The directory is OVERWRITTEN!**
  - if the specified path exists, it is clobbered and replaced with output.

Hi. Welcome to Accessing Hadoop Data Using Hive. In this lesson we will discuss Hive DML.
After completing this lesson, you should be able to:
Load data into Hive Export data out of Hive
And, Run a variety of different HiveQL DML queries
In order to make use of Hive we need to get some actual data into the database.
The first example shows one way to bring data into a Hive table. We load the data from a theoretical file named users.dat into the table named users in this example. The LOCAL keyword tells Hive to look on the local file system (i.e. our Linux file system) for the file named users.dat in the /tmp/data folder. The
local data is COPIED to its final location on
HDFS. If we leave off the LOCAL keyword, Hive assumes
the location we are referring to is on HDFS. If our data file is on HDFS then the data is actually moved to its final location within Hive's grasp in HDFS. When the data is being loaded in to tables, Hive does not do any transformations to the data.
The second example shows another local file called userny.dat being loaded into a partitioned users table. The table is partitioned on country and state. A new HDFS directory will be created within the users table directory called country=US which will contain another directory called state=NY. In this example the local file is again copied into the HDFS directory.
We can also load data from a whole directory into a Hive table. In this example we are loading data from a directory of files into a users table. If you look carefully you'll notice this statement is missing the LOCAL keyword,
therefore Hive will look to HDFS for the data. The actual data is moved to the final location under Hive's control on HDFS. All of
the files that were in the /user/biadmin/userdatafiles directory are copied over into Hive.
Note the OVERWRITE keyword in this statement. It causes the contents of the target table to be deleted and replaced. If we were to leave the OVERWRITE keyword out, then the data files would be added to the contents of the table. If that is your intention, just make sure the new data files are named differently from the ones already existing in Hive, otherwise Hive will overwrite the old one with the new one.
Data can be loaded into Hive from a query.
The first example shows that tables can be created from the results of a query on another table in Hive. The second example is similar, except it uses the INSERT OVERWRITE clause to get data into the new table.
Getting data into Hive is great, but we also need to be able to get data out of Hive in many scenarios. Let's say you had a table in Hive and you wanted all the data in that table and liked the storage format the data was already in. You could just go ahead and copy the data file right out of Hive, since you know were it is located on HDFS.
Alternatively, and perhaps more likely, you want to write out the results of a query to a file. The example here shows you how to use
the INSERT OVERWRITE LOCAL DIRECTORY clause to do this. In our example
we write out a few columns from the sales table to the /mydirectory/dataexports directory on our local system. The data written to the file system is by default serialized as text and the columns are separated by the CONTROL-A character and rows are separated by newlines by default. Non-primitive columns are serialized to JSON format. You may also explicitly specify delimiters and file format for the exported data.
Using the INSERT OVERWRITE statement to HDFS is a good way to extract large amounts of data from Hive because Hive can write to these directories in parallel from within a MapReduce job. A big warning here - the directory in the
INSERT OVERWRITE statement is Overwritten. The directory gets clobbered

and replaced with output. Don't accidentally delete your home directory like I did when I first tried this statement out!

SELECT FROM

users
id
office_id
name
children

- If you know SQL, then HiveQL's DML has few surprises
- Simple SELECT all query:

```
hive> SELECT * FROM users LIMIT 3;
1       1       Bob Smith       ["Mary"]
2       1       Frank Barney    ["James","Liz","Karen"]
3       2       Ellen Lacy      ["Randy","Martin"]
```

  – Note the last column is ARRAY data type. Hive outputs the array elements in brackets.
  – LIMIT puts an upper limit on number rows returned

- SELECT query (ARRAY indexing):

```
hive> SELECT name, children[0] FROM users;
Bob Smith       Mary
Frank Barney    James
Ellen Lacy      Randy
```

  – First element from children ARRAY is selected.

Customer
id
cust_name
phone
Address
ytd_sales

- SELECT query with STRUCT column (address):

```
hive> SELECT cust_name, address FROM customer;
Bruce Smith     {"city":"Burlington","zipcode":05401}
Chuck Barney    {"city":"Jericho","zipcode":05465}
```

  – The address column (type STRUCT) prints out in JSON format

WHERE

- Predicate Operators
  =, <>, !=, <, <=, >, >=, IS NULL, IS NOT NULL, LIKE, RLIKE

- SELECT WHERE query with STRUCT column (address):

```
hive> SELECT cust_name, address.city FROM customer
      WHERE address.city = 'Burlington';
      Bruce Smith     Burlington
```

  – Dot notation used to access struct data

- SELECT WHERE query with GROUP BY clause:

```
hive> SELECT address.city, sum(ytd_sales) FROM customer
      WHERE address.city = 'Burlington'
      GROUP BY address.city;
```

  – GROUP BY usually used with aggregate functions
  – Can also use the HAVING clause with GROUP BY

Column Alias and Nested Select

- SELECT query with column alias

```
hive> SELECT cust_name, address.city as city
      FROM customer;
```

  – Address.city is given the column alias "city"

- Column alias comes in handy when doing nested SELECT...

```
hive> FROM(
          SELECT cust_name, round(ytd_sales * .01) as rewards
          FROM customer;
      ) c
      SELECT c.name, c.rewards
      WHERE c.rewards > 25;
```

  – The first query is aliased as "c". We then select from the results of that query.

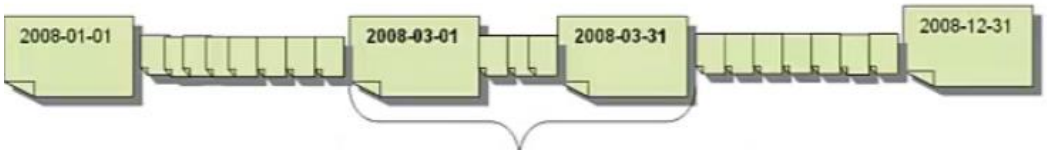
Selecting from Partitions

Taking advantage of partitions

- Partition pruning – Hive scans only a fraction of the table relevant to the partitions specified by the query.
- Hive does partition pruning if the partition predicates are specified in the WHERE clause or the ON clause in a JOIN.
- Example: If table `page_views` is partitioned on `log_date`, the following query retrieves rows for just days between `2008-03-01` and `2008-03-31`.

```
hive> SELECT * FROM page_views
      WHERE log_date >= '2008-03-01' AND log_date <= '2008-03-31';
```

  – Imagine we had a year of page_view data partitioned into 365 partition files. Hive only needs to open and read the 31 partitioned data files in the above example.

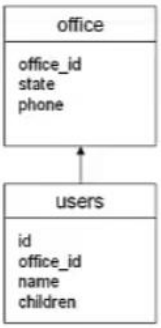


Joins

- Hive supports joins via ANSI join syntax only

```
select ...
  from T1, T2
 where T1.a = T2.b
```

```
select ...
  from T1 JOIN T2
       on T1.a = T2.b
```



- Example join

```
hive> SELECT o.state as state, count(*) as employees
  FROM office o LEFT OUTER JOIN users  u
     on u.office_id = o.office_id
GROUP BY o.state
ORDER BY state;
```

- Note Hive only supports equi-joins
- Inner Join, Left Outer, Right Outer, Full Outer, Left Semi-Join are supported
- Largest table on right for performance
- Limited support for subqueries, only permitted in the FROM clause of a SELECT statement.
  - Sometimes can be rewritten using Joins

Order BY / SORT BY

- ORDER BY clause is similar to other versions of SQL
  - Performs total ordering of result set - through single reducer (Hadoop)
  - Large data sets = high latency
- Hive's SORT BY clause
  - Data is ordered in each reducer - each reducer's output is sorted
  - If multiple reducers - final results may not be sorted as desired

Casting

- Implicit conversions when compare one type to another
- Use the cast() function for explicitly conversions
- If salary is a STRING data type, we can explicitly cast it to FLOAT as seen below

```
hive> SELECT first_name, team
      FROM players
      WHERE cast(salary AS FLOAT) > 100.0;
```

If "salary" not a string that could be converted to a floating point number, then Null

Views

- Allow query to be saved and treated like table
- Logical – doesn't store data
- Hides complexity of long query
- Earlier example of a "complex" query

```
hive> FROM(
        SELECT cust_name, round(ytd_sales * .01) as rewards FROM customer;
      ) c
      SELECT c.name, c.rewards
      WHERE c.rewards > 25;
```

Can be simplified using a View....

```
hive> CREATE VIEW rewards_view AS
       SELECT cust_name, round(ytd_sales * .01) as rewards FROM customer;

hive> SELECT name, rewards FROM rewards_view
      WHERE rewards > 25;
```

- Drop a view

```
hive> DROP VIEW IF EXISTS rewards_view;
```

Explain

- The explain keyword generates a Hive explain plan for the query

```
hive> EXPLAIN SELECT o.state as state, count(*) as employees
  FROM office o LEFT OUTER JOIN users  u
     on u.office_id = o.office_id
GROUP BY o.state
ORDER BY state;

STAGE DEPENDENCIES:
  Stage-1 is a root stage
  Stage-2 depends on stages: Stage-1
  Stage-3 depends on stages: Stage-2
  Stage-0 is a root stage
...
```

- Query requires three MapReduce jobs to implement
  - One to join the two inputs
  - One to perform the GROUP BY
  - One to perform the final sort
- FYI – there also is the EXPLAIN EXTENDED keyword which gives even more detail
- Details are beyond the scope of this presentation

If you know SQL already, then HiveQL's DML has relatively few surprises. We'll cover just a portion of HiveQL here in this presentation, trying to point out some things that make HiveQL different from the SQL you have previously worked with. As we look at the SQL it's easy to forget that we aren't just working with a normal RDBMS here - behind the covers we are taking advantage of Hadoop's ability to process massive distributed datasets. The first example here is a simple SELECT
query. We are selecting all the columns from the users table and limiting the output to a max of 3 rows. The children column is of the
array data type - notice that Hive outputs the array elements in brackets.
The second example here is a query that selects two columns - the name column and the first (zero) element from the children array. The last example shows a select statement

whose output includes a STRUCT column. You can see that this address column of data type STRUCT prints out in JSON format. Also note that the MAP data type in Hive is also output in JSON format.
The WHERE clause is similar to what you've used in other SQL dialects. The predicate operators often used in a WHERE clause are listed here and include equal, not equal, greater than, less than, and so on. The first statement includes a WHERE clause - we also show how the data within a STRUCT column can be accessed using the dot notation. The second example is a statement that includes a GROUP BY clause. GROUP BY is usually used with aggregate functions. You also can use the HAVING clause with GROUP BY.
In HQL we can use column aliases. In the first example address.city is given the column alias of "city". Column aliases come in handy when doing nested select statements. In our second example we select a customers name, and compute their year to date rewards (1% of their sales) which is aliased with the word "rewards". We then alias that whole first query with the letter c. The final SELECT clause then selects just the customers who have earned over $25 of rewards.
Now let's look at how we can select data from partitions. In order to take advantage of partitions it is important to understand how Hive handles them under the covers. Depending on your query, Hive performs partition pruning where it only scans a fraction of the table relevant to the partitions specified by the query. Hive will do the partition pruning if the partition predicates are specified in the WHERE clause or the ON clause in a JOIN. Let's check out an example. If we have a table called page_views that is partitioned on the log_date column, the listed query will retrieve rows for just the days between 2008- 03-01 and 2008-03-31. Imagine that we had a year of page_view data partitioned into 365 separate partition files on HDFS. Hive will only need to open and read the 31 partition data files for the days in the month of March 2008. If we hadn't used partitions, each time we ran this query Hive would need to read all 365 files to satisfy our request.
Hive supports joins via ANSI join syntax. Hive supports Inner, Left Outer, Right Outer, Full Outer and left Semi-Joins. We list an example join here that utilizes the left outer join. Note that Hive only supports equi-joins. Hive also has limited support for subqueries - they are only permitted in the FROM clause of a SELECT statement.
Another useful Hive tidbit is that Hive is optimized to have the largest table on the right in your Join query. The table on the right will be streamed from disk and the other Join tables will attempt to be read from memory - so keep that in mind when creating your queries. There also is a hint that can be specified to tell Hive which table to stream. There's a variety of Join performance tricks for you to investigate. If at least one of the Join tables is small, it is possible to do a MapJoin in Hive.
If the tables being joined are bucketized on the join columns, and the number of buckets in one table is a multiple of the number of buckets in the other table, the buckets can be joined with each other. This is called a Bucket Map Join.
In Hive the ORDER BY clause is similar to other versions of SQL. The ORDER BY clause performs a total ordering of the result set - all through a single reducer. For large data sets this can mean high latency. We can contrast that with Hive's SORT BY clause. Using the SORT BY causes the data to be ordered in EACH reducer - so each reducers output is sorted. If there are multiple reducers then the final results may not be sorted as desired. Hive 0.12 introduced the capability to do a parallel order by, however the feature is still young.

In addition to the implicit conversions Hive will do when you compare one type to another, it is possible to use the cast() function to explicitly convert a vale of one type to another. In this example we cast the salary
which is of STRING data type to a FLOAT. We use the cast function to explicitly do this. If one of the salaries happened to contain a
persons name or some other text accidentally and not a representation of a number, then Hive would return Null after attempting to do the cast.
A View in Hive allows a query to be saved and treated like a table. A view in Hive is simply logical - it does not actually store data. In other words just the view query is stored in the Hive metastore. The view query is not run until we use the view in another query. The nice thing about using views is it can help us handle the complexity of longer queries. Here we have an example of a query that contains
a subquery. We can simplify this by using a view. We first create a view using
the CREATE VIEW clause. We can then run another query that selects from the results of that view. Dropping a view is done using the
Drop View statement.
Sometimes it is very useful to be able to generate a Hive query plan to see what is going on behind the covers of Hive and Hadoop. We
can do this using the EXPLAIN keyword. While the details of EXPLAIN are a bit out of the scope of this presentation, let's check out an example. We see the word EXPLAIN in front of our SELECT query. Hive outputs the explain plan showing us the stage dependencies and a bunch of other information. From this Explain information, we can figure
out how many MapReduce jobs Hive is creating to fulfill our request.
You have now completed this topic. Thank you for watching.

>> Lab:

Sample Data Descriptions

For your convenience, the metadata for the sample data files is listed out here again.

Our sample data is from a fictitious computer retailer. The company sells computer parts and generally serves a single State in the country.

Customer.csv:

```
Purpose: Hold customer records.

Columns:
FNAME                    Customer's First Name
LNAME                    Customer's Last Name
STATUS                   Active or Inactive status
TELNO                    Telephone # Customer's unique ID
CUSTOMER_ID CITY|ZIP     City and Zip code separated by the "|" character.
```

Product.csv:

```
Purpose: Hold product records.

Columns:
PROD_NAME                   Name of product
DESCRIPTION                 Description of computer product
CATEGORY                    Category product belongs to
QTY_ON_HAND                 Quantity of product in warehouse
PROD_NUM                    Unique product number
PACKAGED_WITH               Colon separated list of things that come in package with product.
```

Sales.csv:

```
Purpose: Holds all historical sales records. Company updates once a month.

Columns:
CUST_ID                     ID of customer who made purchase
PROD_NUM                    ID of product that was purchased
QTY                         QTY purchased
DATE                        Date of sale
SALES_ID                    Unique sale ID
```

Loading Data into the Managed Non-Partitioned Tables

The first table you created in the previous lab was the products table. This table is fully managed by Hive and does not cont in the local /tmp/tempdata/hive_data/Product.csv file on HDFS.

```
%jdbc(hive)
LOAD DATA INPATH '/tmp/tempdata/hive_data/Product.csv' OVERWRITE INTO TABLE computersalesdb.products
```

Hive copies the data from the file. Our products table now has data in it. Let's check it out!

List out the contents of the /user/hive/warehouse/computersalesdb.db/products directory. You will see the Product.csv file is there within that directory.

```
%sh
ssh  root@hive 'su - root bash -c "hdfs dfs -ls /user/hive/warehouse/computersalesdb.db/products"'
```

You can easily view the contents of the Product.csv file by running the **hdfs dfs -cat** command.

```
%sh
ssh  root@hive 'su - root bash -c "hdfs dfs -cat /user/hive/warehouse/computersalesdb.db/products/Product.csv"'
```

Look at the data displayed - our products table is now loaded into Hive!

Next you will load sales records into the sales_staging table.

Look at the data displayed - our products table is now loaded into Hive!

Next you will load sales records into the sales_staging table.

```
%jdbc(hive)
LOAD DATA INPATH '/tmp/tempdata/hive_data/Sales.csv' INTO TABLE computersalesdb.sales_staging
```

Notice you left off the OVERWRITE keyword in this statement. This is because we normally would want to add the monthly sales data to our historical list of records already in the sales
We would not want to overwrite all the records in that table with just this month's data.

Again, let's verify that the data is now within the Hive warehouse on HDFS, by checking out the file using the hdfs dfs –cat command.

```
%sh
ssh  root@hive 'su - root bash -c "hdfs dfs -cat /user/hive/warehouse/computersalesdb.db/sales_staging/Sales.csv"'
```

Loading Data into the Managed Partitioned Table

READY

Now that the sales_staging table has data you can work with, let's write some queries that will allow you to load the partitioned sales table (partitioned on sales_date) with data coming from sales_staging

Load sales data from '2012-01-09' into a partition of the sales table.

READY

```
%jdbc(hive)
INSERT OVERWRITE TABLE computersalesdb.sales
    PARTITION (sales_date = '2012-01-09')
    SELECT cust_id, prod_num, qty, sales_id
           FROM computersalesdb.sales_staging ss
           WHERE ss.sale_date = '2012-01-09'
```

READY

Following the same procedures, load sales data from '2012-01-24' into a new partition of the sales table.

```
%jdbc(hive)
INSERT OVERWRITE TABLE computersalesdb.sales
    PARTITION (sales_date = '2012-01-24')
    SELECT cust_id, prod_num, qty, sales_id
           FROM computersalesdb.sales_staging ss
           WHERE ss.sale_date = '2012-01-24'
```

Run the hdfs dfs -ls command and take a look in the sales folder now. You will see a new subdirectory named sales_date=2012-01-24. Within that directory is a data file named 00000(
If you cat that file you will see that it only contains the data for our 2012-01-24 sales.

```
%sh
ssh  root@hive 'su - root bash -c "hdfs dfs -ls /user/hive/warehouse/computersalesdb.db/sales"'
ssh  root@hive 'su - root bash -c "hdfs dfs -ls /user/hive/warehouse/computersalesdb.db/sales/sales_date=2012-01-24"'
ssh  root@hive 'su - root bash -c "hdfs dfs -cat /user/hive/warehouse/computersalesdb.db/sales/sales_date=2012-01-24/000000_0"'
```

Running Queries

Selecting Data

Now that the tables have data in them, let's start running queries against that data.

Select all the data in the products table where the product category is 'Video'. Order the results by prod_num.

```
%jdbc(hive)
SELECT * FROM computersalesdb.products WHERE category= 'Video' ORDER BY prod_num
```

Now select all the products where category='Video' AND the first element of the PACKAGED_WITH array contains 'dvd'.

```
%jdbc(hive)
SELECT * FROM computersalesdb.products WHERE category= 'Video' AND PACKAGED_WITH[0]= 'dvd'
```

Find out how many products there are for each category of item by using the GROUP BY clause.

```
%jdbc(hive)
SELECT category, count(*) FROM computersalesdb.products GROUP BY category
```

Use a nested select to show the product categories that contain more than 3 products.

```
%jdbc(hive)
FROM(
    SELECT category, count(*) as count FROM computersalesdb.products
    GROUP BY category) cats
SELECT * WHERE cats.count > 3
```

Taking Advantage of Partitioned Data

READY

The sales table is partitioned on sales_date. In a previous exercise you loaded data into two partitions of this table (2012-01-09 and 2012-01-24 partitions). Let's take advantage of this partitioni improve latency.

Run a SELECT query that finds only the sales that occurred on 2012-01-24. Order the results by the sale_id.

READY

```
%jdbc(hive)
SELECT * FROM computersalesdb.sales
WHERE sales_date = '2012-01-24'
ORDER BY sales_id
```

Above, you will see that just the 2012-01-24 sales records were returned.

READY

Only the sales_id=2012-01-24 partition file had to be read in to complete this query, saving you some wait time – theoretically a lot of wait time if you had a large number of historical sales data.

Joins

Let's show all Optical category sales that occurred on 2012-01-09. You will need to do an equi-join between the sales and products tables to gather this information.

```
%jdbc(hive)
SELECT s.cust_id, s.prod_num, s.qty, s.sales_id,
      p.prod_name, p.category
      FROM computersalesdb.sales s JOIN computersalesdb.products p
          ON s.prod_num = p.prod_num
      WHERE s.sales_date = '2012-01-09' AND p.category = 'Optical'
```

The join was successful. You got the 3 records we were looking for.

Views

Now let's create a View that stores a query which returns all the sales records (joined with products table) where the product category is 'Optical'.

```
%jdbc(hive)
CREATE VIEW optical_sales AS
        SELECT s.cust_id, s.prod_num, s.qty, s.sales_id,
        p.prod_name, p.category
        FROM computersalesdb.sales s JOIN computersalesdb.products p
            ON s.prod_num = p.prod_num
        WHERE p.category = 'Optical'
```

The View was successfully created.

Now that you have the optical_sales view, you can use it within other queries, just like it were a table. Notice how short this query is.

```
%jdbc(hive)
SELECT * FROM optical_sales
WHERE  qty > 1
```

Exporting Data READY

Imagine your management team wants you to extract all sales of Optical devices and give it to them in a format outside of the Hive environment. You could do this by exporting the data from Hive

You are going to create a new directory on HDFS called "reports". Create this directory within the /tmp directory on HDFS.

```
%sh
ssh  root@hive 'su - root bash -c "hdfs dfs -mkdir /tmp/reports"'
```

READY

Make this new folder writable by all users. READY

```
%sh
ssh  root@hive 'su - root bash -c "hdfs dfs -chmod 777 /tmp/reports"'
ssh  root@hive 'su - root bash -c "hdfs dfs -ls /tmp"'
```

READY

Now utilize the previous query and write the data to the /tmp/reports directory on the HDFS file system.

```
%jdbc(hive)
INSERT OVERWRITE DIRECTORY '/tmp/reports'
SELECT * FROM optical_sales WHERE  qty > 1
```

Now check the HDFS /tmp/reports directory.

```
%sh
ssh  root@hive 'su - root bash -c "hdfs dfs -ls /tmp/reports"'
```

Notice there is one new file called 000000_0! This is the data that Hive wrote out.

Now cat this file and see what the contents look like.

```
%sh
ssh  root@hive 'su - root bash -c "hdfs dfs -cat /tmp/reports/000000_0"'
```

Explain

Let's take a brief look at using EXPLAIN in a Hive query.

You will have Hive explain the execution plan for a simple query that selects all customer records.

```
%jdbc(hive)
EXPLAIN SELECT * FROM computersalesdb.customer
```

MODULE 4 – HIVE OPERATORS AND FUNCTIONS

Relational Operators

- Passed operands compared, generates a TRUE or FALSE value

Operator	Operand types	Description
A = B	all primitive types	TRUE if expression A is equivalent to expression B otherwise FALSE
A != B	all primitive types	TRUE if expression A is *not* equivalent to expression B otherwise FALSE
A < B	all primitive types	TRUE if expression A is less than expression B otherwise FALSE
A <= B	all primitive types	TRUE if expression A is less than or equal to expression B otherwise FALSE
A > B	all primitive types	TRUE if expression A is greater than expression B otherwise FALSE
A >= B	all primitive types	TRUE if expression A is greater than or equal to expression B otherwise FALSE

More Relational Operators...

Operator	Operand types	Description
A IS NULL	all types	TRUE if expression A evaluates to NULL otherwise FALSE
A IS NOT NULL	all types	FALSE if expression A evaluates to NULL otherwise TRUE
A LIKE B	strings	TRUE if string A matches the SQL simple regular expression B, otherwise FALSE. Comparison done character by character.
A RLIKE B	strings	NULL if A or B is NULL, TRUE if any substring of A matches the Java regular expression B (otherwise FALSE).
A REGEXP B	strings	Same as RLIKE

Relational Operator example

LIKE / RLIKE

- Very useful when searching STRING fields
- **Find all users with first name beginning with "Je"**

```
hive> SELECT id, name FROM users WHERE name LIKE 'Je%';

      19 Jessica
      24 Job
      56 Jenn
      ...
```

- RLIKE similar – allows search using regular expressions

Arithmetic Operators

Operator	Operand types	Description
A + B	all number types	A and B added together.
A – B	all number types	B subtracted from A.
A * B	all number types	A multiplied by B.
A / B	all number types	A divided by B.
A % B	all number types	Remainder of A divided by B.
A & B	all number types	Bitwise AND of A and B.
A | B	all number types	Bitwise OR of A and B.
A ^ B	all number types	Bitwise XOR of A and B.
~A	all number types	Bitwise NOT of A.

Logical Operators

Operator	Operand types	Description
A AND B A && B	boolean	If A and B are both TRUE, then this returns TRUE – else returns FALSE.
A OR B A || B	boolean	If either A or B or both are TRUE, then this returns TRUE – else returns FALSE.
NOT A !A	boolean	If A is FALSE, then this returns TRUE – else returns FALSE .

Operators on Complex Types

Operator	Operand types	Description
A[n]	A is an Array and n is an int	Returns the nth element in the array A. The first element has index 0.
M[key]	M is a Map<Key, Value> and key has type K	Returns the value corresponding to the key in the map.
S.x	S is a struct	Returns the x field of struct S.

Built-in Functions

- Wide variety of functions built into Hive

Return Type	Function name (Signature)	Returns….
BIGINT	round(double a)	rounded BIGINT value of the double
BIGINT	floor(double a)	maximum BIGINT value that is equal or less than the double
BIGINT	ceil(double a)	minimum BIGINT value that is equal or greater than the double
STRING	substr(string A, int start)	substring of A starting from start position till the end of string A
STRING	upper(string A)	string resulting from converting all characters of A to upper case
INT	Length(string s)	length of the string
INT	year(string timestamp)	year part of a timestamp string
STRING	get_json_object(string json_string, string path)	Extracts JSON object from a JSON string based on JSON path specified, and return JSON string of the extracted JSON object.

More Built-in Functions…

Return Type	Function name (Signature)	Description
BIGINT	count(*)	Returns the total number of retrieved rows, including rows containing NULL values.
DOUBLE	sum(col)	Returns the sum of the elements in the group.
DOUBLE	avg(col)	Returns the average of the elements in the group.
DOUBLE	min(col) or max(col)	Returns the minimum or maximum value of the column in the group
N rows	explode(array)	Return zero or more rows – one row for each element from the input array.
N rows	explode(map)	Return zero or more rows – one row for each map key-value pair, with field for each map key and field for the map value

- Run the "SHOW FUNCTIONS" and "DESCRIBE FUNCTION" commands to see more!

Built-in Functions in Action

ourtable
string_column
numbers_column

```
Row1, 5
Row2, 2
Row3, 3
Row4, 4
Row5, 1
```

```
hive> SELECT sum(numbers_column) FROM ourtable;
        15
```

- As expected, Hive outputs the sum of the numbers_column

```
hive> SELECT count(*) FROM ourtable;
        5
```

- Hive outputs the count of the rows in the table

Extending Hive's Functionality

- **Custom User Defined Functions**
  - Create custom functions – implement your own logic
  - Implemented in Java
  - Add to Hive session and use like a built-in function
  - UDF, UDAF, UDTF

- **Streaming – Custom Map/Reduce Scripts**
  - Alternative way of transforming data
  - I/O pipe to external process
  - Data passed to process, operates, writes out
  - MAP(), REDUCE(), TRANSFORM() clauses provided

```
hive> SELECT TRANSFORM (foo, bar)
      USING '/bin/cat' AS newFoo, newBar
      FROM mydatabase.mytable;
```

Hi. Welcome to Accessing Hadoop Data Using Hive. In this lesson we will discuss Hive Operators and Functions.
After completing this lesson, you should be able to:
Use a variety of Hive Operators in your queries Utilize Hive's Built-in Functions
And Explain ways to extend Hive functionality
Let's kick things off by looking at the relational operators included in Hive. The passed relational operands are compared and a TRUE or FALSE value is generated. If you already know another SQL dialect, nothing here will shock you. We have an equal/not equal operator, greater than, less than, and greater than or equal, or less than or equal operators. If you'd like more time to review the contents
of the tables in this presentation, please pause the video at any point.
Here are some additional relational operators. We can check if a value is Null or not. We also can use LIKE and RLIKE to do comparisons on STRINGS.
Let's take a look at an example of a relational operator in action. The LIKE and RLIKE operators are very useful when searching STRING fields in your data. In this example we search for all users that have a name that begins with the letters "Je". You can see
our results include names like Jessica, Jeb, and
Jenn. We can use the RLIKE operator in the same
manner as the LIKE operator. RLIKE allows us to search on regular expressions which gives us even more power over our search query.
Hive also contains the arithmetic operators you are used to working with, allowing for addition, subtraction, multiplication, division, and so on.
The following table lists Hive's logical operators. These operators provide support for creating logical expressions. All of them return boolean TRUE or FALSE. We have the

AND, OR, and NOT operators available to us in Hive.
The operators in this table provide mechanisms
to access elements in Hive's Complex Types. These operators and complex types are not likely to be found in the traditional RDBMS. They are closer to what you'd find in a traditional programming language. Array elements in Hive are accessed with the bracket notation as shown. A reminder that array indexes are zero based.
Maps are also accessed using array bracket notation.
Structs are accessed using the dot notation.
Hive also has a variety of built-in functions that you can use in your queries. We list some of the commonly used functions in this table.
This table has a few interesting aggregate functions and table generating functions.
Count, Sum and Average are aggregate functions you are likely to use often.
The explode function allows you to take an array or map and explode it out into zero or more rows.
Please note that there are many more functions built into Hive - take the time to check some of them out. You can run the command "SHOW FUNCTIONS" from within the
Hive CLI to get a listing of built-in functions. You can then run the DESCRIBE FUNCTION command
to view the documentation for that function.
Hive includes windowing and analytic functions that operate as per the SQL standard.
Visit the Hive documentation for a more detailed explanation on how to work with these types of functions.
Here we have a couple examples of built-in Hive functions. To use a function in Hive you just call it by name in your query and pass any required arguments inside the parenthesis. The first query uses the sum function and the second query uses the count function. These both behave how we'd expect in SQL.
Sometimes in Hive we may want to have some functionality that goes beyond what the built-in functions provide. One way we can do this is by creating custom user defined function. UDFs are implemented in Java. Once your UDF Java code is complete you export it to a JAR file and add it to the Hive session where you can then use the function the same way you did the built-in functions. There are 3 different types of user defined functions you can create - regular old UDF's, user-defined aggregate functions, and user-defined
table generating functions. Streaming is an alternative way we can transform
data using Hive. When we run a streaming job, data is passed to an external process of our choosing. Our data is read, processed, and the results are written back out to us. Hive has three clauses - Map, Reduce, and Transform - that can be used for streaming. The Streaming example here is a very basic
one - we send data to the /bin/cat command which comes on most Linux systems. The /bin/cat/
command simply echoes the data sent to it. Hive will get the results back and
print them to the CLI.
You have now completed this topic. Thank you for watching.

>>Lab:

Operators

In this section you will experiment with Relational, Arithmetic, and Logical Operators in the Hive system.

Took 0 sec. Last updated by anonymous at April 29 2020, 10:17:10 AM.

Relational Operators

Use the IS NULL operator to find all records in the products table that have no item in the zero position of the packaged_with array.

Took 0 sec. Last updated by anonymous at April 29 2020, 10:17:10 AM.

```
%jdbc(hive)
SELECT * FROM computersalesdb.products WHERE packaged_with[0] IS NULL
```

Find customers who do not live in the 67890 zipcode. Do this using the != operator to examine the city_zip struct column of the customer table. Reminder: city_zip is described as struct in our table.

```
%jdbc(hive)
SELECT * FROM computersalesdb.customer WHERE city_zip.zip != '67890'
```

Use the LIKE operator to find all product descriptions that contain the word “Tiger”.

```
%jdbc(hive)
SELECT * FROM computersalesdb.products WHERE description LIKE '%Tiger%'
```

You can see that 6 records in our table contain the word Tiger in the description column.

Arithmetic Operators

Have Hive return the product name, qty, and product number of all the products the company sells. Add 10 to the quantities for each product.

```
%jdbc(hive)
SELECT prod_name, qty_on_hand + 10, prod_num FROM computersalesdb.products
```

You will notice that each quantity has 10 added to it.

Logical Operators

Find all the customers with last names of either “Merdec” OR “Melcic”.

```
%jdbc(hive)
SELECT * FROM computersalesdb.customer WHERE lname='Merdec' OR lname='Melcic'
```

Using the OR logical operator allowed you to retrieve the 5 records that match either of the last names we queried on.

Functions

In this section you will experiment with built-in Hive functions.

Let's find out more about the upper function. Run DESCRIBE FUNCTION upper in the Hive CLI.

```
%jdbc(hive)
DESCRIBE FUNCTION upper
```

Convert product category to upper case and find products with a category equal to "CASE".

```
%jdbc(hive)
SELECT * FROM computersalesdb.products WHERE upper(category) = 'CASE'
```

Let's explode the array for the product with prod_num=98820.

```
%jdbc(hive)
SELECT explode(packaged_with) as package_contents FROM computersalesdb.products WHERE prod_num='98820'
```

The array was exploded out and each array element is returned as an individual row.