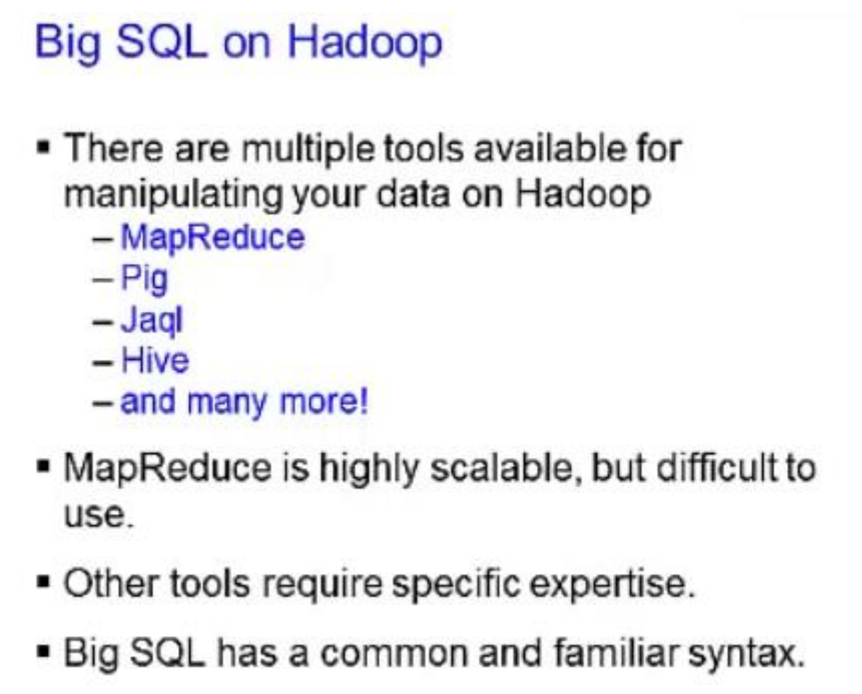
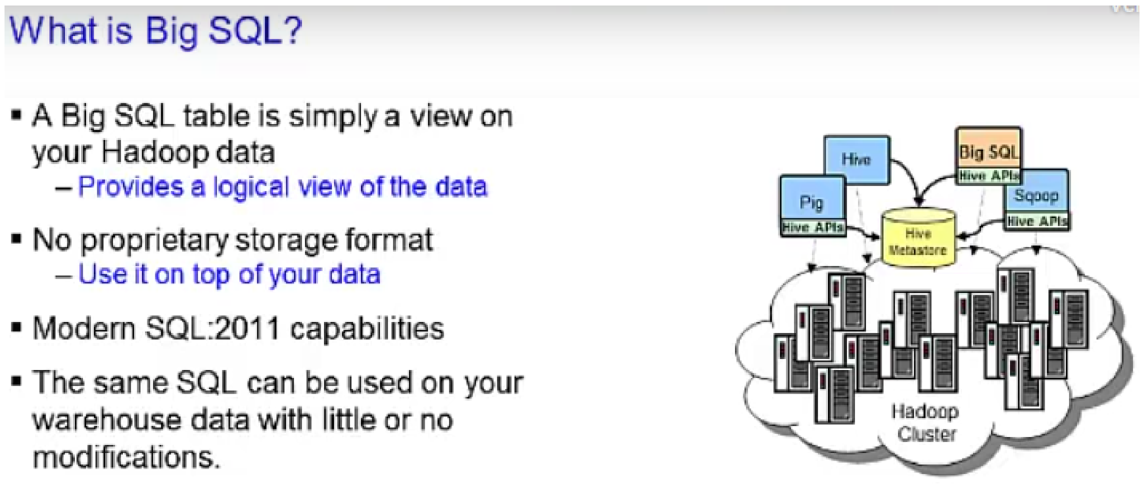
MODULE 1 - BIG SQL OVERVIEW



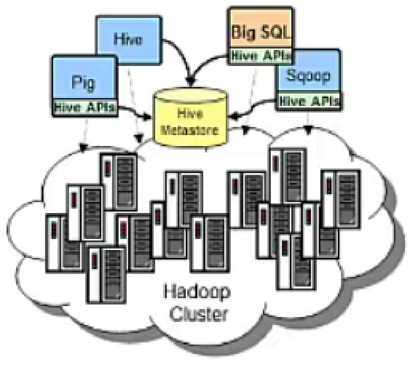
Big SQL on Hadoop

- There are multiple tools available for manipulating your data on Hadoop
  - MapReduce
  - Pig
  - Jaql
  - Hive
  - and many more!
- MapReduce is highly scalable, but difficult to use.
- Other tools require specific expertise.
- Big SQL has a common and familiar syntax.



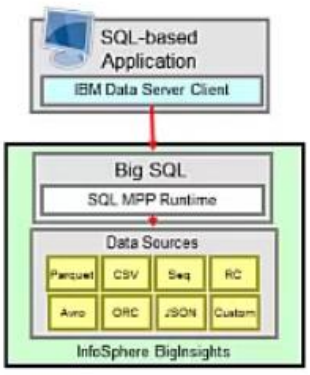
What is Big SQL?

- A Big SQL table is simply a view on your Hadoop data
  - Provides a logical view of the data
- No proprietary storage format
  - Use it on top of your data
- Modern SQL:2011 capabilities
- The same SQL can be used on your warehouse data with little or no modifications.

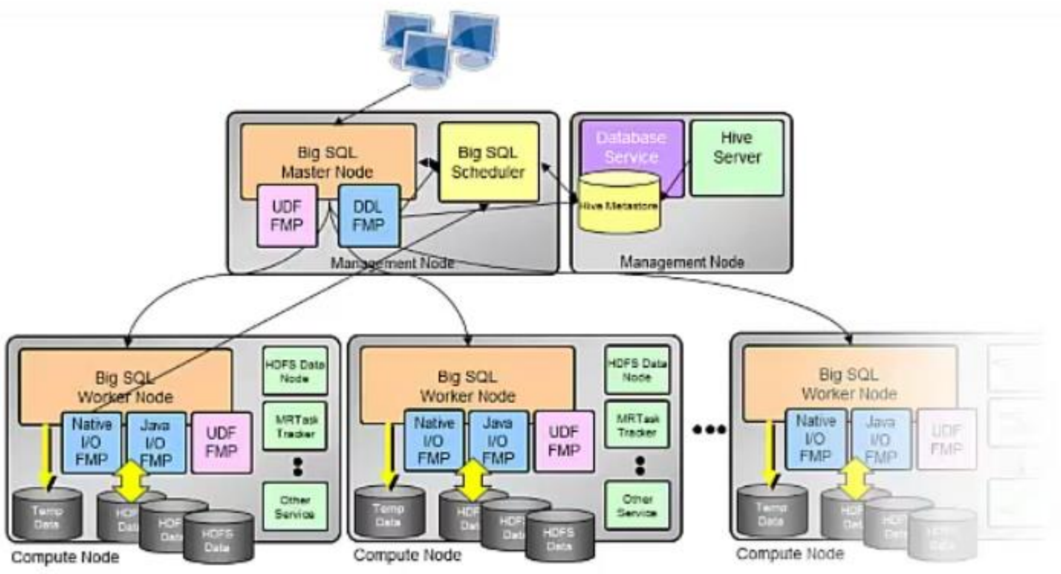


Architected for performance

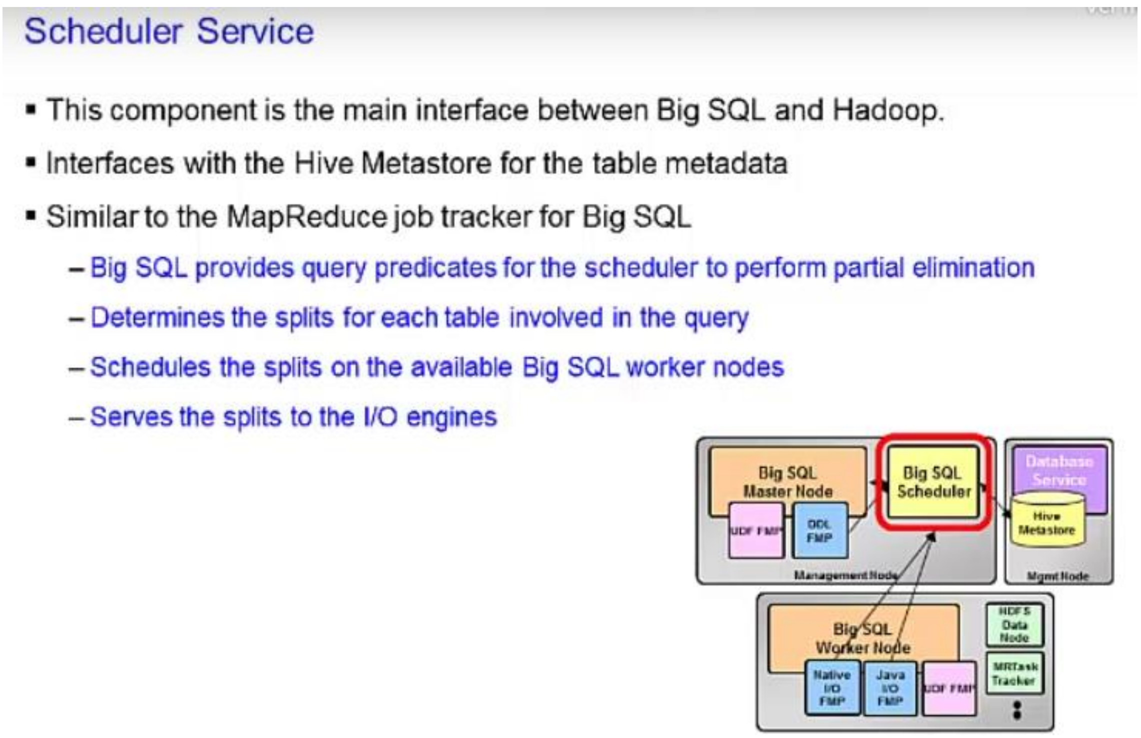
- MapReduce is replaced with a modern, massively parallel processing (MPP) architecture
  - Compiler and runtime are native code (not Java)
  - Big SQL worker daemons live directly on cluster
    - Continuously running (no startup latency)
    - Processing happens locally at the data
  - Message passing allows data to flow directly between nodes
- Operations occur in memory with the ability to spill to disk
  - Supports aggregations and sorts larger than available RAM



Architecture



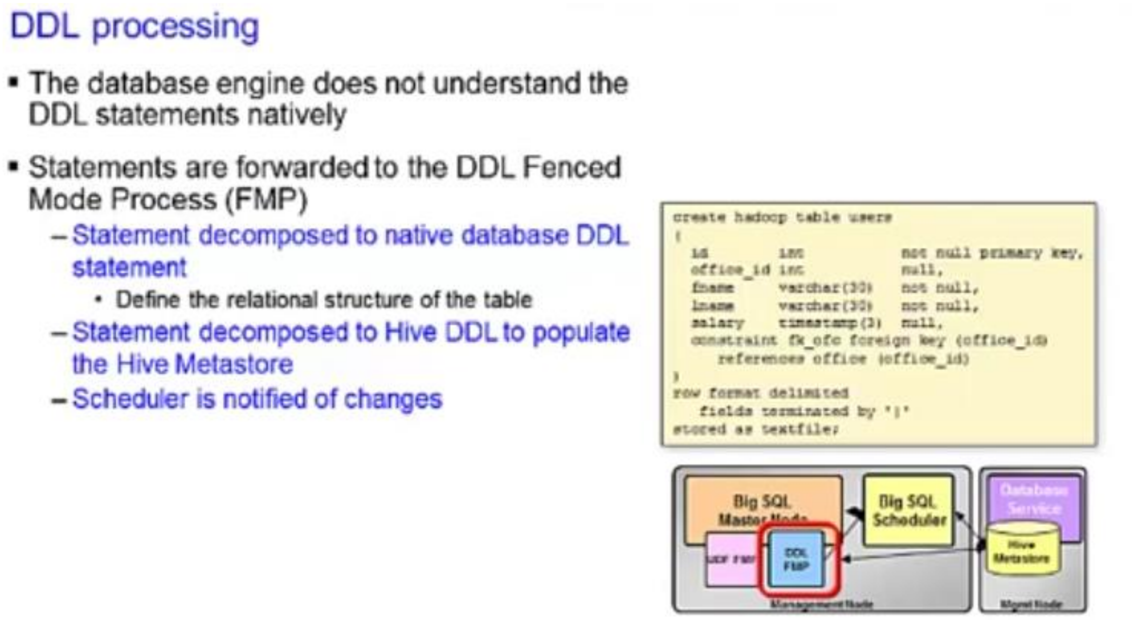
*FMP = Fenced mode process



Scheduler Service

- This component is the main interface between Big SQL and Hadoop.
- Interfaces with the Hive Metastore for the table metadata
- Similar to the MapReduce job tracker for Big SQL
  - Big SQL provides query predicates for the scheduler to perform partial elimination
  - Determines the splits for each table involved in the query
  - Schedules the splits on the available Big SQL worker nodes
  - Serves the splits to the I/O engines

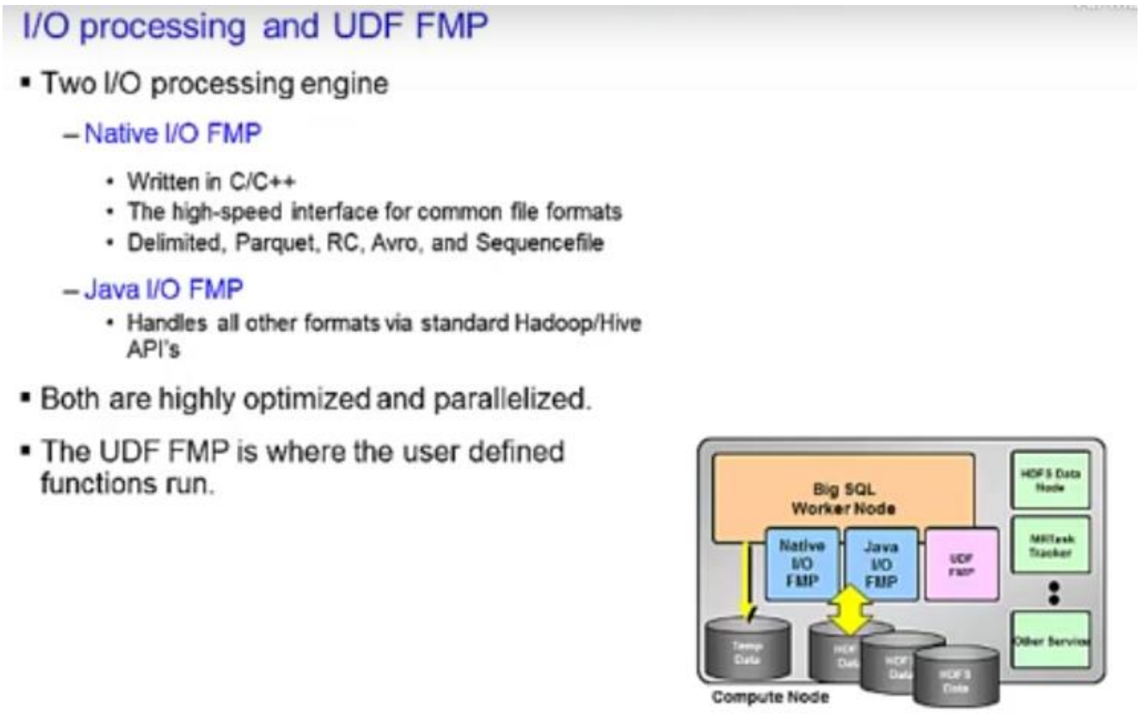
Big SQL Master Node
UDF FMP
DDL FMP
Management Node
Big SQL Scheduler
Database Service
Hive Metastore
Mgmt Node
Big SQL Worker Node
Native I/O FMP
Java I/O FMP
UDF FMP
HDFS Data Node
MRTask Tracker



DDL processing

- The database engine does not understand the DDL statements natively
- Statements are forwarded to the DDL Fenced Mode Process (FMP)
  - Statement decomposed to native database DDL statement
    - Define the relational structure of the table
  - Statement decomposed to Hive DDL to populate the Hive Metastore
  - Scheduler is notified of changes

```
create hadoop table users
(
  id         int            not null primary key,
  office_id  int            null,
  fname      varchar(30)    not null,
  lname      varchar(30)    not null,
  salary     timestamp(3)   null,
  constraint fk_ofc foreign key (office_id)
     references office (office_id)
)
row format delimited
   fields terminated by '|'
stored as textfile;
```



I/O processing and UDF FMP

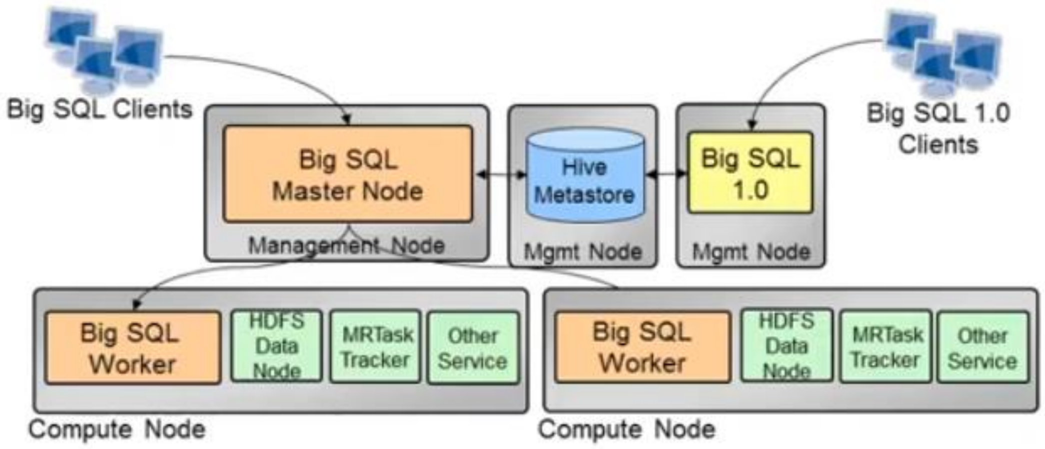
- Two I/O processing engine
  - Native I/O FMP
    - Written in C/C++
    - The high-speed interface for common file formats
    - Delimited, Parquet, RC, Avro, and Sequencefile
  - Java I/O FMP
    - Handles all other formats via standard Hadoop/Hive API's
- Both are highly optimized and parallelized.
- The UDF FMP is where the user defined functions run.

Welcome to the course, Big SQL on Hadoop.
In this lesson, Big SQL Overview, you will get an introduction to Big SQL and see how to use Big SQL on Hadoop data.
After completing this lesson, you will be able to describe the Big SQL architecture.
Know the differences between the current Big SQL and its older version (we will refer to this as Big SQL v1). List some of the Big SQL features. Understand Big SQL terminologies.
Understand and use different methods for working with Big SQL.
There are multiple tools available for you to work with your data on Hadoop. You have your MapReduce, which is highly scalable, but requires the user to understand the Java API and have some programming expertise. What if you do not want anything to do with Java? Well, then you can learn or use Pig, Jaql, Hive, or a number of other tools to work with your data. No? You do not want to learn anything new? Good news, Big SQL provides a common and familiar syntax that lets you work with your data. You can query your data residing in Hadoop using Big SQL. There is no learning curve here. Big SQL is about applying SQL to your existing data -- there are no proprietary storage formats.
What is Big SQL? Simply put, Big SQL is a logical view on top of your Hadoop data. The data resides in Hadoop and all you have to do is use Big SQL to access it. There is no need to change the format, or migrate the data out of Hadoop to do any work on the data.
Big SQL supports modern SQL:2011 capabilities. So, when you migrate your data into Hadoop, the same SQL can be used on your warehouse data with little or no modification. That is one of the big benefits of Big SQL, no need to learn anything new, and you can use your existing queries.
Big SQL is designed for performance. It replaces MapReduce using a modern, massively parallel processing or MPP architecture. The compiler and runtime are written in native code, C/C++, so it is much faster. The SQL engine pushes down the processing to the same node the holds the data so all the processing happens locally at the data. There is also no startup latency -- the daemons are continuously running. The operations occur in memory and if necessary, it can spill over to disk for processing. This allows for support of aggregations and sorts larger than the available RAM.

In this diagram here, you can see the Big SQL daemons that run on each node in the cluster, in orange. You have one master node, the one up at the top. The master node is responsible for accepting client connections, accepting queries, doing all the queries planning, query optimization and then pushing the query execution down to the worker nodes below. The other
colored boxes are the independent support processes that aid Big SQL in its processing.
At the top, right corner, you can see that there is the Hive Metastore involved. Hive is still using it. Big SQL communicates and shares information with Hive.
The scheduler is the main interface between
Big SQL and Hadoop. Remember that Big SQL provides a logical view of the files stored on Hadoop, so it needs to know which file constitutes a "table", how many files there are, type of files, etc. It communicates with the scheduler to ask these questions.
The scheduler also acts like the MapReduce job tracker. For example, when Big SQL gets a query, it tells the scheduler which table it is going to scan and the query predicates to let the scheduler figure out in how many pieces the work can be split. Then that information is passed back to the Big SQL master node to let it run. The worker nodes also communicate with the scheduler to figure out on which splits it needs to work at runtime.
There is a DDL processing engine as you can
see in the diagram. This engine is responsible for taking a statement and turning it into something Hive and Big SQL can understand. It receives the statements from the native database engine to decompose it to something the database can understand. It decomposes to Hive DDL in order to populate the Hive Metastore. Then the scheduler is notified of the new table. One thing to note is that the DDL is run in a fenced mode process, or FMP, which is an isolated process to ensure that it does not affect the other processes.
Here we have our I/O processing engine. With
Big SQL, we have two processing engines. We have a native I/O engine. The code for this engine is written in C/C++, and is highly optimize. It only knows a specific set of file formats as listed here. We also wanted to handle any Hive tables (defined in the Hive catalog). If we see a particular table that isn't recognized, we hand it down to the Java I/O layer. The Java I/O layer handles all the other formats via the standard Hadoop/Hive
API's. Both of these engines are optimized and parallelized,
with multiple threads doing the I/O. Also in this diagram, note the UDF FMP, the pink box. The user defined functions also run in a separate process in order to isolate the database engine from the user code to prevent the user from doing any harm, just as with the DDL processing engine in the previous slide.

Big SQL vs Big SQL v1

- Big SQL is a complete rewrite/redesign from the previous version.
- Why two different versions?
  - Officially, there is only one release, and that is the Big SQL
    - You may see it being refer to as Big SQL 3.0 in some cases
  - BigInsights 3.0 comes with Big SQL
  - Big SQL v1 or 1.0 is used to refer to the previous version



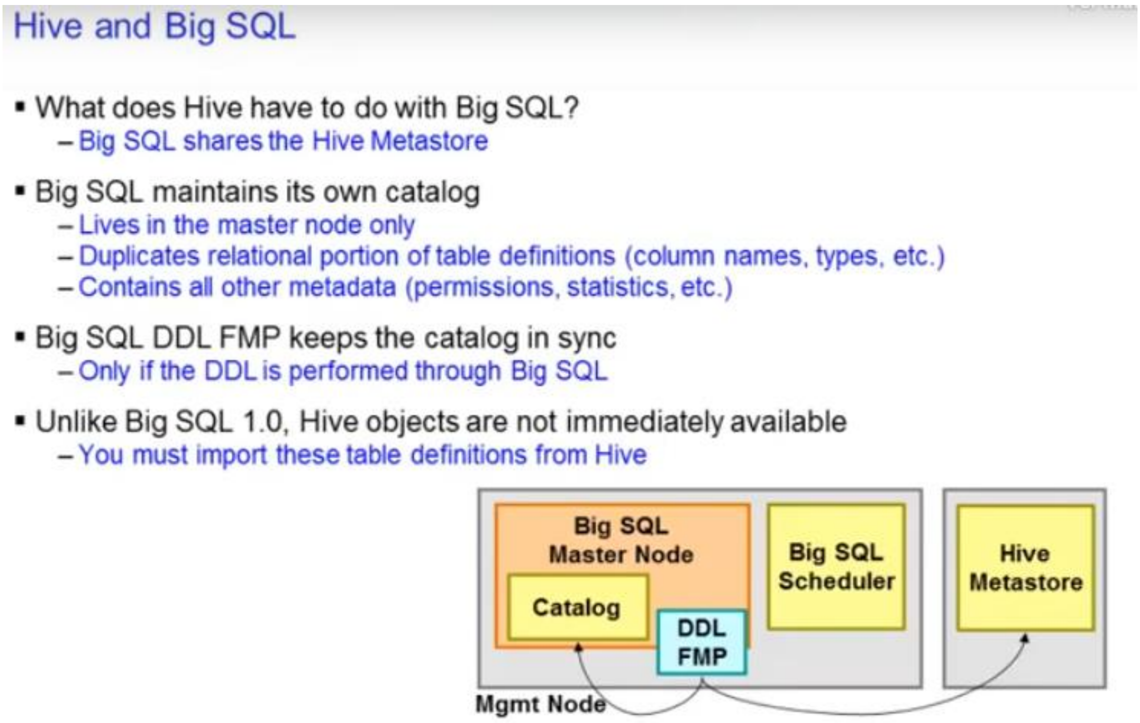
Big SQL vs Big SQL v1 - some differences

- Big SQL does not have support for HBase
  - Need to use the driver for Big SQL v1
- DDL in Big SQL is a largely a superset of Big SQL 1.0
- Big SQL 1.0 treated single and double quotes as the same
  - Big SQL reserves double quotes for identifiers (SQL standard rules)
- The hadoop keyword is required for creating tables in Big SQL

```
CREATE HADOOP TABLE T1
(
   C1 VARCHAR(30) NOT NULL
)
ROW FORMAT DELIMITED;
```

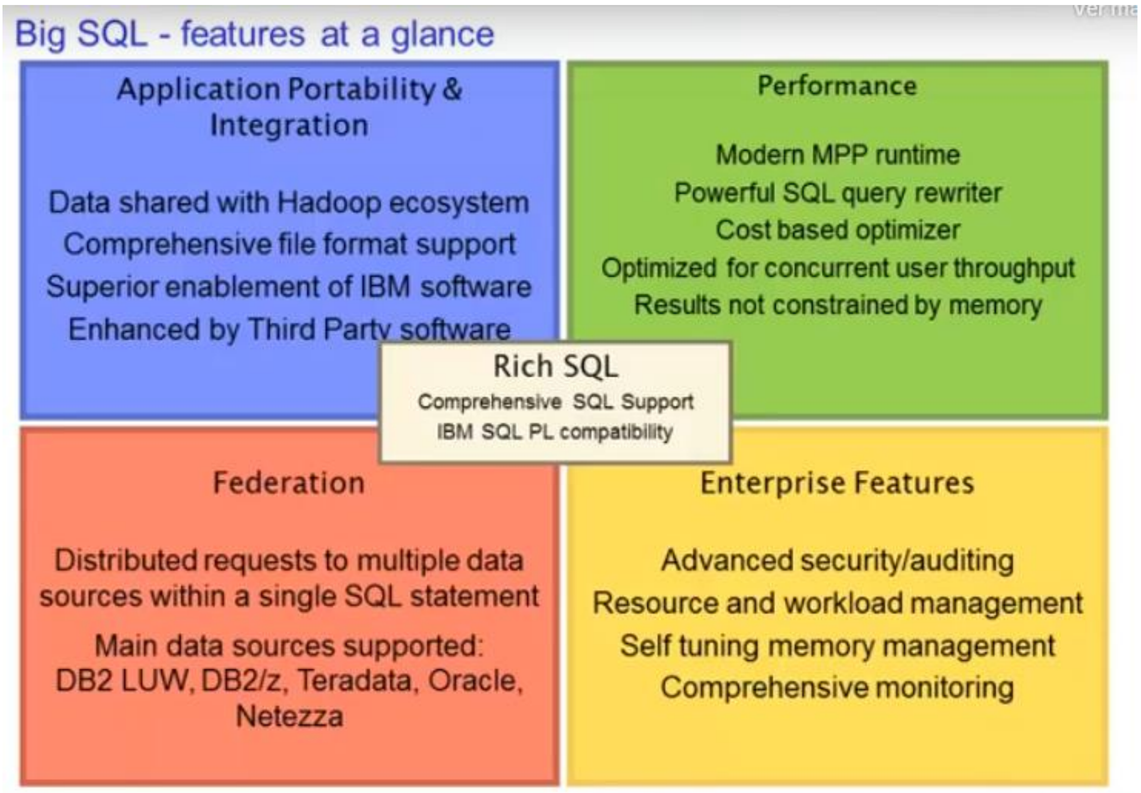
- The need for the hadoop keyword can be lifted using compatibility mode:

```
1> SET SYSHADOOP.COMPATIBILITY_MODE=1;
1> CREATE TABLE T1
2> (
3>   C1 VARCHAR(30) NOT NULL
4> )
ROW FORMAT DELIMITED;
```



Hive and Big SQL

- What does Hive have to do with Big SQL?
  - Big SQL shares the Hive Metastore
- Big SQL maintains its own catalog
  - Lives in the master node only
  - Duplicates relational portion of table definitions (column names, types, etc.)
  - Contains all other metadata (permissions, statistics, etc.)
- Big SQL DDL FMP keeps the catalog in sync
  - Only if the DDL is performed through Big SQL
- Unlike Big SQL 1.0, Hive objects are not immediately available
  - You must import these table definitions from Hive



Big SQL - features at a glance

Application Portability & Integration

Data shared with Hadoop ecosystem
Comprehensive file format support
Superior enablement of IBM software
Enhanced by Third Party software

Performance

Modern MPP runtime
Powerful SQL query rewriter
Cost based optimizer
Optimized for concurrent user throughput
Results not constrained by memory

Rich SQL

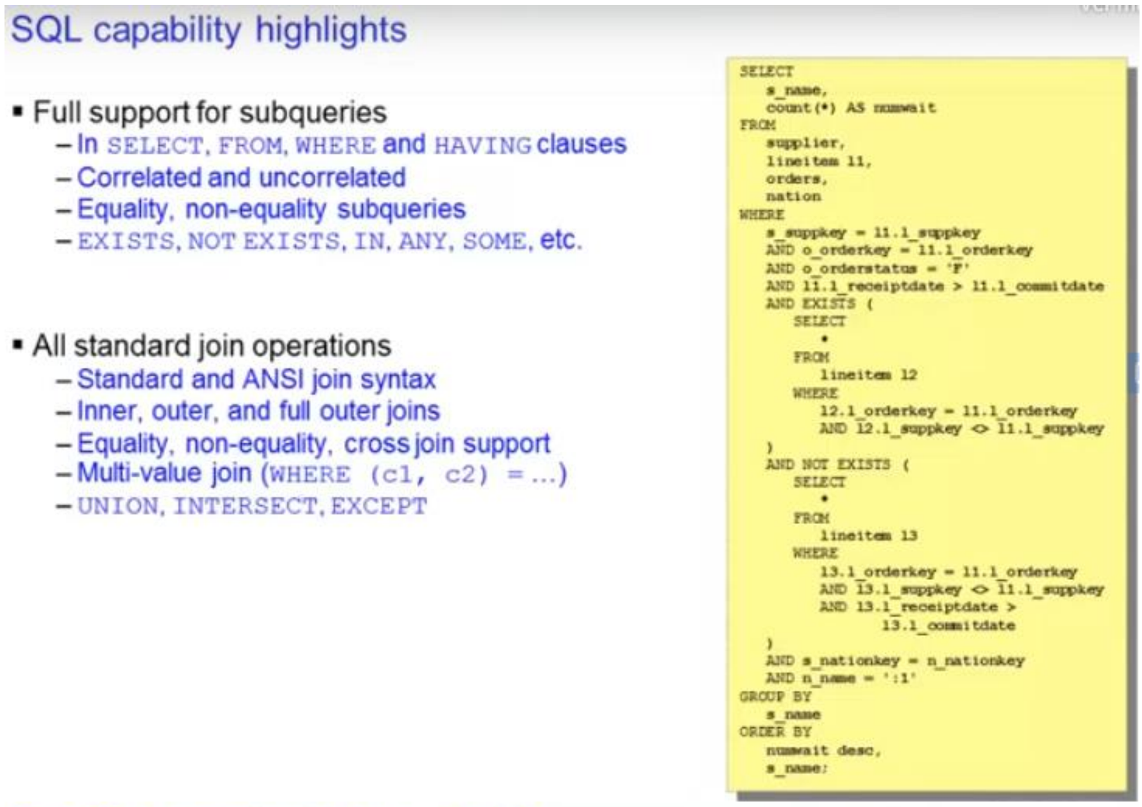
Comprehensive SQL Support
IBM SQL PL compatibility

Federation

Distributed requests to multiple data sources within a single SQL statement
Main data sources supported: DB2 LUW, DB2/z, Teradata, Oracle, Netezza

Enterprise Features

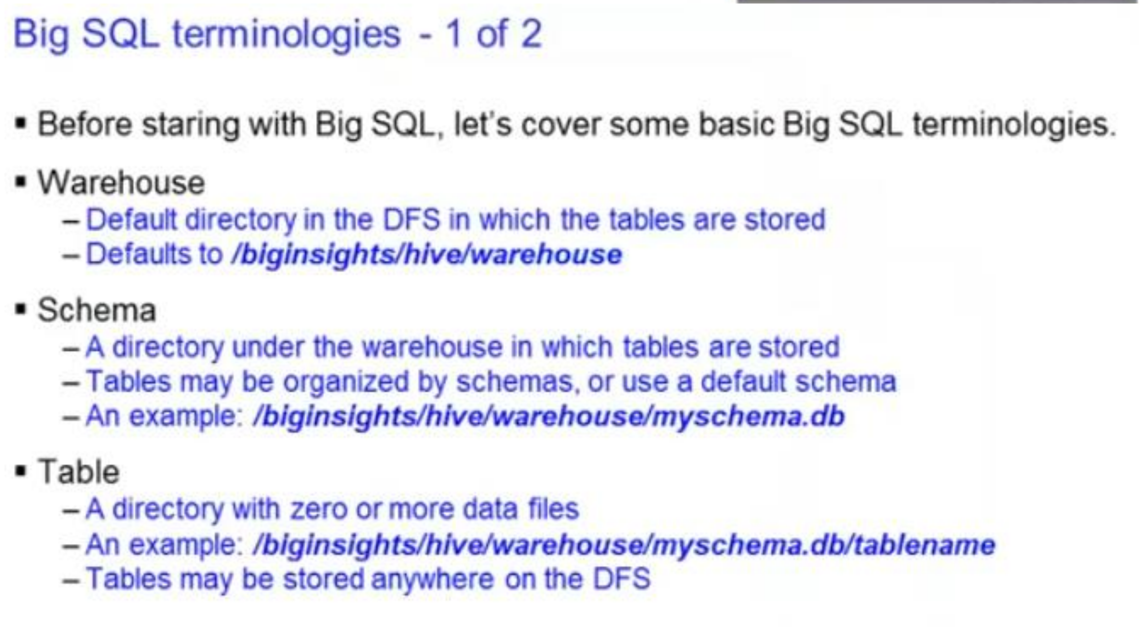
Advanced security/auditing
Resource and workload management
Self tuning memory management
Comprehensive monitoring



SQL capability highlights

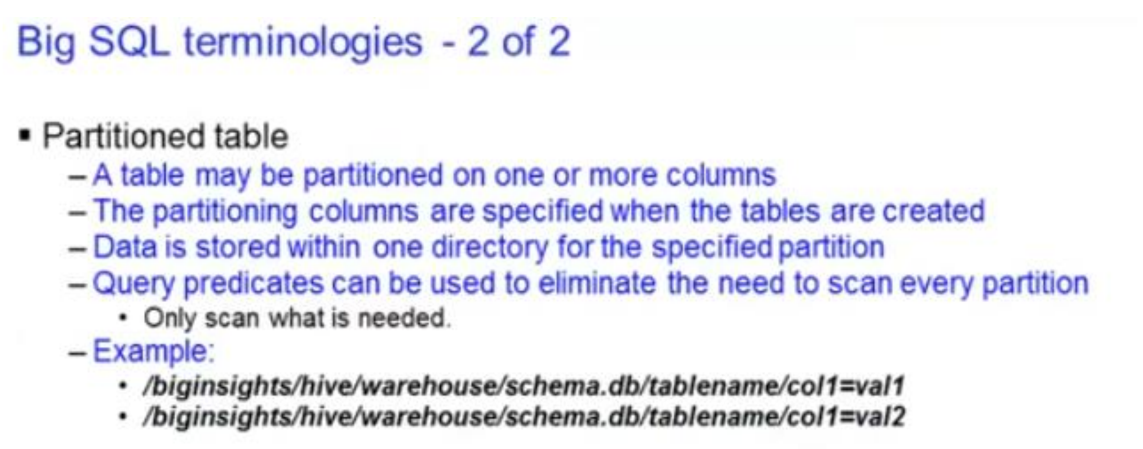
- Full support for subqueries
  - In `SELECT`, `FROM`, `WHERE` and `HAVING` clauses
  - Correlated and uncorrelated
  - Equality, non-equality subqueries
  - `EXISTS`, `NOT EXISTS`, `IN`, `ANY`, `SOME`, etc.
- All standard join operations
  - Standard and ANSI join syntax
  - Inner, outer, and full outer joins
  - Equality, non-equality, cross join support
  - Multi-value join (`WHERE (c1, c2) = …`)
  - `UNION`, `INTERSECT`, `EXCEPT`

```
SELECT
   s_name,
   count(*) AS numwait
FROM
   supplier,
   lineitem l1,
   orders,
   nation
WHERE
   s_suppkey = l1.l_suppkey
   AND o_orderkey = l1.l_orderkey
   AND o_orderstatus = 'F'
   AND l1.l_receiptdate > l1.l_commitdate
   AND EXISTS (
      SELECT
         *
      FROM
         lineitem l2
      WHERE
         l2.l_orderkey = l1.l_orderkey
         AND l2.l_suppkey <> l1.l_suppkey
   )
   AND NOT EXISTS (
      SELECT
         *
      FROM
         lineitem l3
      WHERE
         l3.l_orderkey = l1.l_orderkey
         AND l3.l_suppkey <> l1.l_suppkey
         AND l3.l_receiptdate >
               l3.l_commitdate
   )
   AND s_nationkey = n_nationkey
   AND n_name = ':1'
GROUP BY
   s_name
ORDER BY
   numwait desc,
   s_name;
```



Big SQL terminologies - 1 of 2

- Before staring with Big SQL, let's cover some basic Big SQL terminologies.
- Warehouse
  - Default directory in the DFS in which the tables are stored
  - Defaults to ***/biginsights/hive/warehouse***
- Schema
  - A directory under the warehouse in which tables are stored
  - Tables may be organized by schemas, or use a default schema
  - An example: ***/biginsights/hive/warehouse/myschema.db***
- Table
  - A directory with zero or more data files
  - An example: ***/biginsights/hive/warehouse/myschema.db/tablename***
  - Tables may be stored anywhere on the DFS



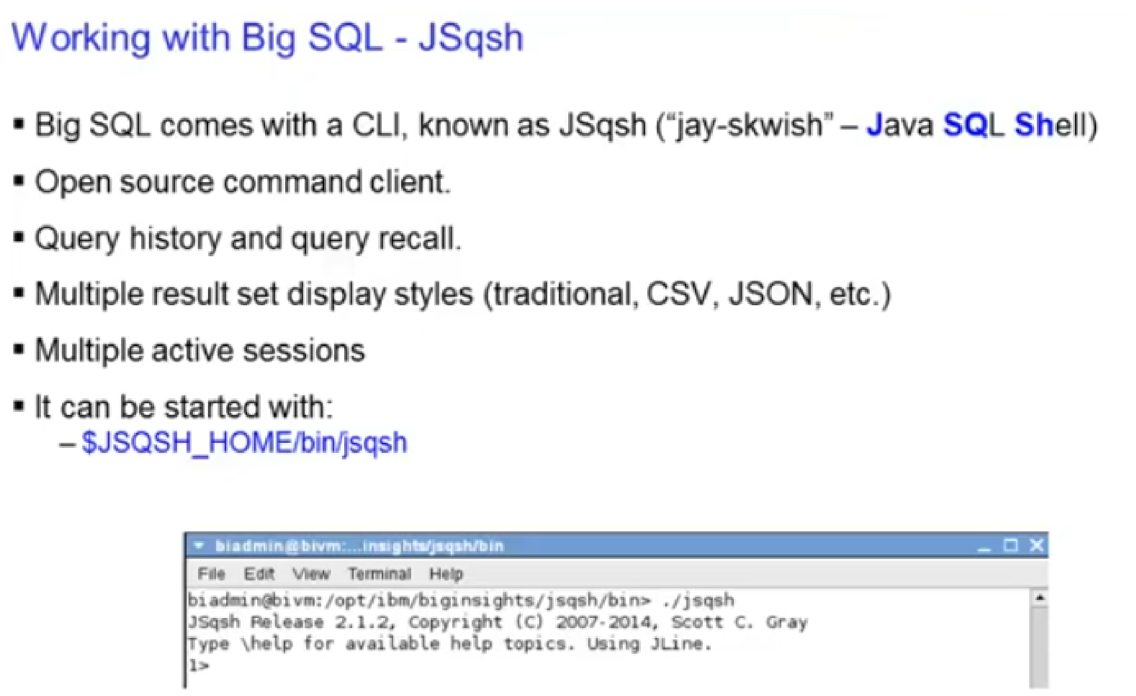
Big SQL terminologies - 2 of 2

- Partitioned table
  - A table may be partitioned on one or more columns
  - The partitioning columns are specified when the tables are created
  - Data is stored within one directory for the specified partition
  - Query predicates can be used to eliminate the need to scan every partition
    - Only scan what is needed.
  - Example:
    - ***/biginsights/hive/warehouse/schema.db/tablename/col1=val1***
    - ***/biginsights/hive/warehouse/schema.db/tablename/col1=val2***

You made have heard or read about there being
two versions of Big SQL at one point. In fact, there is only one release, and it is being
referred to as Big SQL. There may be cases where you see it being referred to as Big
SQL 3.0. It is not a separate product, so there is no actual version numbers associated.
Big SQL v1 refers to the previous version, but remember, there are no version numbers
associated. This is only to refer to the previous version. Big SQL is a complete redesign from
version 1. They share almost no code together. In fact, in order to use one or the other,
you must specify the correct driver. Existing clients may continue to use Big SQL v1, but
all new clients should be using Big SQL. The next slide will highlight some differences
between the two. It is important to understand again, that
Big SQL is not a standalone product, so it does not have a product version number. BigInsights
3.0 includes both Big SQL and Big SQL v1. The user can choose which version to use,
but by default, it is Big SQL. The most common reason to use Big SQL v1 would be for HBase
support.
Big SQL does not have support for HBase. In order to use Big SQL on HBase tables, you
must use Big SQL v1. The DDL in Big SQL is largely a superset of Big SQL v1. Remember,
Big SQL supports modern SQL:2011 capabilities, so it conforms to industry standards. There
may be some changes that you will need to make to your queries from Big SQL v1 in order
for it to run in Big SQL. Some of these differences are highlighted here.
Big SQL v1 treats single and double quotes as the same. Big SQL reserves double quotes
for identifiers. This adheres to SQL standard rules. In addition, the hadoop keyword is
required for creating tables in Big SQL. You can lift that restriction by setting the compatibility
mode. You can find more differences between these
two in the IBM Knowledge Center, with the biggest difference being the HBase support.
If you do not need HBase support, use the current Big SQL.
What does Hive have to do with Big SQL? Essentially,
Big SQL shares the Hive Metastore, for table definitions, location, storage format, and
encoding of input files. Big SQL maintains its own catalog in the master node. So as
long as the data is defined in the Hive Metastore and accessible in the Hadoop cluster, both
Big SQL and Hive can get to it. The DDL FMP, fenced mode process, essentially keeps
everything
in sync. When you create a table in Big SQL, it makes sure that table is created in the
Hive MetaStore and the Big SQL catalog. If you drop a table, the DDL FMP cleans it up
in both places. Now, this is only true if the statement is performed through Big SQL.
Unlike Big SQL 1.0, objects, created via Hive, are not immediately available. You must import
those table definitions from Hive.
Here are the features of Big SQL at a glance. Big SQL comes with comprehensive SQL support
and IBM SQL PL compatibility. I'll briefly go through this. Looking at

the top left with the Application Portability & Integration: Data is shared with the Hadoop
ecosystem. Remember that Big SQL only provides a logical view to the data stored in Hadoop.
There is a comprehensive list of file format support. For performance, Big SQL is designed
to move away from MapReduce by using an MPP runtime. There is a SQL query rewriter, cost
based optimizer, and the results are not constrained by memory. With federation your query
can
be written for multiple data sources such as DB2 LUW, DB2/z, Teradata, Oracle, Netezza.
Enterprise features include advanced security/auditing, resource and workload management,
self-tuning
memory management and comprehensive monitoring.
Here are some of the SQL capability highlights. If you are familiar with regular SQL queries
then this list might not seem that impressive. But, that just means that you are already
familiar with Big SQL. This visual highlights the capabilities that Hive or Impala doesn't
do, or doesn't do well. Big SQL has full support for subqueries. You
can create a subquery from the SELECT, FROM, WHERE and the HAVING clause. You can create
either correlated or uncorrelated subqueries. You can do equality or non-equality subqueries.
You can use EXISTS, NOT EXISTS, IN, ANY, SOME. All the standard join operations are
supported.
You have your inner, outer, and full outer joins. You have all the standard operators.
You have the equality, non-equality and cross join support. You have your multi-value join.
The sample code shows a valid Big SQL query.
Here are some Big SQL terminologies. Warehouse is the default directory in the
DFS in which the tables are stored. The default location is /biginsights/hive/warehouse
Schema is the directory under the warehouse directory in which the tables are stored.
The tables may be organized by schemas, or it can use a default schema. An example would
be /biginsights/hive/warehouse/myschema.db Table is a directory with zero or more data
files. An example of a table directory within DFS is
/biginsights/hive/warehouse/myschema.db/tablename.
The tables may be stored anywhere on the DFS. This visual here is just showing an example
of a location.
A table may be partitioned on one or more columns. You specify the partitioning columns
when you create the table. When a table is partitioned, the data is stored within the
directory for the specified partition. Query predicates can then be used to eliminate the
need to scan every partition and only scan what is needed - speeding up the query.
For example, if your table was partitioned by col1, there will be two directories on
the DFS. The name of the two directories are, in fact,
"col1=val1" and "col1=val2" for each respective partitions.

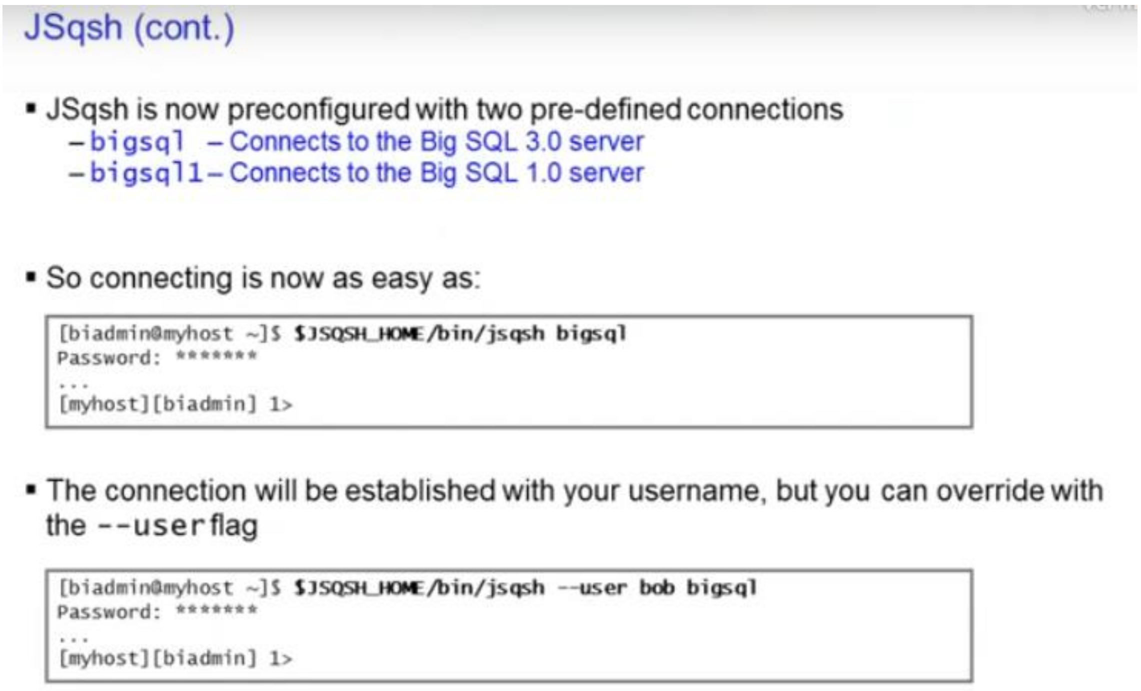
MODULE 2 – BIG SQL DATA TYPES



Working with Big SQL - JSqsh

- Big SQL comes with a CLI, known as JSqsh ("jay-skwish" – **J**ava **SQ**L **Sh**ell)
- Open source command client.
- Query history and query recall.
- Multiple result set display styles (traditional, CSV, JSON, etc.)
- Multiple active sessions
- It can be started with:
  – $JSQSH_HOME/bin/jsqsh

```
biadmin@bivm:/opt/ibm/biginsights/jsqsh/bin> ./jsqsh
JSqsh Release 2.1.2, Copyright (C) 2007-2014, Scott C. Gray
Type \help for available help topics. Using JLine.
1>
```



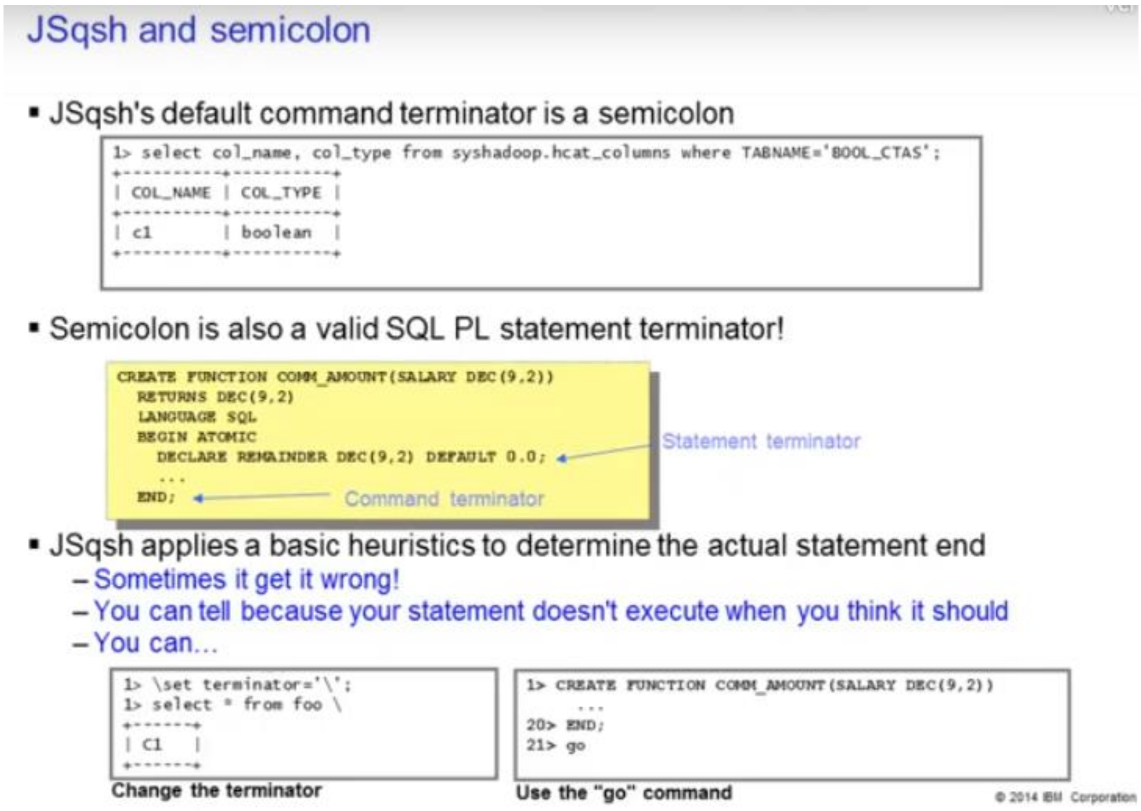
JSqsh (cont.)

- JSqsh is now preconfigured with two pre-defined connections
  – `bigsql` – Connects to the Big SQL 3.0 server
  – `bigsql1` – Connects to the Big SQL 1.0 server
- So connecting is now as easy as:

```
[biadmin@myhost ~]$ $JSQSH_HOME/bin/jsqsh bigsql
Password: *******
...
[myhost][biadmin] 1>
```

- The connection will be established with your username, but you can override with the `--user` flag

```
[biadmin@myhost ~]$ $JSQSH_HOME/bin/jsqsh --user bob bigsql
Password: *******
...
[myhost][biadmin] 1>
```



JSqsh and semicolon

- JSqsh's default command terminator is a semicolon

```
1> select col_name, col_type from syshadoop.hcat_columns where TABNAME='BOOL_CTAS';
+----------+----------+
| COL_NAME | COL_TYPE |
+----------+----------+
| c1       | boolean  |
+----------+----------+
```

- Semicolon is also a valid SQL PL statement terminator!

```
CREATE FUNCTION COMM_AMOUNT(SALARY DEC(9,2))
  RETURNS DEC(9,2)
  LANGUAGE SQL
  BEGIN ATOMIC
    DECLARE REMAINDER DEC(9,2) DEFAULT 0.0;
    ...
  END;
```

Statement terminator

Command terminator

- JSqsh applies a basic heuristics to determine the actual statement end
  - Sometimes it get it wrong!
  - You can tell because your statement doesn't execute when you think it should
  - You can…

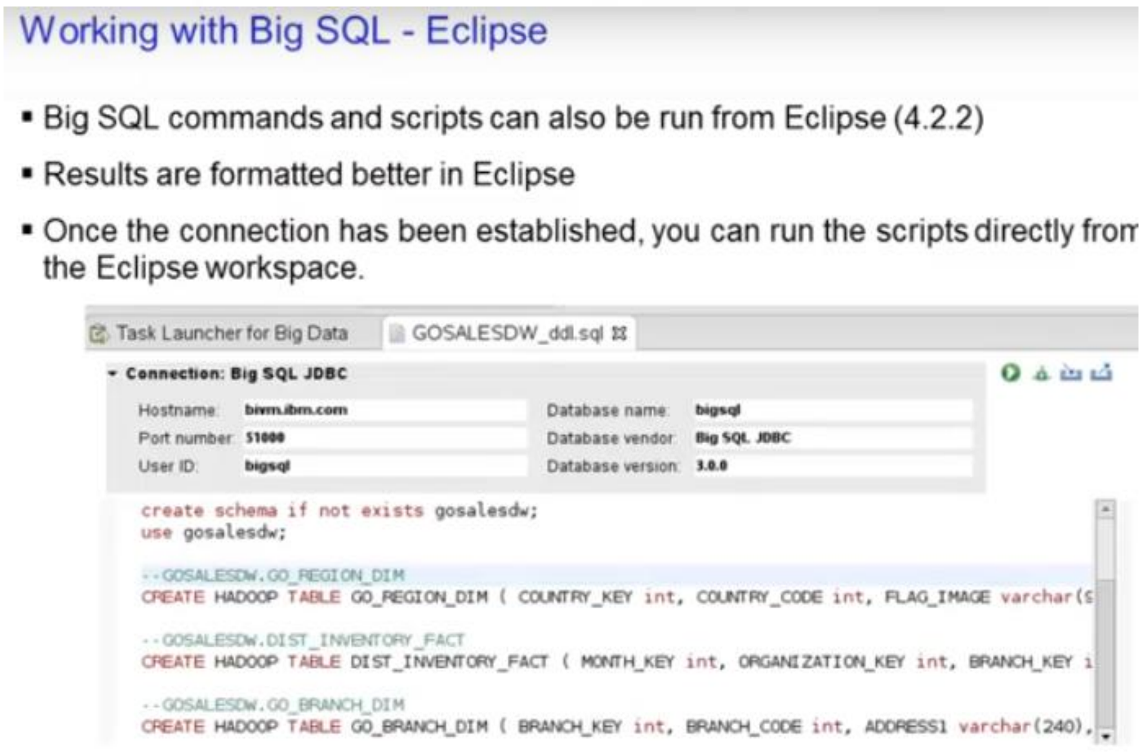
```
1> \set terminator='\';
1> select * from foo \
+------+
| C1   |
+------+
```

Change the terminator

```
1> CREATE FUNCTION COMM_AMOUNT(SALARY DEC(9,2))
   ...
20> END;
21> go
```

Use the "go" command

© 2014 IBM Corporation



Working with Big SQL - Eclipse

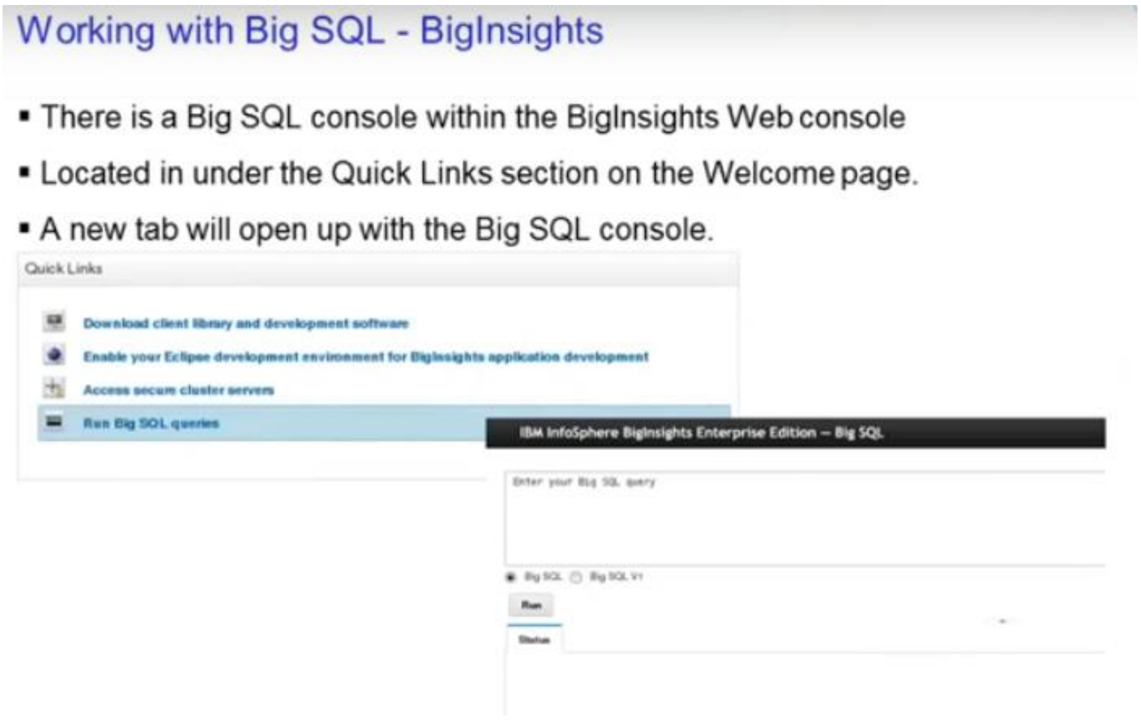
- Big SQL commands and scripts can also be run from Eclipse (4.2.2)
- Results are formatted better in Eclipse
- Once the connection has been established, you can run the scripts directly from the Eclipse workspace.

Task Launcher for Big Data | GOSALESDW_ddl.sql

Connection: Big SQL JDBC

Hostname:	bivm.ibm.com	Database name:	bigsql
Port number:	51000	Database vendor:	Big SQL JDBC
User ID:	bigsql	Database version:	3.0.0

```
create schema if not exists gosalesdw;
use gosalesdw;

--GOSALESDW.GO_REGION_DIM
CREATE HADOOP TABLE GO_REGION_DIM ( COUNTRY_KEY int, COUNTRY_CODE int, FLAG_IMAGE varchar(9

--GOSALESDW.DIST_INVENTORY_FACT
CREATE HADOOP TABLE DIST_INVENTORY_FACT ( MONTH_KEY int, ORGANIZATION_KEY int, BRANCH_KEY i

--GOSALESDW.GO_BRANCH_DIM
CREATE HADOOP TABLE GO_BRANCH_DIM ( BRANCH_KEY int, BRANCH_CODE int, ADDRESS1 varchar(240),
```

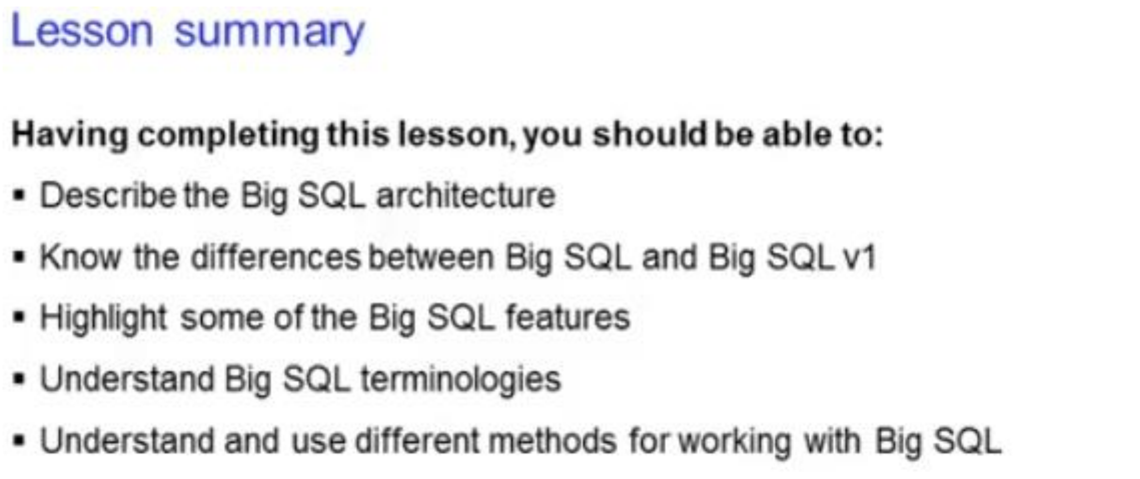


Working with Big SQL - BigInsights

- There is a Big SQL console within the BigInsights Web console
- Located in under the Quick Links section on the Welcome page.
- A new tab will open up with the Big SQL console.

Quick Links
Download client library and development software
Enable your Eclipse development environment for BigInsights application development
Access secure cluster servers
Run Big SQL queries
IBM InfoSphere BigInsights Enterprise Edition – Big SQL
Enter your Big SQL query
Big SQL
Big SQL V1
Run
Status

the next lesson in this course.



Lesson summary

Having completing this lesson, you should be able to:

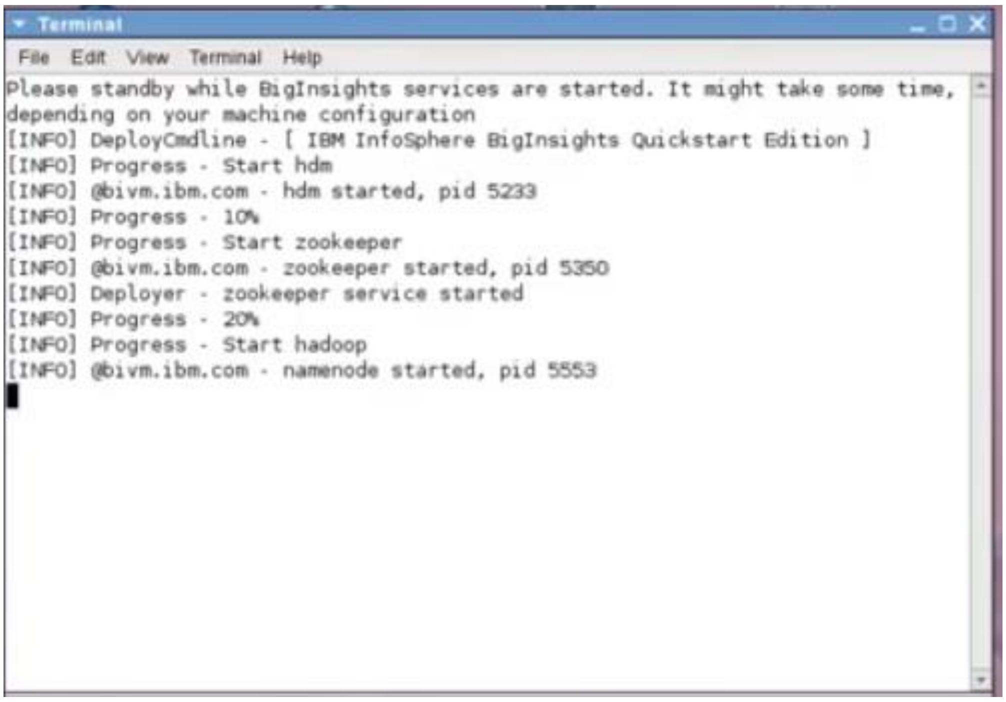
- Describe the Big SQL architecture
- Know the differences between Big SQL and Big SQL v1
- Highlight some of the Big SQL features
- Understand Big SQL terminologies
- Understand and use different methods for working with Big SQL

Big SQL has a CLI known as JSqsh. JSqsh is
an open source client that works with any JDBC driver, not just Big SQL. JSqsh has the capability to do query history and query recall. It displays results in various styles depending on the file type such as traditional, CSV, JSON, etc.). You can also have multiple active sessions. The CLI can be started with $JSQSH_HOME/bin/jsqsh. You will get to use this in the lab exercise.

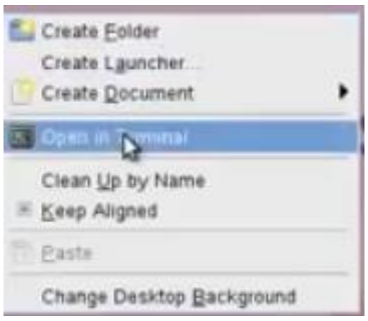
There are two pre-defined connections to which
you can connect using jsqsh. The bigsql connection connects to the Big SQL 3.0 server. The bigsql1
connects to the Big SQL 1.0 server. So connecting to either of the connection is a simple as providing the connection name after the jsqsh command. By default, you will connect with your OS username, but you can override that using the --user flag.

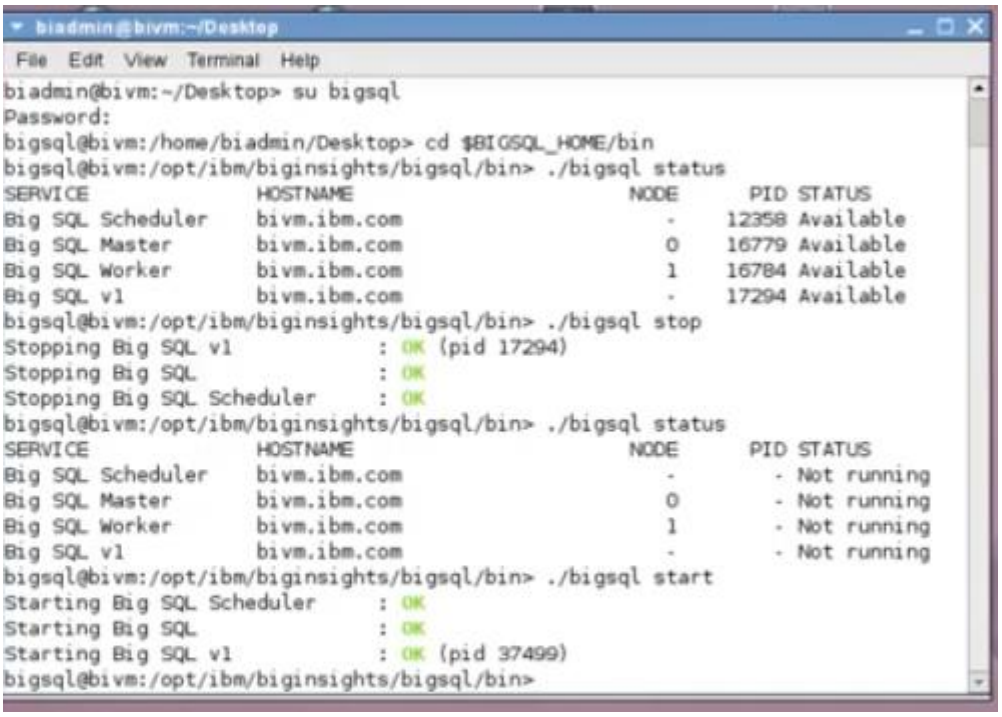
If you have used JSqsh before, you will know

that the default command terminator is a semicolon. With the new Big SQL, the current version,
the semicolon is also a valid SQL PL statement terminator, so while JSqsh can take a best guess to determine the actual statement, it can sometimes get it wrong. When that happens,
the statement will not run until you explicitly execute the "go" command. The semicolon in Big SQL is actually the go command underneath. Alternatively, you can change the default terminator from a semicolon to another terminator such as the backslash.
Another way to work with Big SQL is through
Eclipse. Most users actually prefer Eclipse because it provides the results in a better format. It is easier to read. Once you have set up the Big SQL connection within Eclipse, you can run Big SQL queries directly from there and get the results back in the Eclipse console.
Finally, a third way you can interact with
Big SQL is through the BigInsights web console. Once you log in to the BigInsights console on the Welcome page, click the Run Big SQL queries under the Quick Links section. This will open up a new window or tab with the Big SQL console. You can run your queries directly from there.
Having completed this lesson, you should now
be able describe the Big SQL architecture. Know the differences between Big SQL and Big SQL v1. Highlight some of the Big SQL features. Understand Big SQL terminologies. Understand
and use different methods for working with Big SQL.
You have completed this lesson. Proceed to



```
Please standby while BigInsights services are started. It might take some time,
depending on your machine configuration
[INFO] DeployCmdline - [ IBM InfoSphere BigInsights Quickstart Edition ]
[INFO] Progress - Start hdm
[INFO] @bivm.ibm.com - hdm started, pid 5233
[INFO] Progress - 10%
[INFO] Progress - Start zookeeper
[INFO] @bivm.ibm.com - zookeeper started, pid 5350
[INFO] Deployer - zookeeper service started
[INFO] Progress - 20%
[INFO] Progress - Start hadoop
[INFO] @bivm.ibm.com - namenode started, pid 5553
```



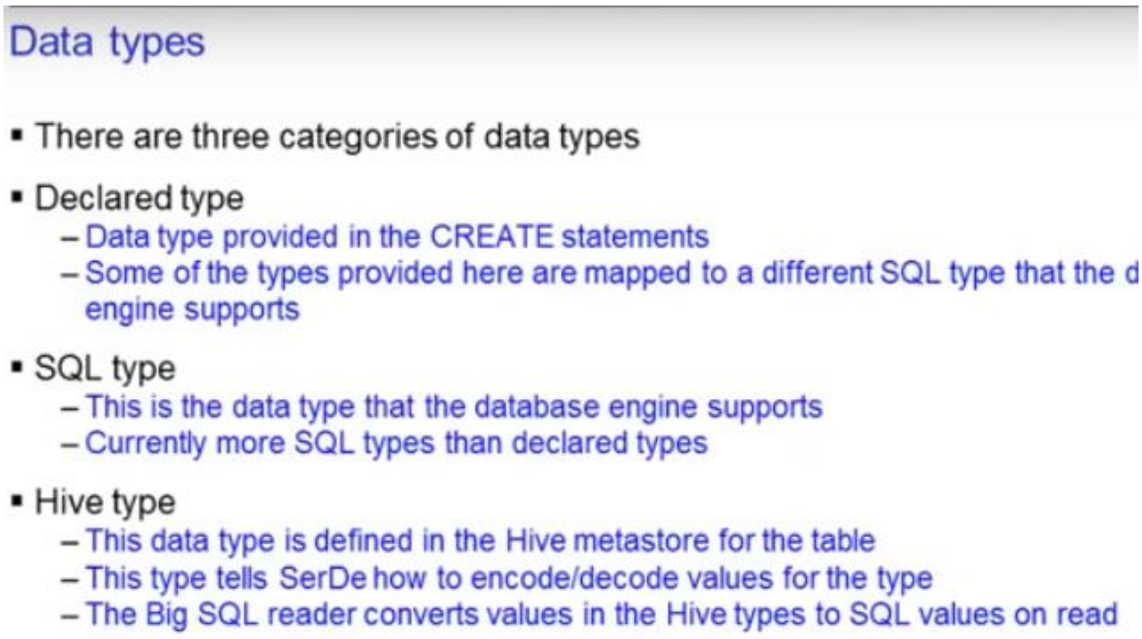


```
biadmin@bivm:~/Desktop> su bigsql
Password:
bigsql@bivm:/home/biadmin/Desktop> cd $BIGSQL_HOME/bin
bigsql@bivm:/opt/ibm/biginsights/bigsql/bin> ./bigsql status
SERVICE              HOSTNAME                      NODE     PID STATUS
Big SQL Scheduler    bivm.ibm.com                   -     12358 Available
Big SQL Master       bivm.ibm.com                   0     16779 Available
Big SQL Worker       bivm.ibm.com                   1     16784 Available
Big SQL v1           bivm.ibm.com                   -     17294 Available
bigsql@bivm:/opt/ibm/biginsights/bigsql/bin> ./bigsql stop
Stopping Big SQL v1            : OK (pid 17294)
Stopping Big SQL               : OK
Stopping Big SQL Scheduler     : OK
bigsql@bivm:/opt/ibm/biginsights/bigsql/bin> ./bigsql status
SERVICE              HOSTNAME                      NODE     PID STATUS
Big SQL Scheduler    bivm.ibm.com                   -         - Not running
Big SQL Master       bivm.ibm.com                   0         - Not running
Big SQL Worker       bivm.ibm.com                   1         - Not running
Big SQL v1           bivm.ibm.com                   -         - Not running
bigsql@bivm:/opt/ibm/biginsights/bigsql/bin> ./bigsql start
Starting Big SQL Scheduler     : OK
Starting Big SQL               : OK
Starting Big SQL v1            : OK (pid 37499)
bigsql@bivm:/opt/ibm/biginsights/bigsql/bin>
```

In this lab exercise, you will be connecting to the IBM Big SQL Server and running SQL queries. IBM Big SQL is a component of The IBM InfoSphere BigInsights product. Before you can start working with your Hadoop data using Big SQL, you need to be able to connect to the Big SQL server. There are three different methods for doing so and you will get a chance to work with all three in this lab exercise. The first of the three methods that you will use is JSqsh. JSqsh is an open source CLI for JDBC applications such as Big SQL. That means that you can also use JSqsh for other JDBC applications. The second method is to use Eclipse. Users generally prefer using Eclipse because the results are formatted in a way that makes it easier for the user to understand. The third method is the BigInsights web console. It is a web based tool where you can run your queries and see your results.
You will need to first start up BigInsights. There are two ways you can do this. Using commands in the terminal or the shortcut icon on the desktop. The lab exercise guide shows you how to use the commands. I will use the desktop shortcut. I'll speed up the recording through the startup phase.
Now that BigInsights has started, open up a new terminal. I am going to show you another way you can manage the Big SQL service. First,
switch to the bigsql user. The password is bigsql.
Then, change directory to the $BIGSQL_HOME/bin.
From within that directory execute bigsql status to get the status of the Big SQL service.
You'll notice here that there is a Big SQL v1 instance. This current release of BigInsights comes with Big SQL and Big SQL v1, which is the older, legacy version. You would only

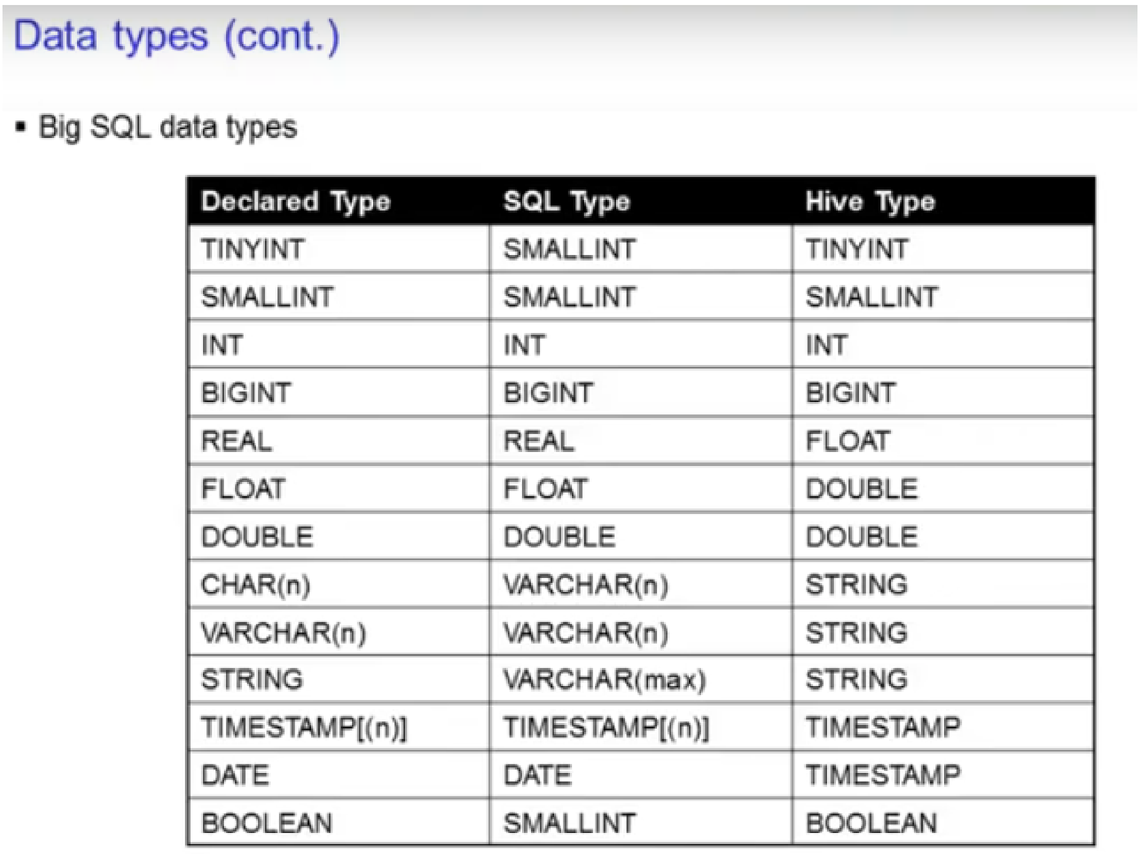
really use v1 if you need support for HBase or if your applications requires specific
v1 features that is not yet supported in the current release of Big SQL.
To stop the Big SQL instance, type in bigsql stop.
Check the status again to see that the services have stopped.
Go ahead and restart Big SQL using bigsql start.
Close the terminal. We are done with this section. Proceed to the next video to continue
on with the lab exercise.

MODULE 2 –



Data types

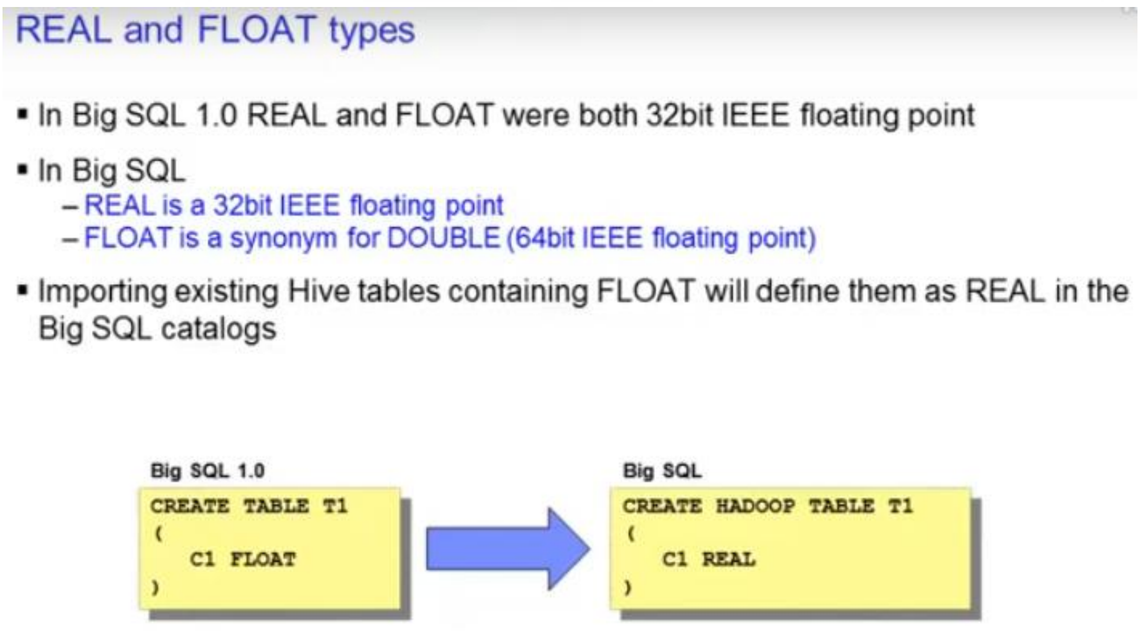
- There are three categories of data types
- Declared type
  - Data type provided in the CREATE statements
  - Some of the types provided here are mapped to a different SQL type that the d engine supports
- SQL type
  - This is the data type that the database engine supports
  - Currently more SQL types than declared types
- Hive type
  - This data type is defined in the Hive metastore for the table
  - This type tells SerDe how to encode/decode values for the type
  - The Big SQL reader converts values in the Hive types to SQL values on read



Data types (cont.)

- Big SQL data types

Declared Type	SQL Type	Hive Type
TINYINT	SMALLINT	TINYINT
SMALLINT	SMALLINT	SMALLINT
INT	INT	INT
BIGINT	BIGINT	BIGINT
REAL	REAL	FLOAT
FLOAT	FLOAT	DOUBLE
DOUBLE	DOUBLE	DOUBLE
CHAR(n)	VARCHAR(n)	STRING
VARCHAR(n)	VARCHAR(n)	STRING
STRING	VARCHAR(max)	STRING
TIMESTAMP[(n)]	TIMESTAMP[(n)]	TIMESTAMP
DATE	DATE	TIMESTAMP
BOOLEAN	SMALLINT	BOOLEAN



REAL and FLOAT types

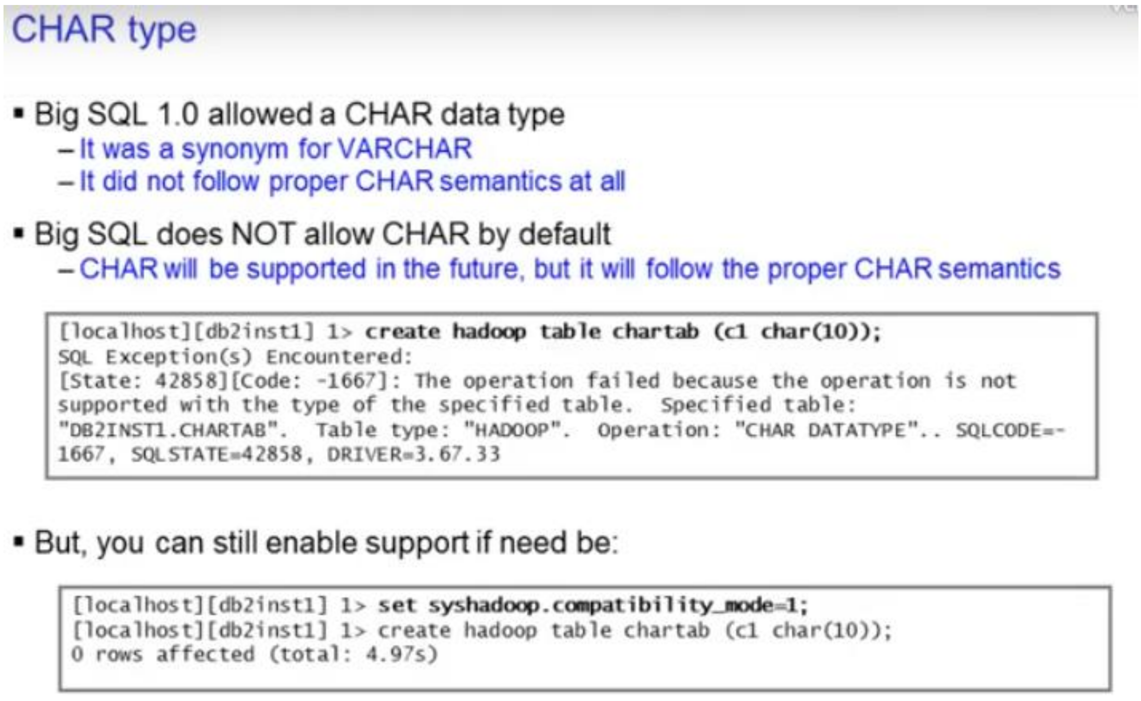
- In Big SQL 1.0 REAL and FLOAT were both 32bit IEEE floating point
- In Big SQL
  - REAL is a 32bit IEEE floating point
  - FLOAT is a synonym for DOUBLE (64bit IEEE floating point)
- Importing existing Hive tables containing FLOAT will define them as REAL in the Big SQL catalogs

Big SQL 1.0

```
CREATE TABLE T1
(
   C1 FLOAT
)
```

Big SQL

```
CREATE HADOOP TABLE T1
(
   C1 REAL
)
```



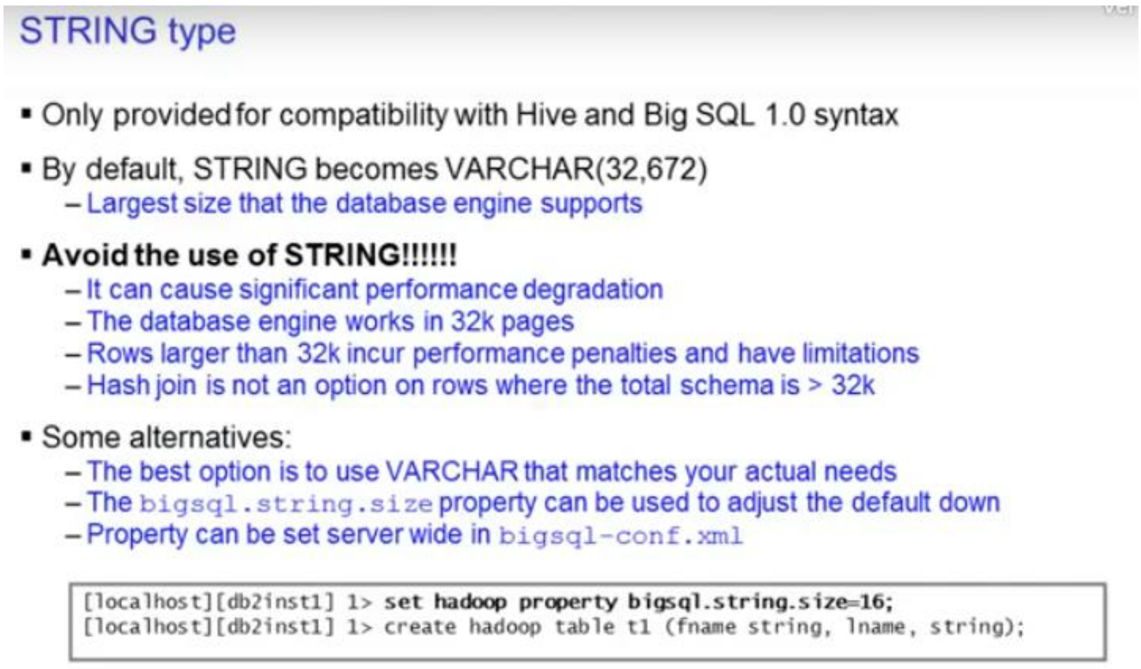
CHAR type

- Big SQL 1.0 allowed a CHAR data type
  - It was a synonym for VARCHAR
  - It did not follow proper CHAR semantics at all
- Big SQL does NOT allow CHAR by default
  - CHAR will be supported in the future, but it will follow the proper CHAR semantics

```
[localhost][db2inst1] 1> create hadoop table chartab (c1 char(10));
SQL Exception(s) Encountered:
[State: 42858][Code: -1667]: The operation failed because the operation is not
supported with the type of the specified table.  Specified table:
"DB2INST1.CHARTAB".  Table type: "HADOOP".  Operation: "CHAR DATATYPE".. SQLCODE=-
1667, SQLSTATE=42858, DRIVER=3.67.33
```

- But, you can still enable support if need be:

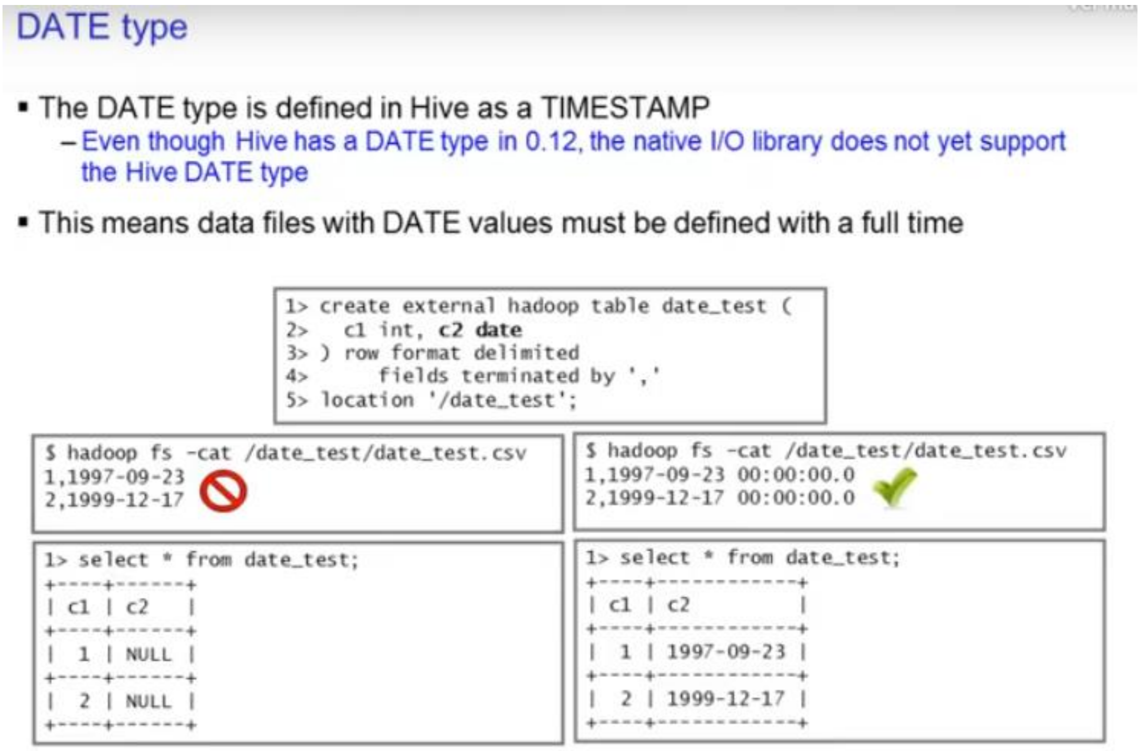
```
[localhost][db2inst1] 1> set syshadoop.compatibility_mode=1;
[localhost][db2inst1] 1> create hadoop table chartab (c1 char(10));
0 rows affected (total: 4.97s)
```



STRING type

- Only provided for compatibility with Hive and Big SQL 1.0 syntax
- By default, STRING becomes VARCHAR(32,672)
  - Largest size that the database engine supports
- **Avoid the use of STRING!!!!!!**
  - It can cause significant performance degradation
  - The database engine works in 32k pages
  - Rows larger than 32k incur performance penalties and have limitations
  - Hash join is not an option on rows where the total schema is > 32k
- Some alternatives:
  - The best option is to use VARCHAR that matches your actual needs
  - The `bigsql.string.size` property can be used to adjust the default down
  - Property can be set server wide in `bigsql-conf.xml`

```
[localhost][db2inst1] 1> set hadoop property bigsql.string.size=16;
[localhost][db2inst1] 1> create hadoop table t1 (fname string, lname, string);
```



DATE type

- The DATE type is defined in Hive as a TIMESTAMP
  - Even though Hive has a DATE type in 0.12, the native I/O library does not yet support the Hive DATE type
- This means data files with DATE values must be defined with a full time

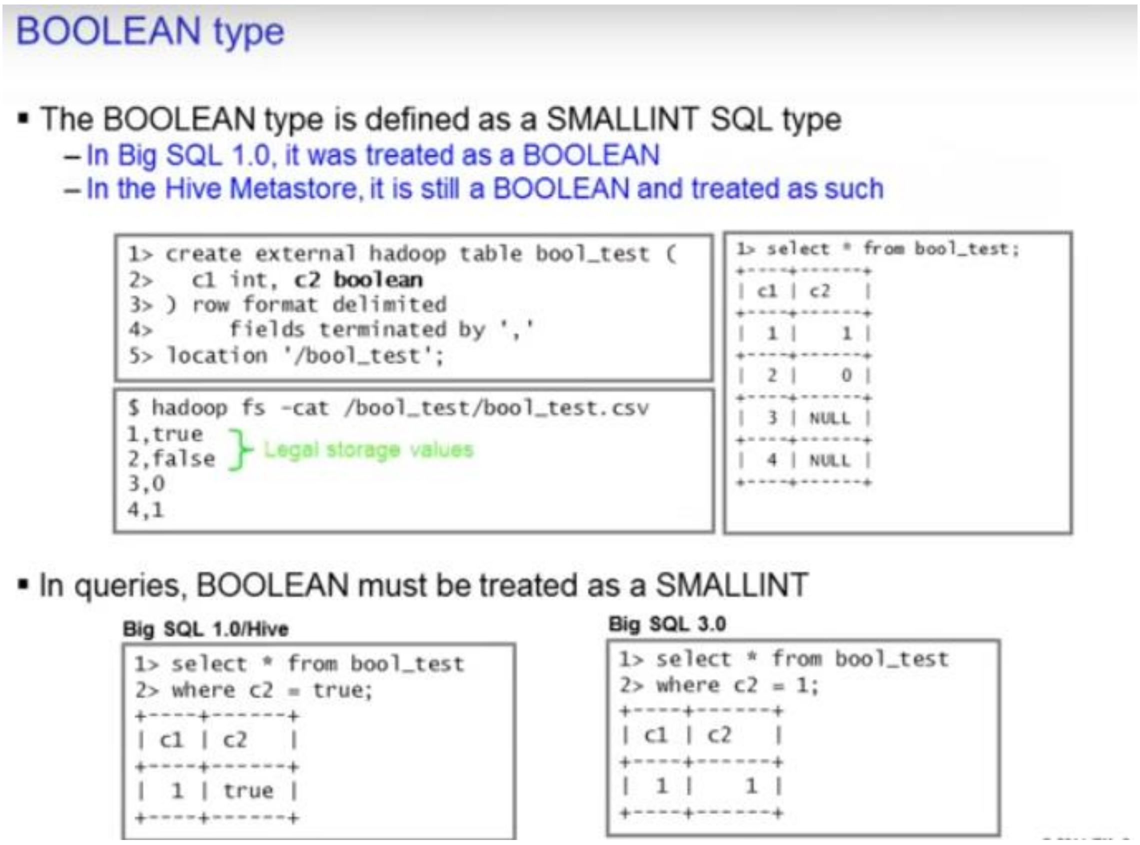
```
1> create external hadoop table date_test (
2>    c1 int, c2 date
3> ) row format delimited
4>       fields terminated by ','
5> location '/date_test';
```

```
$ hadoop fs -cat /date_test/date_test.csv
1,1997-09-23
2,1999-12-17
```

```
$ hadoop fs -cat /date_test/date_test.csv
1,1997-09-23 00:00:00.0
2,1999-12-17 00:00:00.0
```

```
1> select * from date_test;
+----+------+
| c1 | c2   |
+----+------+
|  1 | NULL |
+----+------+
|  2 | NULL |
+----+------+
```

```
1> select * from date_test;
+----+------------+
| c1 | c2         |
+----+------------+
|  1 | 1997-09-23 |
+----+------------+
|  2 | 1999-12-17 |
+----+------------+
```



BOOLEAN type

- The BOOLEAN type is defined as a SMALLINT SQL type
  - In Big SQL 1.0, it was treated as a BOOLEAN
  - In the Hive Metastore, it is still a BOOLEAN and treated as such

```
1> create external hadoop table bool_test (
2>    c1 int, c2 boolean
3> ) row format delimited
4>       fields terminated by ','
5> location '/bool_test';
```

```
$ hadoop fs -cat /bool_test/bool_test.csv
1,true
2,false } Legal storage values
3,0
4,1
```

```
1> select * from bool_test;
+----+------+
| c1 | c2   |
+----+------+
|  1 |    1 |
+----+------+
|  2 |    0 |
+----+------+
|  3 | NULL |
+----+------+
|  4 | NULL |
+----+------+
```

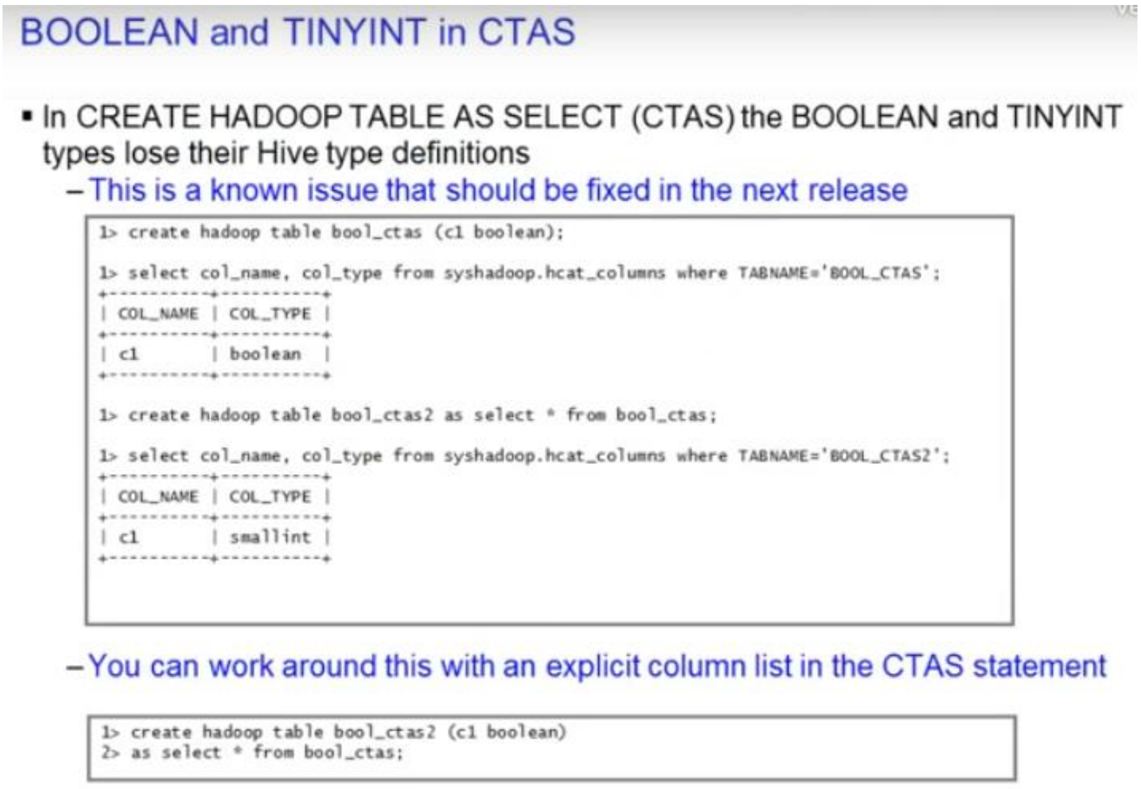
- In queries, BOOLEAN must be treated as a SMALLINT

Big SQL 1.0/Hive

```
1> select * from bool_test
2> where c2 = true;
+----+------+
| c1 | c2   |
+----+------+
|  1 | true |
+----+------+
```

Big SQL 3.0

```
1> select * from bool_test
2> where c2 = 1;
+----+------+
| c1 | c2   |
+----+------+
|  1 |    1 |
+----+------+
```



BOOLEAN and TINYINT in CTAS

- In CREATE HADOOP TABLE AS SELECT (CTAS) the BOOLEAN and TINYINT types lose their Hive type definitions
  - This is a known issue that should be fixed in the next release

```
1> create hadoop table bool_ctas (c1 boolean);

1> select col_name, col_type from syshadoop.hcat_columns where TABNAME='BOOL_CTAS';
+----------+----------+
| COL_NAME | COL_TYPE |
+----------+----------+
| c1       | boolean  |
+----------+----------+

1> create hadoop table bool_ctas2 as select * from bool_ctas;

1> select col_name, col_type from syshadoop.hcat_columns where TABNAME='BOOL_CTAS2';
+----------+----------+
| COL_NAME | COL_TYPE |
+----------+----------+
| c1       | smallint |
+----------+----------+
```

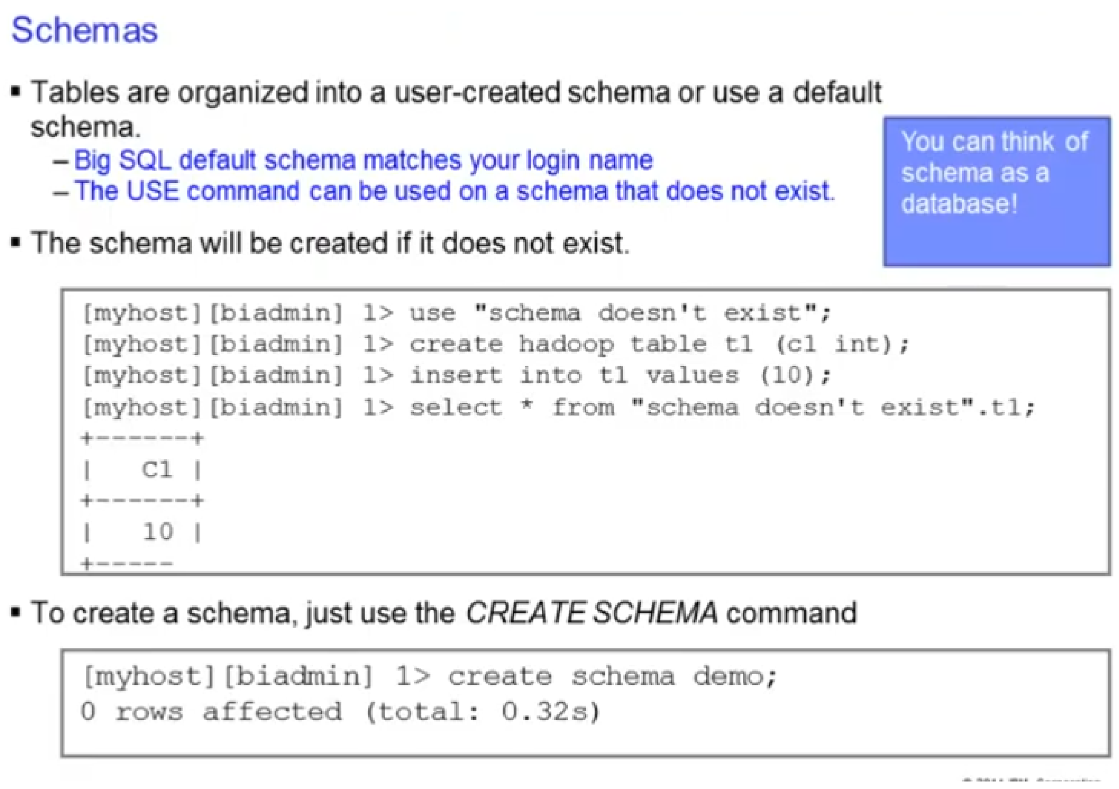
  - You can work around this with an explicit column list in the CTAS statement

```
1> create hadoop table bool_ctas2 (c1 boolean)
2> as select * from bool_ctas;
```

Welcome to the course, Big SQL on Hadoop.
In this lesson, Big SQL data types, you will learn about the different Big SQL data types.
After completing this lesson, you will be
able to list and explain the Big SQL data types. Create Big SQL schemas. Create Big
SQL tables. Understand and use the file formats supported by Big SQL.
There are three categories of data types around
Big SQL that you will see here. Declared type: data types that are provided
in the CREATE statements. These particular types only exists in the CREATE statement
and nowhere else. Some of the types provided here are mapped to a different SQL type that
the engine supports. For example, you declare with one type, but in order for the engine
to use that type, it is mapped to a different type, which is supported.
SQL types: data types that the database engine supports. There are currently more SQL types
than declared types. That just means that there are some types that you cannot use in
the CREATE statement, but you can use otherwise, such as the DECIMAL type.
Hive type: data types that the Hive catalog understands. When you create a table, Big
SQL has to tell the Hive catalog what that column is. It basically tells the Hive SerDe
how to interpret the value. The Big SQL reader converts the values in the Hive types to SQL
values on read.
Here is a table that shows the relationship between the three data types categories. As
a quick refresher, the declared type is the type provided to the CREATE statement. Some
of those types are mapped to the SQL type that the database engine understands. When
a table is created, the data types are converted to Hive types in order for the metadata to
be stored in the Hive catalog. For example, when you create a table, Hive
has a type called a STRING. The Big SQL runtime does not have this notion of a STRING. It
cannot support arbitrary string, so when it is defined in the catalog, it is defined as
varchar. In other words, when you create a table, and specify a string column, the Big
SQL engine actually creates that column as a varchar of 32KB.

The types on this chart are just a subset of all the types. There are many more SQL
types than there are declared types -- meaning there are some that you cannot use in the
CREATE statement. I will go over a few more of the types on the following slides.
Remember, Big SQL 1.0 just refers to the previous
version of Big SQL. There is no version number associated with Big SQL, it is not a separate
product. In Big SQL 1.0, REAL and FLOAT were both 32bit IEEE floating point. In Big SQL,
REAL is a 32bit IEEE floating point. FLOAT is essentially a DOUBLE, a 64bit IEEE floating
point. This means that if you have to have the same data type in Big SQL, you should
change from FLOAT to REAL. If you have a Hive CREATE table statement
that contains FLOAT, you should change it to REAL if you want it to continue to have
32bit value. Otherwise, it will be created as a DOUBLE, a 64bit value.
In Big SQL 1.0, the CHAR type did not follow
the proper CHAR semantic at all. It was basically a synonym for VARCHAR. Because of that
reason,
the current Big SQL does not allow CHAR, by default.
If you try to create table with a CHAR column, you will get an error. CHAR will be supported
in the future using the proper CHAR semantics. However, if you happen to have a lot of
CREATE
table statements with columns defined as CHAR and you desperately need to use it, there
is an option to enable support. Just set the compatibility_mode=1 and you can run the
statements
without any errors.
Now we get to the STRING type. As I mentioned, Big SQL does not have this notion of a
STRING,
so when it sees a STRING type in the CREATE command, it converts it into a VARCHAR(32,672),
which is the largest size the database engine support. This becomes increasingly inefficient
and can cause performance degradation. The database engine works in 32k page so rows
larger than 32k incur performance penalties. For example, hash join is not possible on
rows where total schema is > 32k. Here are some alternatives you can use instead
of using the STRING type. The best option is to use VARCHAR with the size that you need.
If you must use STRING, you could adjust the default size down to what is appropriate for
your needs. There is also a server wide setting in the bigsql-conf.xml file.
Moving on to the DATE type. In Hive, the DATE
type is mapped to a TIMESTAMP. This means that when you define a DATE column, the values
must be defined with a full timestamp. Hive does have a DATE type, but the native I/O
library does not yet support that. Here is an example. In the top box, you can
see the CREATE command to create an external Hadoop table with a DATE column. On the left
side, you can see that you have a file with a list of dates only, without a time. If you
query it, you get NULLs back. The right way to do this is to append the time value to
the date as shown on the right side. When you query that, you get the appropriate date
values back.
The time value that you provide must be in
the format that you see here: 00:00:00.0[0...]. The LOAD HADOOP command automatically
takes
care of converting the files with DATE to TIMESTAMP format so you do not have to worry
about that if you use the LOAD command. For everything else, you will need to make
sure you include the time value.
The database runtime engine does not fully
support the concept of a BOOLEAN. The BOOLEAN type is defined as a SMALLINT SQL type in
Big SQL. In Big SQL 1.0, it was defined as a BOOLEAN
type. In the Hive metastore, it is also a BOOLEAN. But in Big SQL queries, it is treated
as a SMALLINT. For example:

You create a table with a BOOLEAN column. The only valid values for those particular columns are true and false. 0's and 1's are not valid BOOLEAN values. The example table have both true/false and 0/1. So when you select the table, you see the values of 1 and 0 for true and false. But for the 0 and 1 value, the database engine does not recognize it (since it isn't a valid BOOLEAN), so it returns NULL.
Remember, the runtime engine uses SMALLINT so true is mapped to the value of 1 and false is 0. In queries, you must take care to change any
BOOLEAN to SMALLINT. For example, if you are doing a comparison, you must change from true
to 1 (false to 0) for the queries to work properly.
In CREATE HADOOP TABLE as SELECT (CTAS), the
BOOLEAN and TINYINT type loses their Hive type definitions. The declared types BOOLEAN and TINYINT are both mapped to SMALLINT in the SQL runtime engine.
The example shows the first table being created with a single BOOLEAN column.
Then it creates a second table using the CTAS statement on the first table.
The second table loses the BOOLEAN definition and becomes TINYINT.
A way to circumvent that is to use an explicit column list in the CTAS statement as shown in the bottom screenshot. This allows it to keep the BOOLEAN type.



Schemas

- Tables are organized into a user-created schema or use a default schema.
  - Big SQL default schema matches your login name
  - The USE command can be used on a schema that does not exist.
- The schema will be created if it does not exist.

You can think of schema as a database!

```
[myhost][biadmin] 1> use "schema doesn't exist";
[myhost][biadmin] 1> create hadoop table t1 (c1 int);
[myhost][biadmin] 1> insert into t1 values (10);
[myhost][biadmin] 1> select * from "schema doesn't exist".t1;
+-------+
|   C1  |
+-------+
|   10  |
+-----
```

- To create a schema, just use the *CREATE SCHEMA* command

```
[myhost][biadmin] 1> create schema demo;
0 rows affected (total: 0.32s)
```

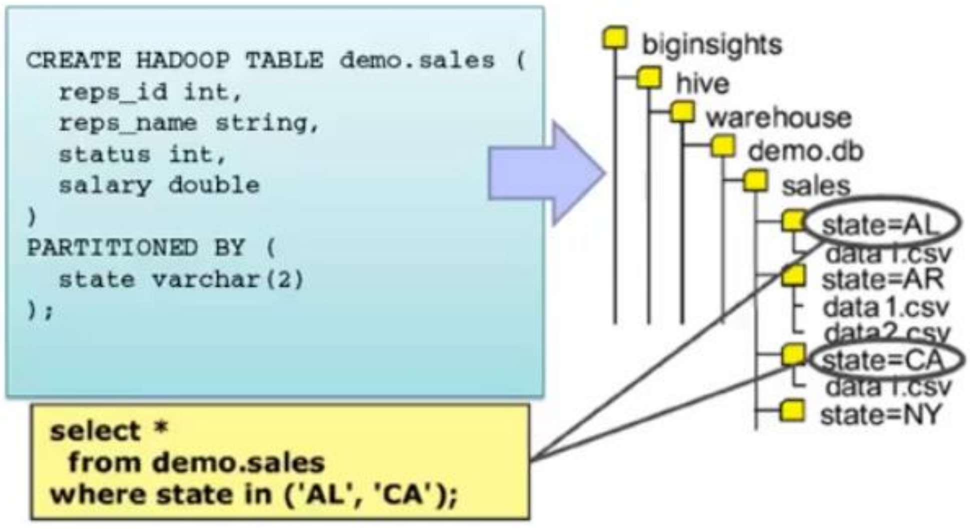
CREATE TABLE

- HADOOP keyword
  - Must be specified unless you enable the SYSHADOOP.COMPATIBILITY_MODE
- EXTERNAL keyword
  - Indicates that the table is not managed by the database manager
  - When the table is dropped, the definition is removed, the data remains unaffected.
- LOCATION keyword
  - Specifies the DFS directory to store the data files

```
CREATE EXTERNAL HADOOP TABLE T1
(
   C1 INT NOT NULL PRIMARY KEY CHECK (C1 > 0),
   C2 VARCHAR(10) NULL,
   …
)
…
LOCATION
'/user/biadmin/tables/user'
```

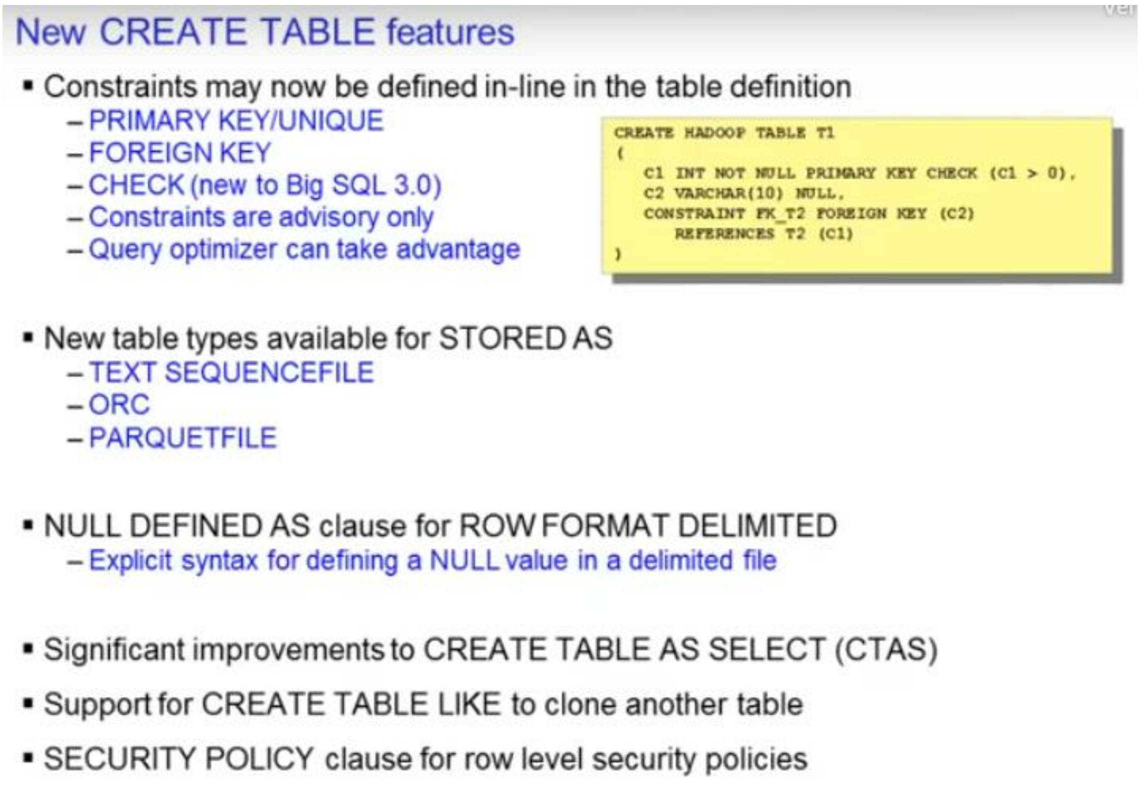
CREATE TABLE - partitioned tables

- Similar to Hive, Big SQL also has partitioned tables.
- Partitioned on one or more columns.
- Query predicates is used to eliminate unwanted data – speeding up the query.



```
CREATE HADOOP TABLE demo.sales (
  reps_id int,
  reps_name string,
  status int,
  salary double
)
PARTITIONED BY (
  state varchar(2)
);
```

```
select *
 from demo.sales
where state in ('AL', 'CA');
```

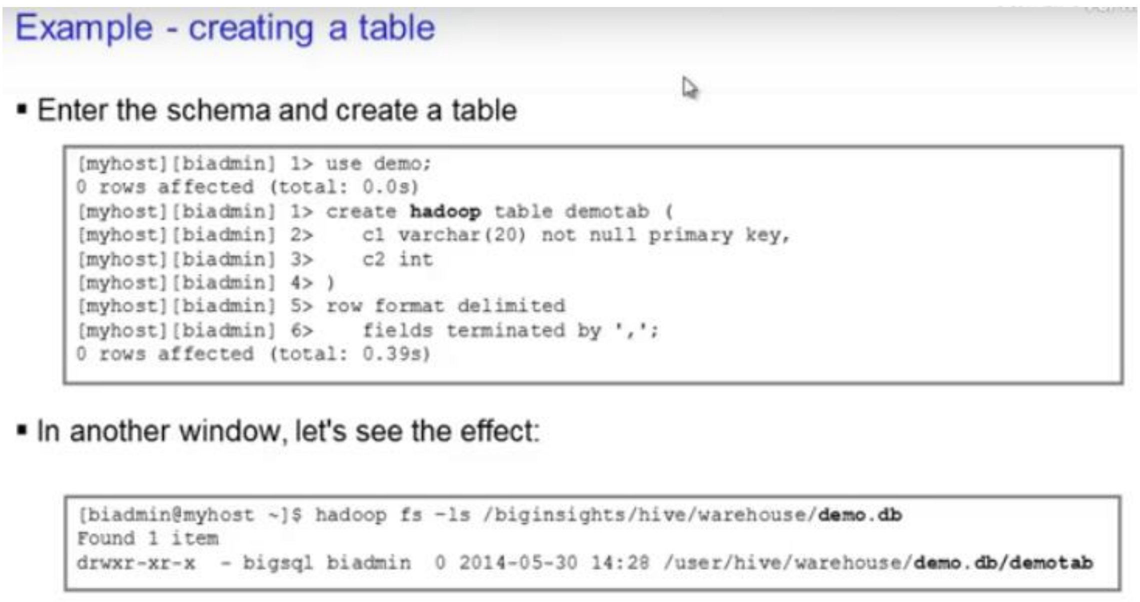


New CREATE TABLE features

- Constraints may now be defined in-line in the table definition
  - PRIMARY KEY/UNIQUE
  - FOREIGN KEY
  - CHECK (new to Big SQL 3.0)
  - Constraints are advisory only
  - Query optimizer can take advantage

```
CREATE HADOOP TABLE T1
(
   C1 INT NOT NULL PRIMARY KEY CHECK (C1 > 0),
   C2 VARCHAR(10) NULL,
   CONSTRAINT FK_T2 FOREIGN KEY (C2)
      REFERENCES T2 (C1)
)
```

- New table types available for STORED AS
  - TEXT SEQUENCEFILE
  - ORC
  - PARQUETFILE
- NULL DEFINED AS clause for ROW FORMAT DELIMITED
  - Explicit syntax for defining a NULL value in a delimited file
- Significant improvements to CREATE TABLE AS SELECT (CTAS)
- Support for CREATE TABLE LIKE to clone another table
- SECURITY POLICY clause for row level security policies



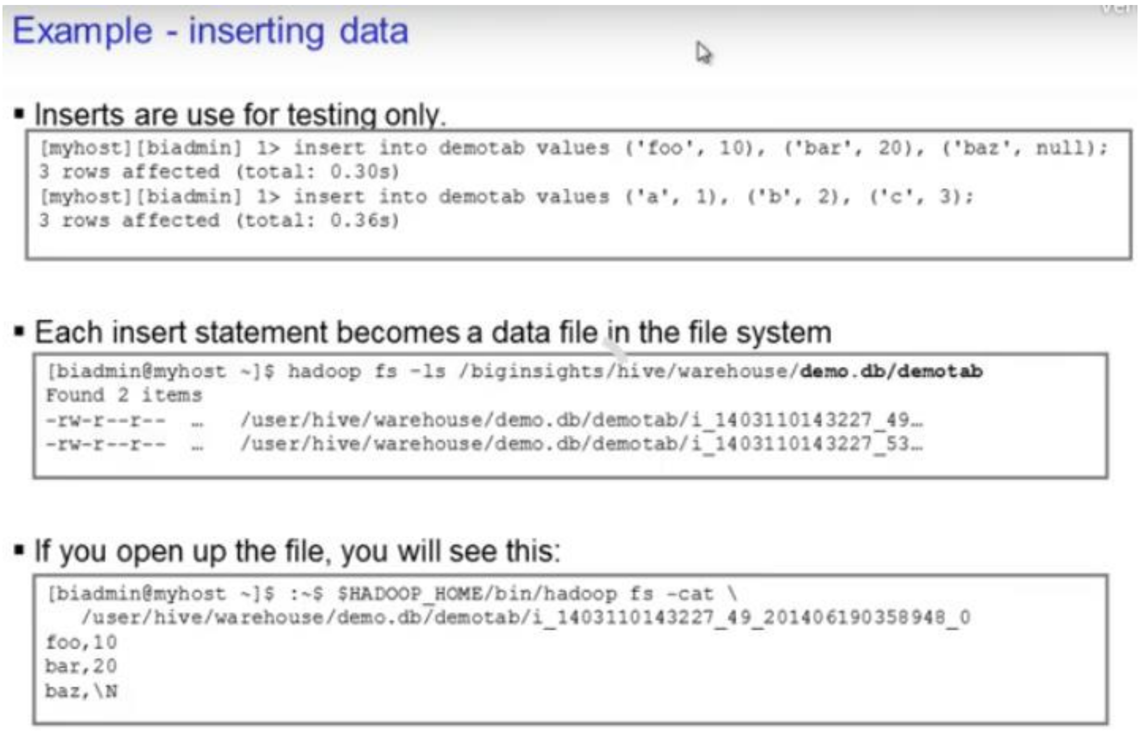
Example - creating a table

- Enter the schema and create a table

```
[myhost][biadmin] 1> use demo;
0 rows affected (total: 0.0s)
[myhost][biadmin] 1> create hadoop table demotab (
[myhost][biadmin] 2>    c1 varchar(20) not null primary key,
[myhost][biadmin] 3>    c2 int
[myhost][biadmin] 4> )
[myhost][biadmin] 5> row format delimited
[myhost][biadmin] 6>    fields terminated by ',';
0 rows affected (total: 0.39s)
```

- In another window, let's see the effect:

```
[biadmin@myhost ~]$ hadoop fs -ls /biginsights/hive/warehouse/demo.db
Found 1 item
drwxr-xr-x  - bigsql biadmin  0 2014-05-30 14:28 /user/hive/warehouse/demo.db/demotab
```



Example - inserting data

- Inserts are use for testing only.

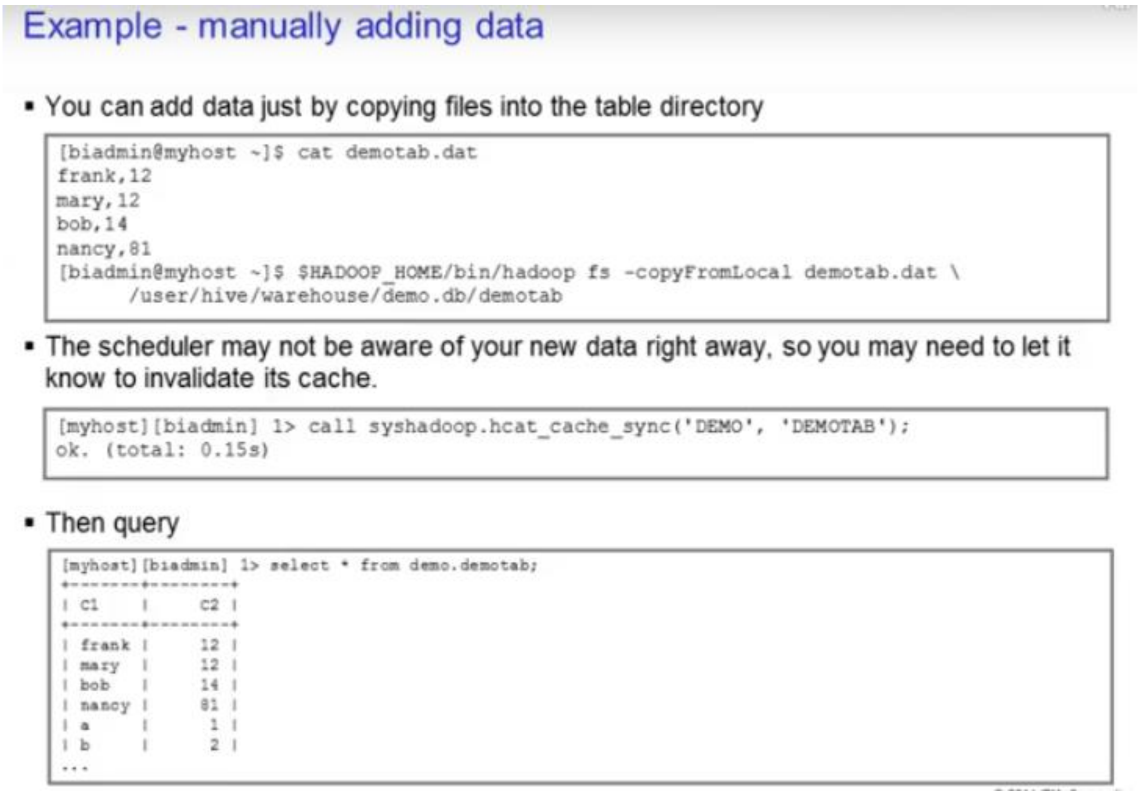
```
[myhost][biadmin] 1> insert into demotab values ('foo', 10), ('bar', 20), ('baz', null);
3 rows affected (total: 0.30s)
[myhost][biadmin] 1> insert into demotab values ('a', 1), ('b', 2), ('c', 3);
3 rows affected (total: 0.36s)
```

- Each insert statement becomes a data file in the file system

```
[biadmin@myhost ~]$ hadoop fs -ls /biginsights/hive/warehouse/demo.db/demotab
Found 2 items
-rw-r--r--  …   /user/hive/warehouse/demo.db/demotab/i_1403110143227_49…
-rw-r--r--  …   /user/hive/warehouse/demo.db/demotab/i_1403110143227_53…
```

- If you open up the file, you will see this:

```
[biadmin@myhost ~]$ :~$ $HADOOP_HOME/bin/hadoop fs -cat \
   /user/hive/warehouse/demo.db/demotab/i_1403110143227_49_201406190358948_0
foo,10
bar,20
baz,\N
```



Example - manually adding data

- You can add data just by copying files into the table directory

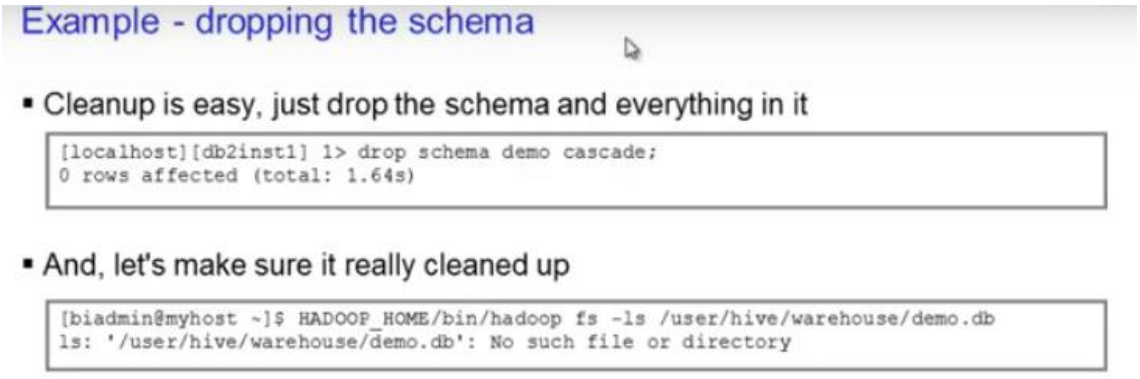
```
[biadmin@myhost ~]$ cat demotab.dat
frank,12
mary,12
bob,14
nancy,81
[biadmin@myhost ~]$ $HADOOP_HOME/bin/hadoop fs -copyFromLocal demotab.dat \
      /user/hive/warehouse/demo.db/demotab
```

- The scheduler may not be aware of your new data right away, so you may need to let it know to invalidate its cache.

```
[myhost][biadmin] 1> call syshadoop.hcat_cache_sync('DEMO', 'DEMOTAB');
ok. (total: 0.15s)
```

- Then query

```
[myhost][biadmin] 1> select * from demo.demotab;
+-------+--------+
| C1    |     C2 |
+-------+--------+
| frank |     12 |
| mary  |     12 |
| bob   |     14 |
| nancy |     81 |
| a     |      1 |
| b     |      2 |
...
```



Example - dropping the schema

- Cleanup is easy, just drop the schema and everything in it

```
[localhost][db2inst1] 1> drop schema demo cascade;
0 rows affected (total: 1.64s)
```

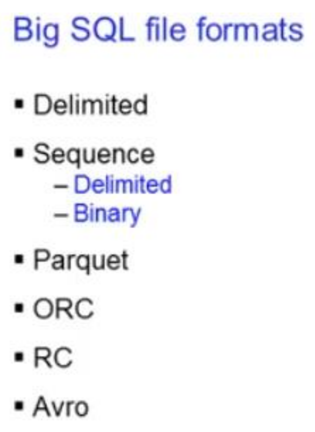
- And, let's make sure it really cleaned up

```
[biadmin@myhost ~]$ HADOOP_HOME/bin/hadoop fs -ls /user/hive/warehouse/demo.db
ls: '/user/hive/warehouse/demo.db': No such file or directory
```

You should know that schemas are a way to
organize tables. The default schema now matches your login name.
In Big SQL 1.0, the default schema was just "default".
If you want to specify another default schema, you use the USE command.
If you execute the USE command on a schema that does not currently exist, it gets created.
Once the USE command has been invoked, the specified schema becomes the default.
In the first screenshot, a new (non-existing) schema is created via the USE command. Then a table is created with an unqualified table name. (No schema was explicitly specified.)
This results in the table being created under the current default schema. If you had qualified the table, then it will be created in that schema instead of the default schema.
The second screenshot shows creating a schema using the CREATE SCHEMA command.
When a schema is created, by default, a directory is defined in /user/hive/warehouse. The name
of the created directory is schemaName.db where schemaName is the name that you specified.
So for the schema called demo, by default you should see a directory /user/hive/warehouse/demo.db.
It is possible to define a schema and explicitly place the schema directory elsewhere in HDFS besides the Hive warehouse.
You will need to specify the HADOOP keyword
to create a table for the Hadoop environment. You can omit that if you enable the environment
variable SYSHADOOP.COMPATIBILITY_MODE. Optionally, you can specify the EXTERNAL keyword to create
an external table. This indicates that the table is not managed by the database manager.
When the table is dropped, the definition of that table is removed, but the data remains untouched. Typically the EXTERNAL keyword is used in conjunction with the LOCATION keyword
to specify the directory within the DFS to store the data files.
Big SQL has another similarity with Hive, partitioned
tables. Big SQL can created partitioned tables to split up the data into multiple files.
The benefit of this is the access to the data does not require all of the data to be read.
The query predicate will determine which partition to scan through.
Let us look at the sample code here: In this code, you are creating a partitioned
table on the state column. Notice that you do not specify the state column in the original table definition, but in the PARTITIONED BY clause. When you run the query to select data from a particular state or states, only the partitions specified in the query are retrieved.
In the example, select * from demo.sales where state in ("AL",
"CA") only the partitions where state=AL and state=CA
are retrieved.
There are some new CREATE TABLE features in

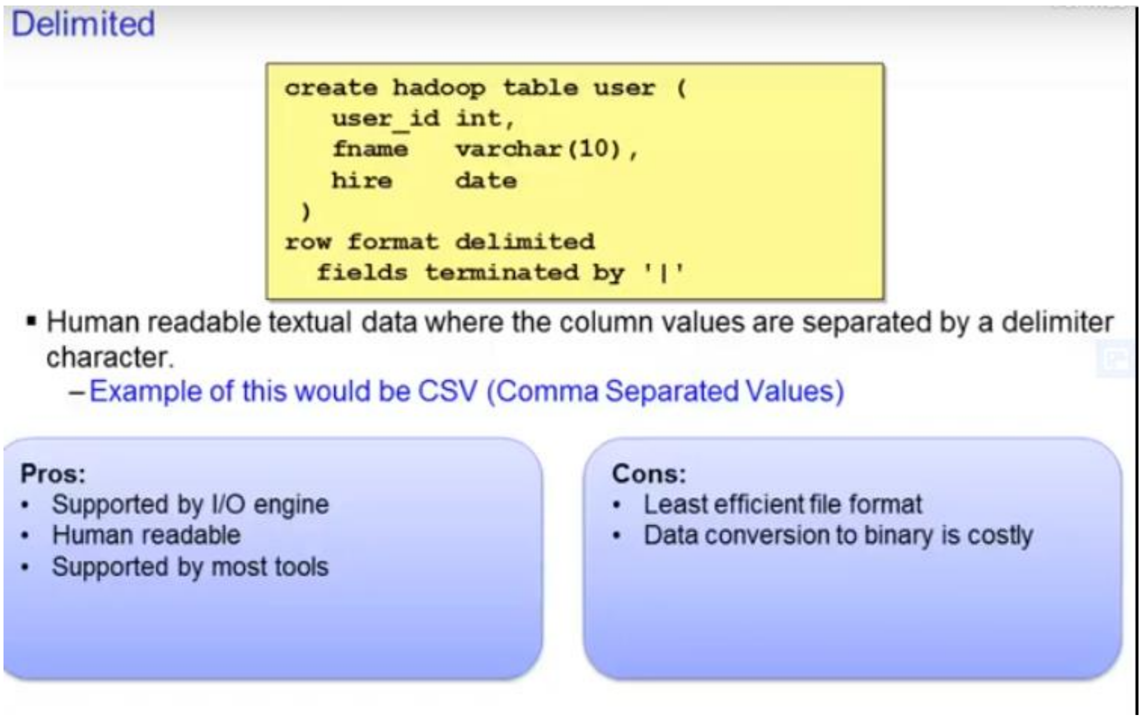
Big SQL that were not available in Big SQL 1.0.
Namely, constraints could not be defined in-line in the table definition.
In Big SQL 1.0, constraints were only allowed via ALTER statements.
In the current Big SQL, you can define: PRIMARY KEY/UNIQUE,
FOREIGN KEY, CHECK (which is new to Big SQL 3.0)
Note that constraints are only advisory, or applied on read, and not actually enforced.
The query optimizer, however, can leverage these constraints to create better query plans.
So even though they are not enforced, they are useful for query optimization.
There are also new table types available for the STORED AS clause. Big SQL supports TEXT SEQUENCE FILE, Hive's ORC file, PARQUETFILE, which Twitter and Cloudera have been working on, and a few others as well. These file formats will be discussed in more detail in the following slides. The NULL DEFINED AS syntax allows you to explicitly
declare how a NULL value is to be represented. Significant improvements have been made to the CREATE TABLE AS SELECT(CTAS). There is support for CREATE TABLE LIKE to
clone another table. There is also a SECURITY POLICY clause for
row level security policies.
Here is an example of creating a table.
The default schema was overridden to become demo via the USE command. Then an unqualified
table, demotab, was created using the hadoop keyword.
This means that the table is defined to both Big SQL and Hive. If the hadoop keyword had not been specified, then the table would have only been defined in the database management system. The table has two columns, a primary key column
of type varchar(20) and an int column. The table is row format delimited and the fields
are terminated with a comma. Once the table has been created, assuming that the demo schema
was defined in its default location, it would be located as a directory within the default warehouse directory, /user/hive/warehouse/demo.db/demotab. It is possible to create a table with an explicit
schema as well. So assuming that the default schema is demo but you wanted to create a table, t1, under the xyz schema, you could code:
create hadoop table xyz.t1 (...)
Here is an example of inserting some data.
Remember that the insert statement is good for testing only. It is extremely inefficient
for querying and using in a production environment. Normally you would want to use the bulk operations
like LOAD or INSERT from SELECT, which you will see in the lab exercise associated with
this unit. So here you have two sample inserts into the
demotab table with 3 columns. Notice that the third column contains a NULL.
Each of these inserts produces a data file in the file system. So if you do a listing
in the HDFS in the default warehouse directory, you should see two data files in it.
If you open up the file, you should see the actual values. Notice that the NULL is represented by the \N in the Hive system. You can specify a different NULL representation if you wish.
Here is an example of manually adding data
by copying the files into the table directory. In the first screenshot, you do a cat of the
data file to see its content. Then copy that file into the database directory within the
HDFS. The scheduler may not be aware of the new
data, so you need tell it to invalidate its cache. You do so by executing the command:
call syshadoop.hcat_cache_sync('DEMO', 'DEMOTAB');
Then you can run the query against that table to see the new data.
Cleanup is simple. This will remove the schema
as well as all tables. You just execute the statement:

drop schema demo cascade; To make sure it is really gone, just go back to do a listing of the demo.db directory.



Big SQL file formats

- Delimited
- Sequence
  - Delimited
  - Binary
- Parquet
- ORC
- RC
- Avro



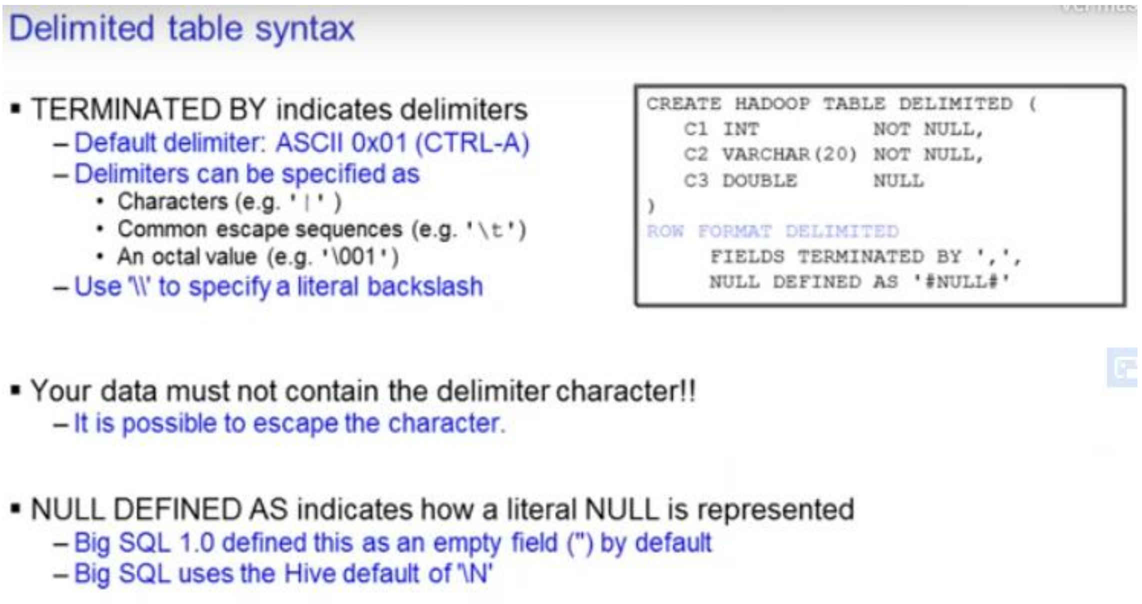
Delimited

```
create hadoop table user (
   user_id int,
   fname   varchar(10),
   hire    date
 )
row format delimited
  fields terminated by '|'
```

- Human readable textual data where the column values are separated by a delimiter character.
  - Example of this would be CSV (Comma Separated Values)

Pros:
- Supported by I/O engine
- Human readable
- Supported by most tools

Cons:
- Least efficient file format
- Data conversion to binary is costly

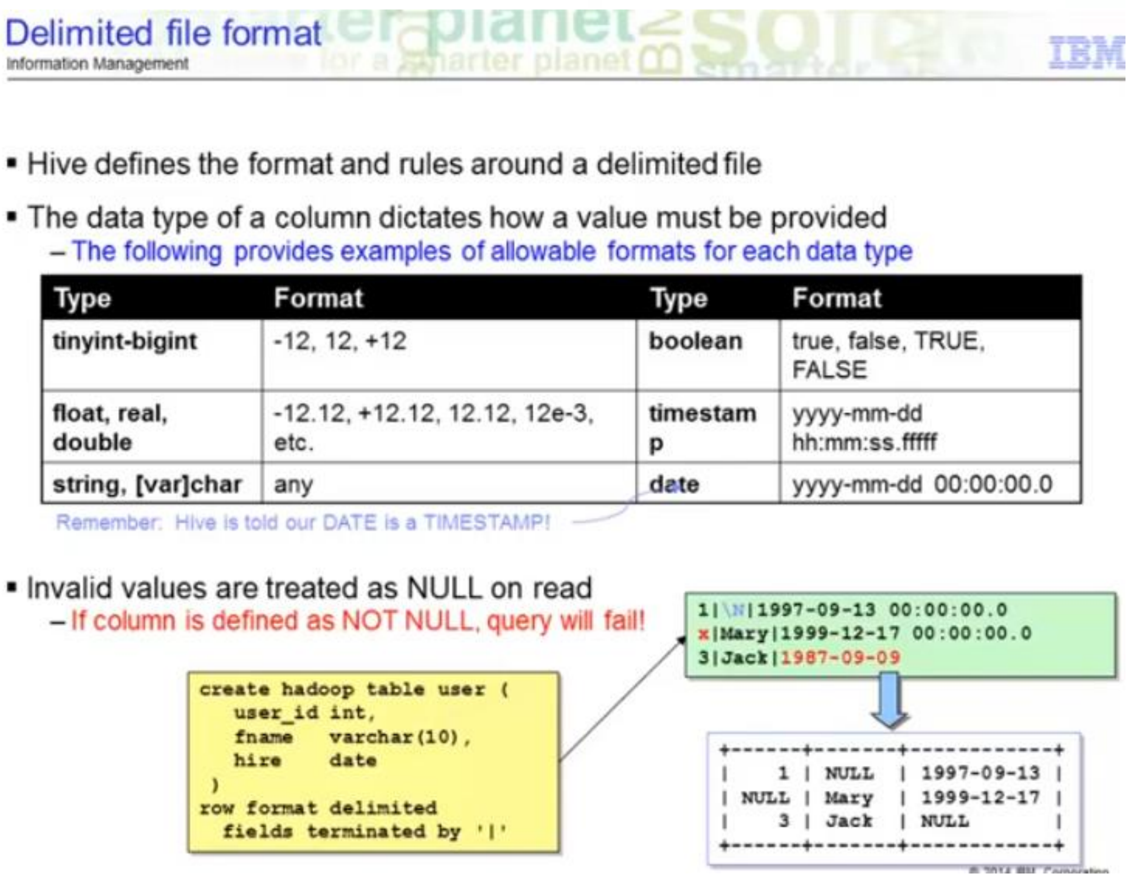


Delimited table syntax

- TERMINATED BY indicates delimiters
  - Default delimiter: ASCII 0x01 (CTRL-A)
  - Delimiters can be specified as
    - Characters (e.g. '|')
    - Common escape sequences (e.g. '\t')
    - An octal value (e.g. '\001')
  - Use '\\' to specify a literal backslash

```
CREATE HADOOP TABLE DELIMITED (
   C1 INT         NOT NULL,
   C2 VARCHAR(20) NOT NULL,
   C3 DOUBLE      NULL
)
ROW FORMAT DELIMITED
     FIELDS TERMINATED BY ',',
     NULL DEFINED AS '#NULL#'
```

- Your data must not contain the delimiter character!!
  - It is possible to escape the character.
- NULL DEFINED AS indicates how a literal NULL is represented
  - Big SQL 1.0 defined this as an empty field ('') by default
  - Big SQL uses the Hive default of '\N'



Delimited file format

Information Management

IBM

- Hive defines the format and rules around a delimited file
- The data type of a column dictates how a value must be provided
  - The following provides examples of allowable formats for each data type

Type	Format	Type	Format
tinyint-bigint	-12, 12, +12	boolean	true, false, TRUE, FALSE
float, real, double	-12.12, +12.12, 12.12, 12e-3, etc.	timestamp	yyyy-mm-dd hh:mm:ss.fffff
string, [var]char	any	date	yyyy-mm-dd 00:00:00.0

Remember: Hive is told our DATE is a TIMESTAMP!

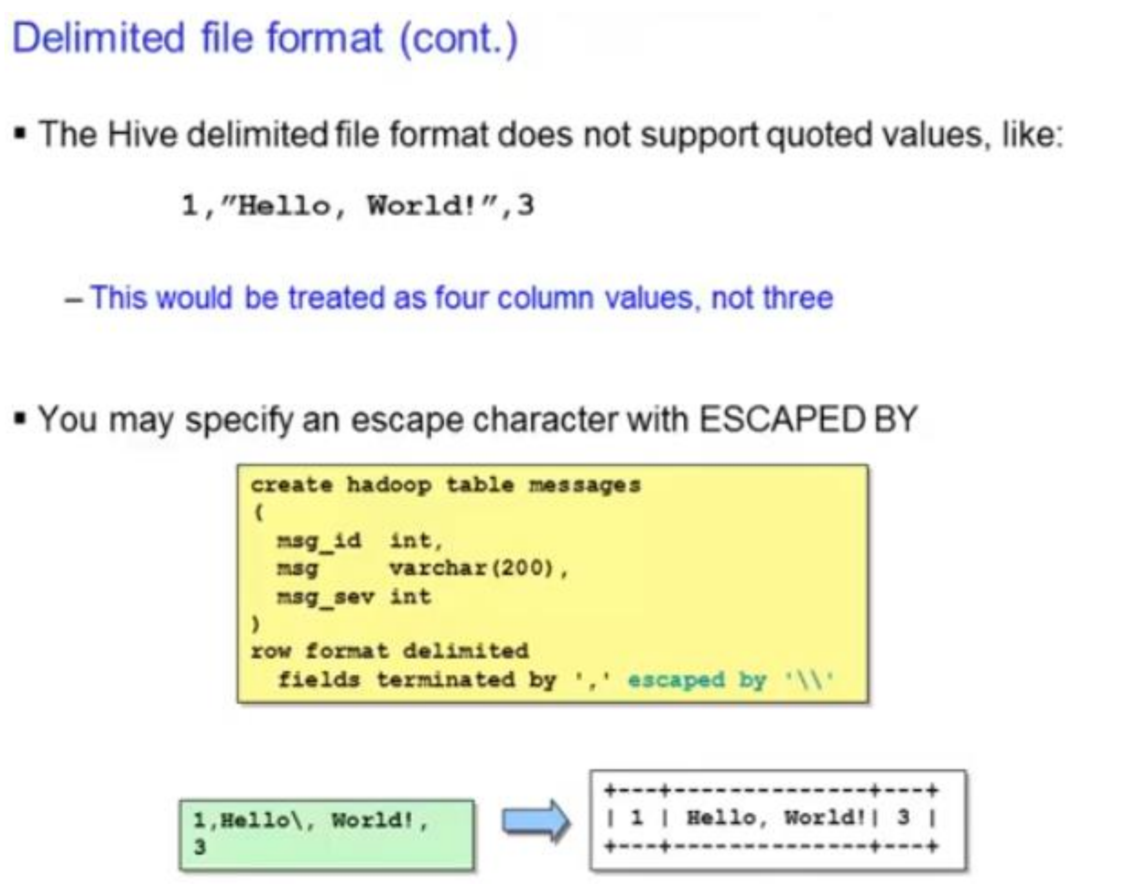
- Invalid values are treated as NULL on read
  - If column is defined as NOT NULL, query will fail!

```
create hadoop table user (
   user_id int,
   fname   varchar(10),
   hire    date
 )
row format delimited
  fields terminated by '|'
```

```
1|\N|1997-09-13 00:00:00.0
x|Mary|1999-12-17 00:00:00.0
3|Jack|1987-09-09
```

```
+------+-------+------------+
|    1 | NULL  | 1997-09-13 |
| NULL | Mary  | 1999-12-17 |
|    3 | Jack  | NULL       |
+------+-------+------------+
```

© 2014 IBM Corporation



Delimited file format (cont.)

- The Hive delimited file format does not support quoted values, like:

```
1,"Hello, World!",3
```

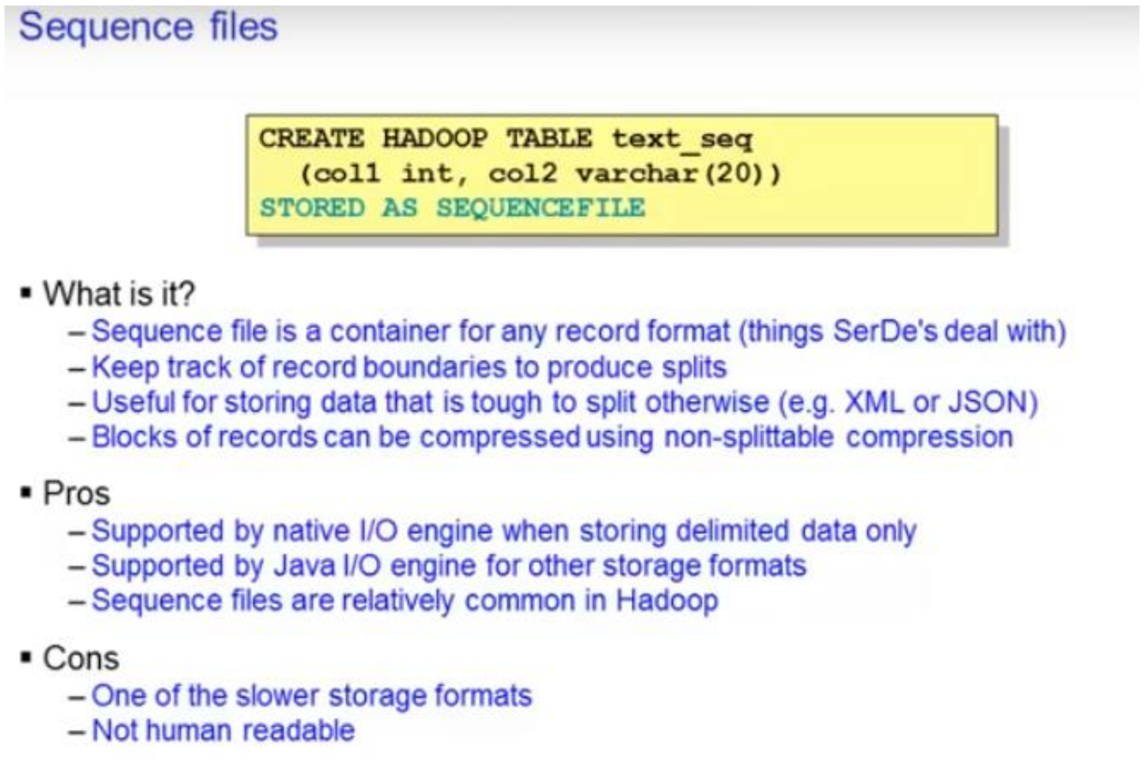
  - This would be treated as four column values, not three

- You may specify an escape character with ESCAPED BY

```
create hadoop table messages
(
  msg_id  int,
  msg     varchar(200),
  msg_sev int
)
row format delimited
  fields terminated by ',' escaped by '\\'
```

```
1,Hello\, World!,
3
```

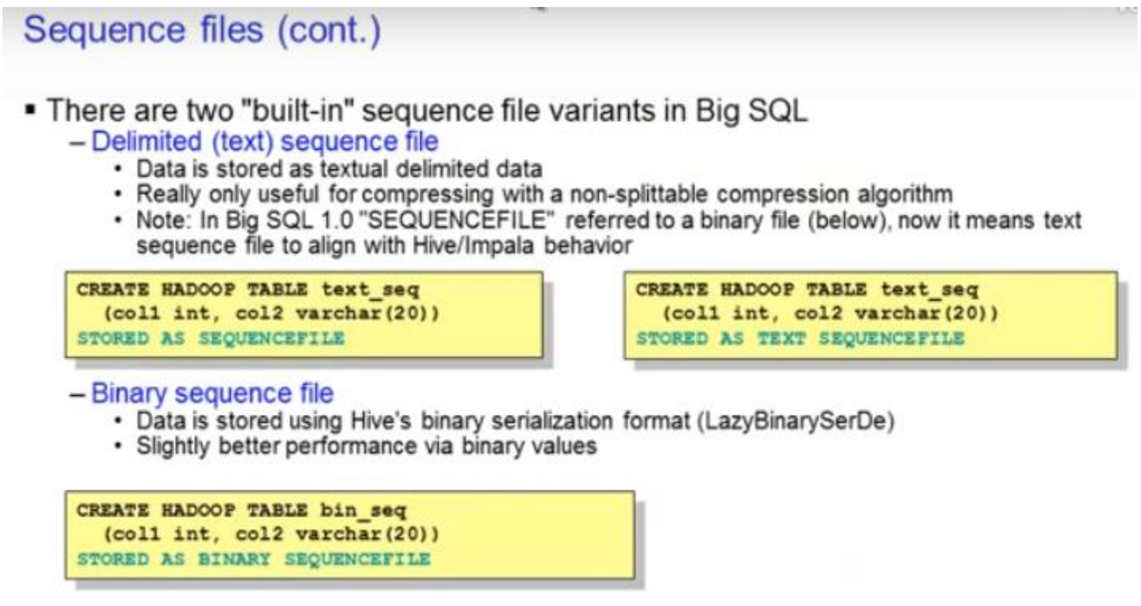
```
+---+---------------+---+
| 1 | Hello, World!| 3 |
+---+---------------+---+
```



Sequence files

```
CREATE HADOOP TABLE text_seq
   (col1 int, col2 varchar(20))
STORED AS SEQUENCEFILE
```

- What is it?
  - Sequence file is a container for any record format (things SerDe's deal with)
  - Keep track of record boundaries to produce splits
  - Useful for storing data that is tough to split otherwise (e.g. XML or JSON)
  - Blocks of records can be compressed using non-splittable compression
- Pros
  - Supported by native I/O engine when storing delimited data only
  - Supported by Java I/O engine for other storage formats
  - Sequence files are relatively common in Hadoop
- Cons
  - One of the slower storage formats
  - Not human readable



Sequence files (cont.)

- There are two "built-in" sequence file variants in Big SQL
  - Delimited (text) sequence file
    - Data is stored as textual delimited data
    - Really only useful for compressing with a non-splittable compression algorithm
    - Note: In Big SQL 1.0 "SEQUENCEFILE" referred to a binary file (below), now it means text sequence file to align with Hive/Impala behavior

```
CREATE HADOOP TABLE text_seq
  (col1 int, col2 varchar(20))
STORED AS SEQUENCEFILE
```

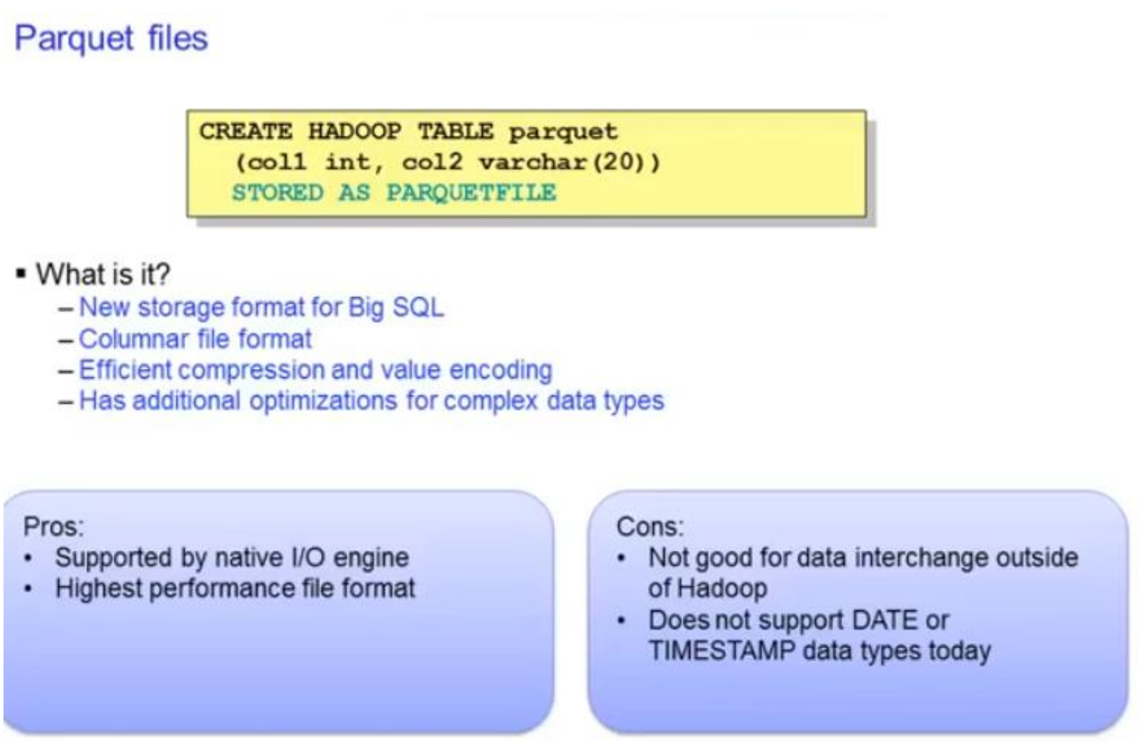
```
CREATE HADOOP TABLE text_seq
  (col1 int, col2 varchar(20))
STORED AS TEXT SEQUENCEFILE
```

  - Binary sequence file
    - Data is stored using Hive's binary serialization format (LazyBinarySerDe)
    - Slightly better performance via binary values

```
CREATE HADOOP TABLE bin_seq
  (col1 int, col2 varchar(20))
STORED AS BINARY SEQUENCEFILE
```

Here are the Big SQL file formats that will
be covered in detail in the following slides. Delimited,
Sequence Delimited,
Sequence Binary, Parquet,
ORC, RC,
and Avro
A delimited file is essentially a human readable
text file where the column values are separated by a delimiter character. A common example
of this would be CSV (Comma separated values). Pros:
- Supported by the I/O engine - Human readable
- Supported by most tools Cons:
- Least efficient file format - Data conversion to binary is costly
The TERMINATED BY clause allows you to specify
the type of delimiter your file uses. The default delimiter is the ASCII 0x01 (CTRL-A).
Typically you would probably use a pipe symbol or some escape characters. If you need to
use a backslash, you would use a double backslash. Your data must not contain the delimited character,
or else the data will be messed up. You can escape characters and I will go over this
in a bit. There is a NULL DEFINED AS clause that lets
you indicate how you want a NULL value to be represented. In Big SQL, the default Hive
representation of a NULL value is the \N. In the previous version of Big SQL, it was
an empty field by default.
Here is the file format for a DELIMITED file.
Hive defines the format and rules around a delimited file. In the chart are examples
of allowable formats for each data type. Remember that Hive is told that our DATE is a TIMESTAMP,
so the format is a timestamp. If you provide data in an invalid format,
the values are treated as NULL on read. This causes the query to fail if your column is not
defined as NOT NULL. In this example, you see that a table has
three columns, a user_id of type int, a fname of type varchar(10) and a hire of type date.
Then in the data file, you have three rows. The first row contains the values of type

int, \N and the timestamp. This gets added correctly because the data format matches
the columns' data types. The second row, in the first column, it is expecting an int,
but it gets a char, so when you insert the data, you get a NULL. If the column was defined
with a NOT NULL, the query fails. In the third row, the last column's date is just a date,
without an additional time value. It is inserted as a NULL because Hive expects a TIMESTAMP
for the DATE column.
Here you see how to escape certain characters
if they are being used as a delimiter. Hive does not support quoted values, so even if
you provide a quotation around a value, it still counts any delimited within the quotation
marks. For example, if you needed to insert the value
1, "Hello, World!", 3 This would be treated as four columns because
of the three commas that separate the data. To do this successfully, you must specify
an escape character using the ESCAPED BY clause. So assuming your escape character is the backslash,
you would insert the row as: 1, Hello\, World!, 3
That correctly inserts the rows into the three columns of that table.
Sequence file is a container for any record
format with which SerDe's deal. It maintains the boundaries of the files to produce splits
effectively. This file format is useful for storing data that is normally tough to split
like XML or JSON. There is an option to compress blocks of records using non-splittable compression.
Some of the pros of using sequence files are that it is supported by the native I/O engine
when storing delimited data. For other storage formats, it is supported by the Java I/O engine.
It is also one of the more commonly used formats in Hadoop.
A couple of cons of sequence files are that they are relatively slow compared to other
formats and they are not human readable.
Sequence files come in two flavors. You have
the delimited (text) sequence file and the binary sequence file.
The delimited sequence file stores the data as textual delimited data. This is only useful
for compressing with non-splittable compression algorithms.
In Big SQL 1.0, SEQUENCEFILE referred to the binary file, which is the second of the two
sequence file formats. In the current release of Big SQL, SEQUENCEFILE is in text format.
The binary sequence file stores the data using Hive's binary serialization format using
the LazyBinarySerDe. It yields slightly better performance than the text version.



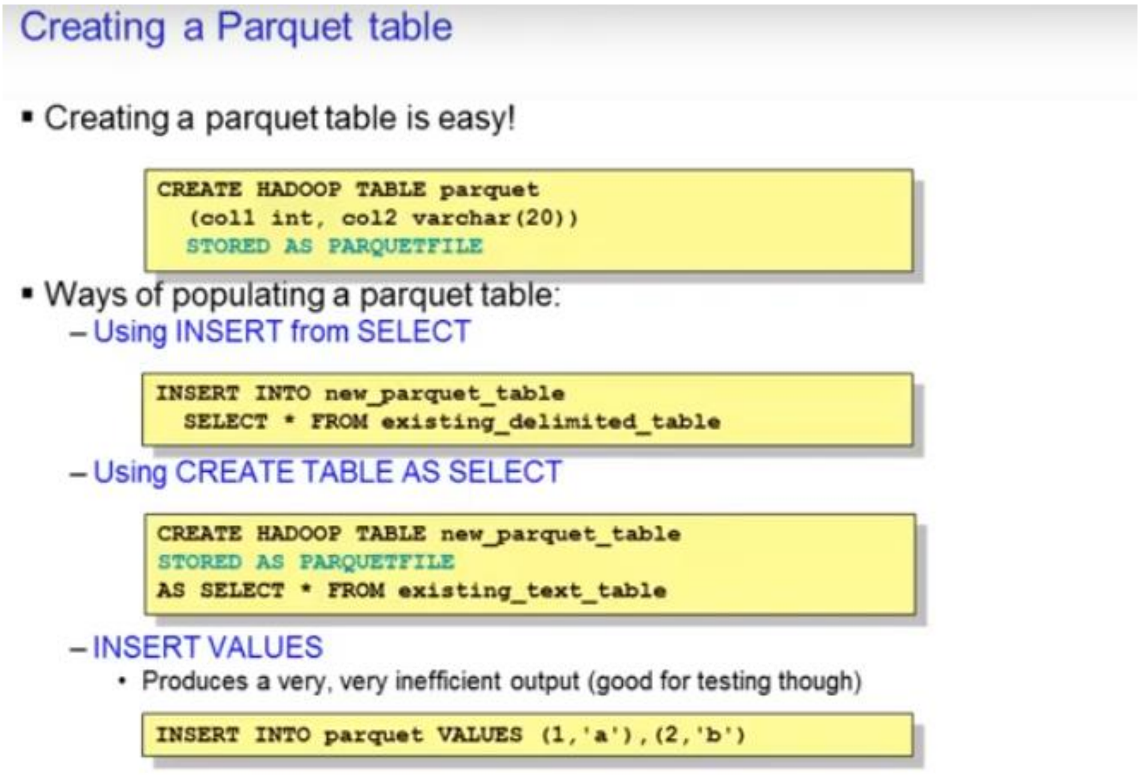
Parquet files

```
CREATE HADOOP TABLE parquet
  (col1 int, col2 varchar(20))
  STORED AS PARQUETFILE
```

- What is it?
  - New storage format for Big SQL
  - Columnar file format
  - Efficient compression and value encoding
  - Has additional optimizations for complex data types

Pros:
- Supported by native I/O engine
- Highest performance file format

Cons:
- Not good for data interchange outside of Hadoop
- Does not support DATE or TIMESTAMP data types today



Creating a Parquet table

- Creating a parquet table is easy!

```
CREATE HADOOP TABLE parquet
  (col1 int, col2 varchar(20))
  STORED AS PARQUETFILE
```

- Ways of populating a parquet table:
  - Using INSERT from SELECT

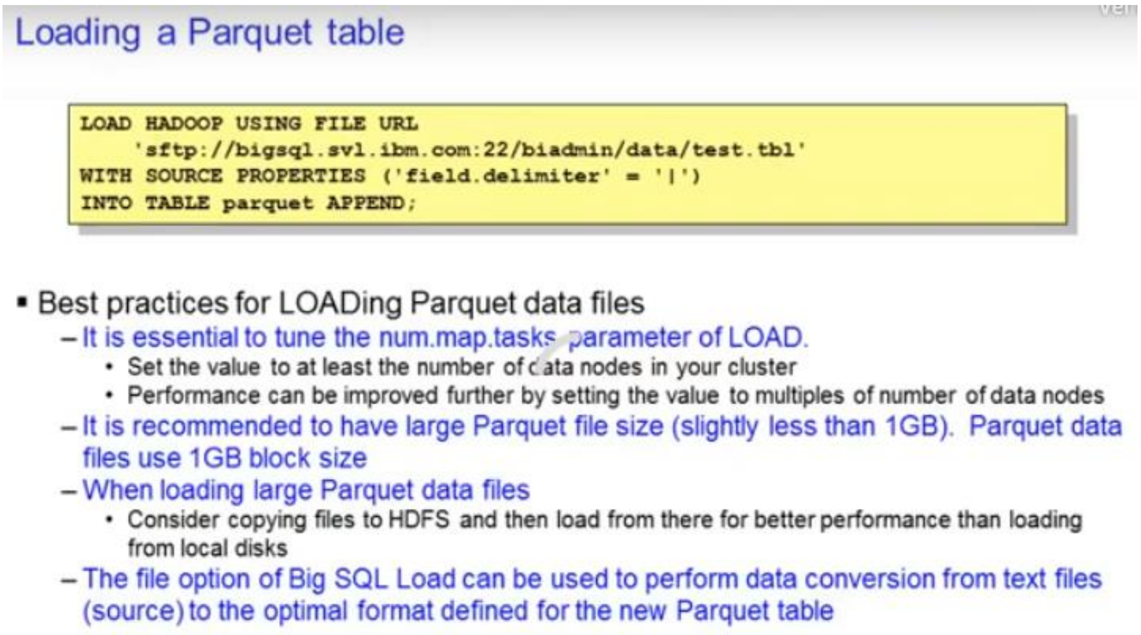
```
INSERT INTO new_parquet_table
  SELECT * FROM existing_delimited_table
```

  - Using CREATE TABLE AS SELECT

```
CREATE HADOOP TABLE new_parquet_table
STORED AS PARQUETFILE
AS SELECT * FROM existing_text_table
```

  - INSERT VALUES
    - Produces a very, very inefficient output (good for testing though)

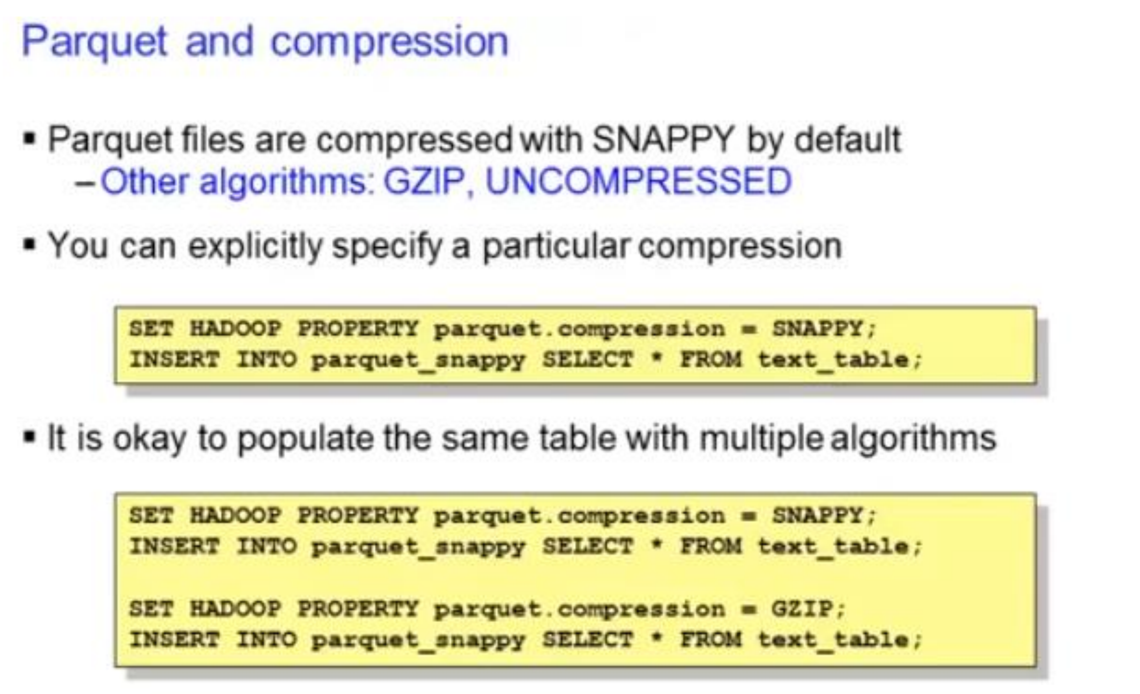
```
INSERT INTO parquet VALUES (1,'a'),(2,'b')
```



Loading a Parquet table

```
LOAD HADOOP USING FILE URL
     'sftp://bigsql.svl.ibm.com:22/biadmin/data/test.tbl'
WITH SOURCE PROPERTIES ('field.delimiter' = '|')
INTO TABLE parquet APPEND;
```

- Best practices for LOADing Parquet data files
  - It is essential to tune the num.map.tasks parameter of LOAD.
    - Set the value to at least the number of data nodes in your cluster
    - Performance can be improved further by setting the value to multiples of number of data nodes
  - It is recommended to have large Parquet file size (slightly less than 1GB). Parquet data files use 1GB block size
  - When loading large Parquet data files
    - Consider copying files to HDFS and then load from there for better performance than loading from local disks
  - The file option of Big SQL Load can be used to perform data conversion from text files (source) to the optimal format defined for the new Parquet table



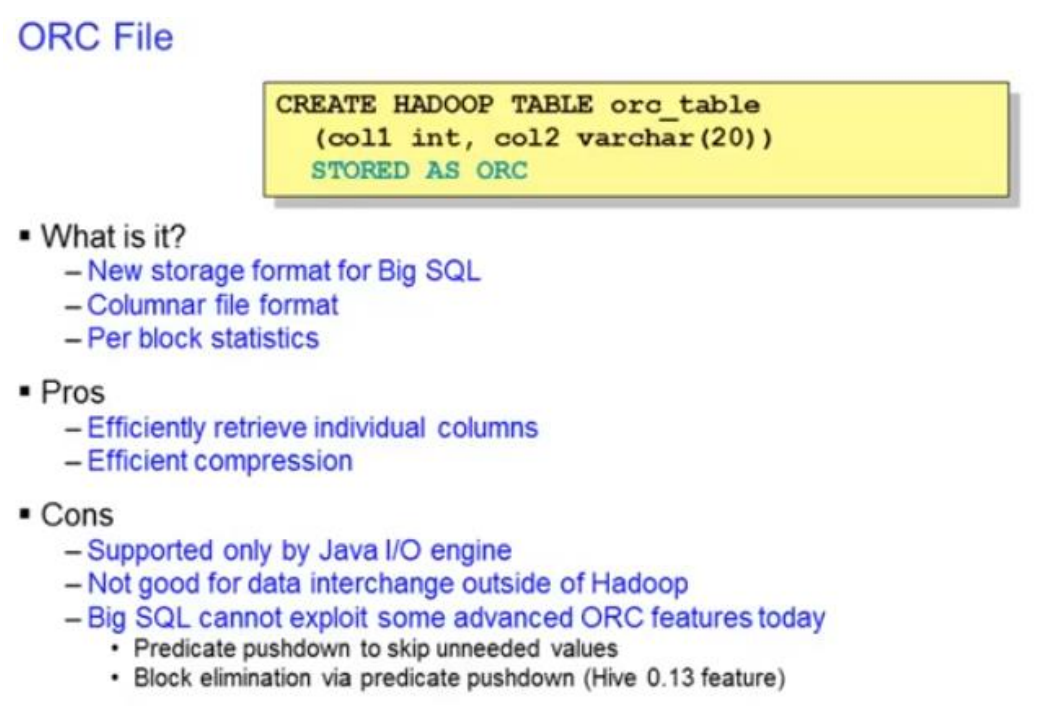
Parquet and compression

- Parquet files are compressed with SNAPPY by default
  - Other algorithms: GZIP, UNCOMPRESSED
- You can explicitly specify a particular compression

```
SET HADOOP PROPERTY parquet.compression = SNAPPY;
INSERT INTO parquet_snappy SELECT * FROM text_table;
```

- It is okay to populate the same table with multiple algorithms

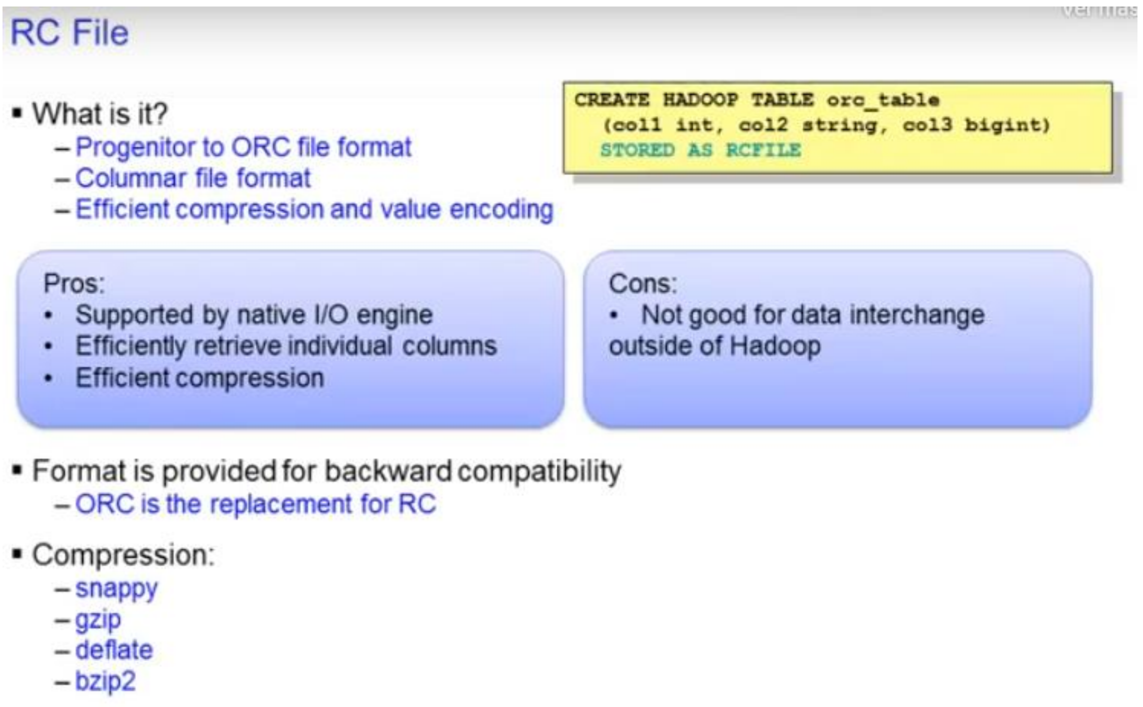
```
SET HADOOP PROPERTY parquet.compression = SNAPPY;
INSERT INTO parquet_snappy SELECT * FROM text_table;

SET HADOOP PROPERTY parquet.compression = GZIP;
INSERT INTO parquet_snappy SELECT * FROM text_table;
```



ORC File

```
CREATE HADOOP TABLE orc_table
  (col1 int, col2 varchar(20))
  STORED AS ORC
```

- What is it?
  - New storage format for Big SQL
  - Columnar file format
  - Per block statistics
- Pros
  - Efficiently retrieve individual columns
  - Efficient compression
- Cons
  - Supported only by Java I/O engine
  - Not good for data interchange outside of Hadoop
  - Big SQL cannot exploit some advanced ORC features today
    - Predicate pushdown to skip unneeded values
    - Block elimination via predicate pushdown (Hive 0.13 feature)



RC File

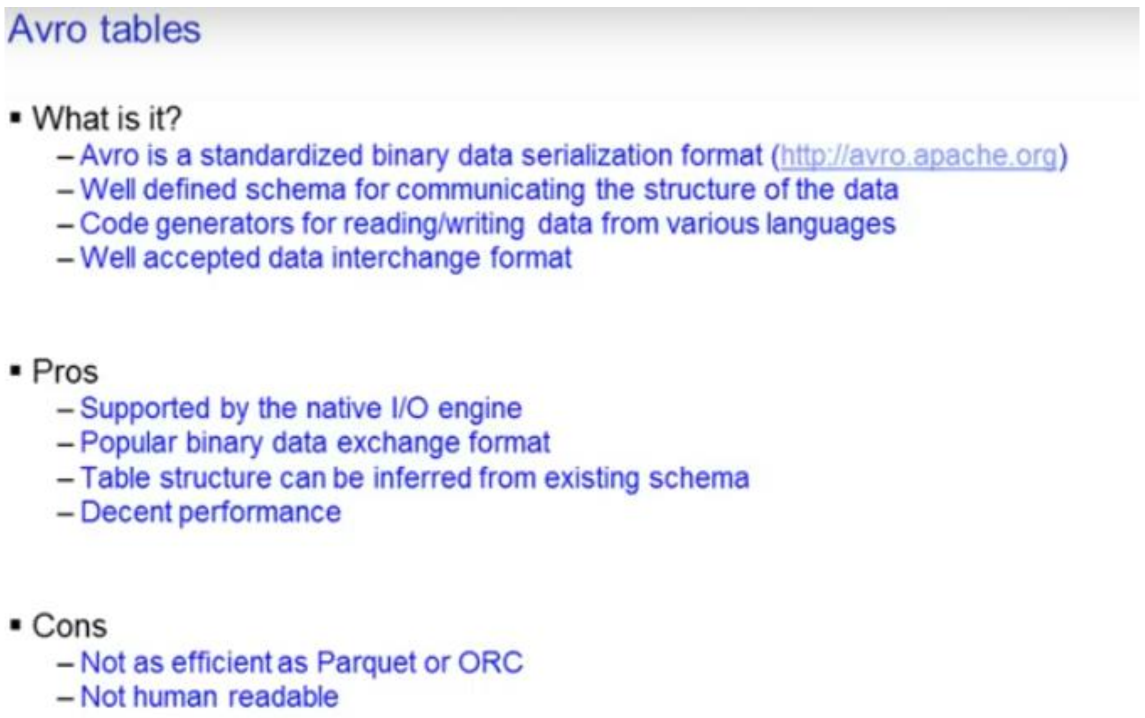
- What is it?
  - Progenitor to ORC file format
  - Columnar file format
  - Efficient compression and value encoding

```
CREATE HADOOP TABLE orc_table
  (col1 int, col2 string, col3 bigint)
  STORED AS RCFILE
```

Pros:
- Supported by native I/O engine
- Efficiently retrieve individual columns
- Efficient compression

Cons:
- Not good for data interchange outside of Hadoop

- Format is provided for backward compatibility
  - ORC is the replacement for RC
- Compression:
  - snappy
  - gzip
  - deflate
  - bzip2

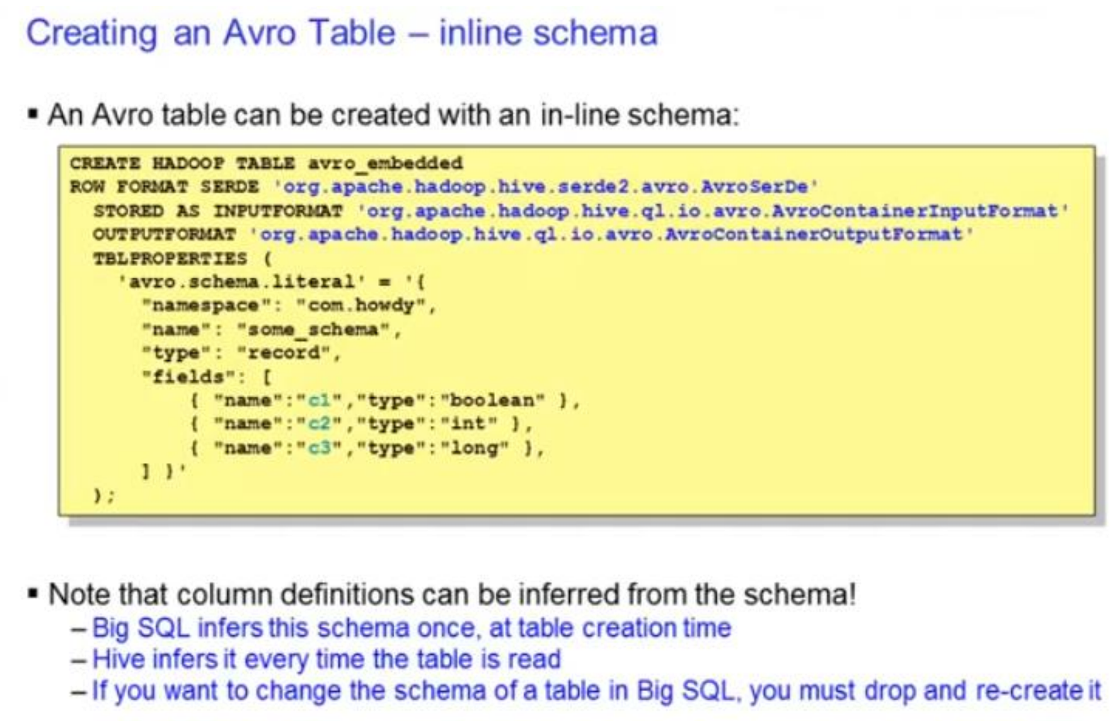


Avro tables

- What is it?
  - Avro is a standardized binary data serialization format (http://avro.apache.org)
  - Well defined schema for communicating the structure of the data
  - Code generators for reading/writing data from various languages
  - Well accepted data interchange format

- Pros
  - Supported by the native I/O engine
  - Popular binary data exchange format
  - Table structure can be inferred from existing schema
  - Decent performance

- Cons
  - Not as efficient as Parquet or ORC
  - Not human readable

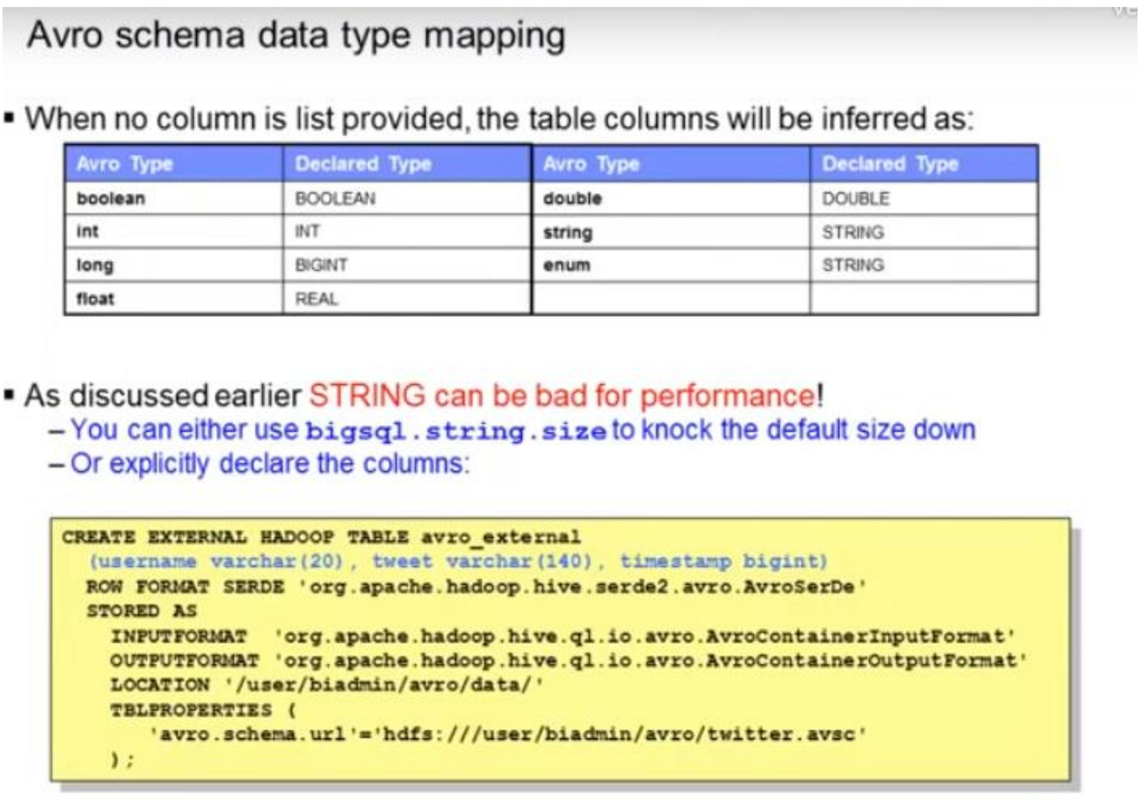


Creating an Avro Table – inline schema

- An Avro table can be created with an in-line schema:

```
CREATE HADOOP TABLE avro_embedded
ROW FORMAT SERDE 'org.apache.hadoop.hive.serde2.avro.AvroSerDe'
  STORED AS INPUTFORMAT 'org.apache.hadoop.hive.ql.io.avro.AvroContainerInputFormat'
  OUTPUTFORMAT 'org.apache.hadoop.hive.ql.io.avro.AvroContainerOutputFormat'
  TBLPROPERTIES (
    'avro.schema.literal' = '{
      "namespace": "com.howdy",
      "name": "some_schema",
      "type": "record",
      "fields": [
          { "name":"c1","type":"boolean" },
          { "name":"c2","type":"int" },
          { "name":"c3","type":"long" },
      ] }'
  );
```

- Note that column definitions can be inferred from the schema!
  - Big SQL infers this schema once, at table creation time
  - Hive infers it every time the table is read
  - If you want to change the schema of a table in Big SQL, you must drop and re-create it



Avro schema data type mapping

- When no column is list provided, the table columns will be inferred as:

Avro Type	Declared Type	Avro Type	Declared Type
boolean	BOOLEAN	double	DOUBLE
int	INT	string	STRING
long	BIGINT	enum	STRING
float	REAL		

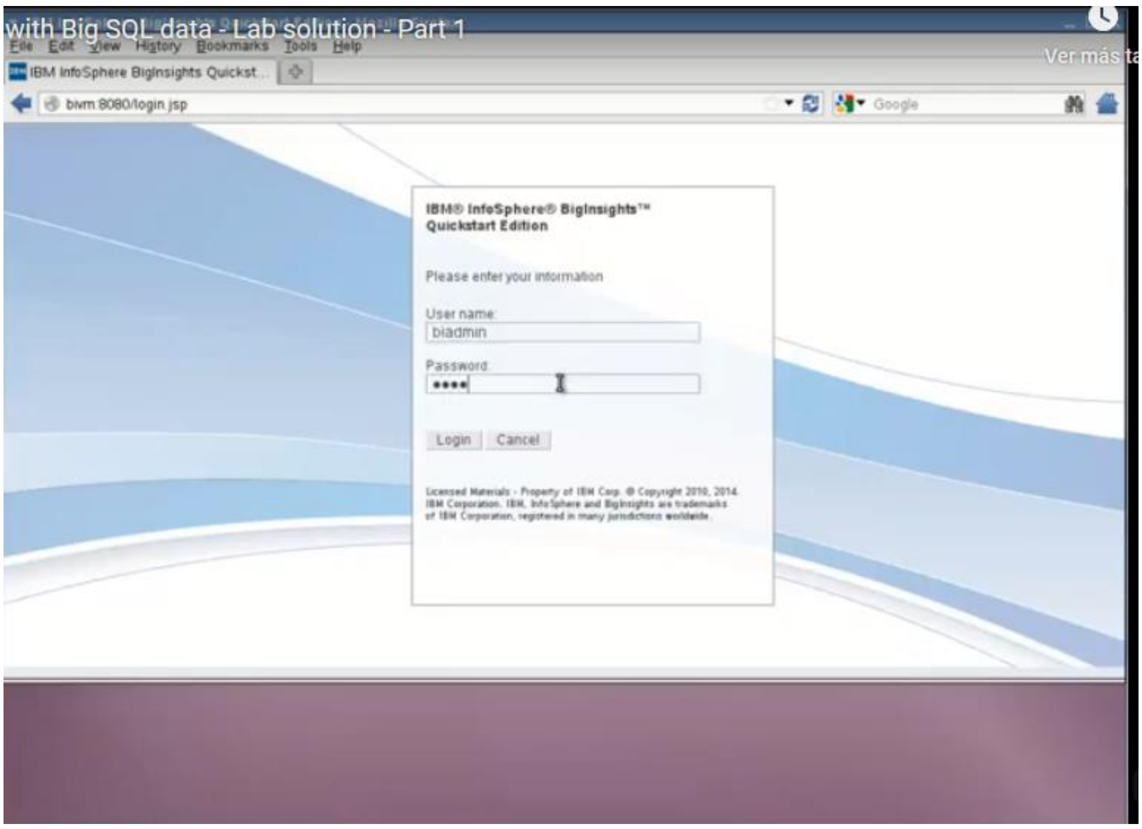
- As discussed earlier STRING can be bad for performance!
  - You can either use `bigsql.string.size` to knock the default size down
  - Or explicitly declare the columns:

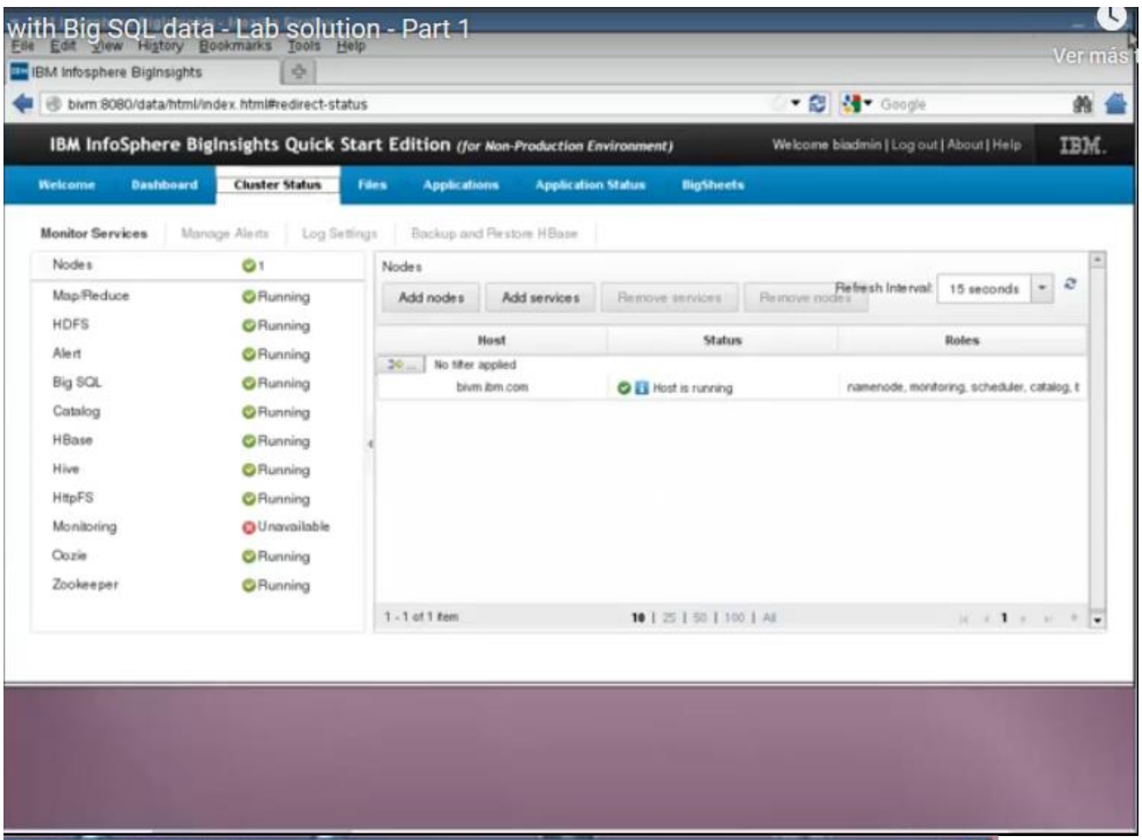
```
CREATE EXTERNAL HADOOP TABLE avro_external
  (username varchar(20), tweet varchar(140), timestamp bigint)
  ROW FORMAT SERDE 'org.apache.hadoop.hive.serde2.avro.AvroSerDe'
  STORED AS
    INPUTFORMAT  'org.apache.hadoop.hive.ql.io.avro.AvroContainerInputFormat'
    OUTPUTFORMAT 'org.apache.hadoop.hive.ql.io.avro.AvroContainerOutputFormat'
    LOCATION '/user/biadmin/avro/data/'
    TBLPROPERTIES (
       'avro.schema.url'='hdfs:///user/biadmin/avro/twitter.avsc'
    );
```

Parquet file is the new storage format for
Big SQL. It is a columnar storage format with efficient compression and value encoding.
It also has optimizations for complex data types.
Pros: - Parquet file is supported by the native
I/O engine - Highest performance among the file formats.
Cons: - Not good for data interchange outside of
Hadoop. - Does not support the DATE and TIMESTAMP
types today.
Here are some examples of how to work with
Parquet tables. You saw how to create a parquet table in the previous slide.
CREATE HADOOP TABLE parquet (col1 int, col2 varchar(20)) STORED AS PARQUETFILE;
Essentially everything is the same as creating a regular table, except you specify the STORED
AS PARQUETFILE clause. There are three ways to populate a parquet
table. 1. Using INSERT from SELECT statement to basically
transform an existing delimited table into a parquet table.
INSERT INTO new_parquet_table SELECT * FROM existing_delimited_table;
2. Using a CREATE TABLE AS SELECT CREATE HADOOP TABLE new_parquet_table STORED
AS PARQUETFILE AS SELECT * FROM existing_text_table; 3. Using INSERT to test out your
queries.
INSERT INTO parquet VALUES (1, 'a'), (2, 'b');
Reminder: INSERT is only good for testing. It is highly
inefficient to be using for production.
Another alternative to getting data into Parquet
tables is using the LOAD operation. There are a few best practices for using LOAD.
It is essential that you tune the num.map.tasks parameter of LOAD. Set the value to at least
the number of data nodes in your cluster. This allows at least one map task to run for
each data node in the cluster. Setting it to multiples of the number of data nodes can
further improve performance. Parquet block size is 1GB, so try to have

the Parquet file size close to 1GB increments to run at optimal performance.
Loading from HDFS has better performance, so when you can, load the data files into
HDFS first before performing the LOAD operation. You can use the file option of the Big
SQL load in order to perform data conversion from text to Parquet.
A few notes on Parquet and compression. By
default, Parquet files are compressed with SNAPPY. You can specify other compression
algorithm such as GZIP or UNCOMPRESSED. Set the HADOOP PROPERTY parquet.compression
flag
to the compression algorithm of choice. It is also valid to populate the same table
with multiple algorithms. Just set the property before each insert or load operation.
Optimized Row Columnar (ORC) is a new storage
format for Big SQL. It has a columnar storage format which stores a collection of rows in
one file and within the collection the row data is stored in a columnar format. This
allows parallel processing of row collections in a cluster. ORC keeps track of per block
statistics. Pros:
- ORC files allow for effective retrieval of individual columns
- Efficient compression. Cons:
- ORC is only supported by the Java I/O engine - Not good for any data exchange outside of
Hadoop. - Big SQL cannot exploit some of the advance
ORC features today. Those unsupported advanced features are predicate pushdown, to skip
unneeded
values, and block elimination via predicate pushdown. Both are supported in Hive 0.13.
ORC defines compression at DDL time. You specify
it in the TBLPROPERTIES clause. The available compression schemes are NONE, ZLIB, or
SNAPPY.
ORC is replacing the Record Columnar (RC)
format. Essentially the same columnar file format with efficient compression and value
encoding. Pros:
- Supported by the native I/O engine. - Efficiently retrieve individual columns
- Efficient compression Cons:
- Not good for data exchange outside of Hadoop. There is a backward compatibility for RC with
ORC. The following compression algorithms are supported:
snappy, gzip, deflate, bzip2.
Avro is an open source, standardized binary
data serialization format. It is a well-defined schema for communicating the structure of
the data. There are code generators available for reading and writing data from various
languages. It is also a well-accepted data interchange format.
Pros: It is supported by the native I/O engine.
It is a popular data exchange format. The table structure can be inferred from
existing schema. It has decent performance.
Cons: Not as efficient as Parquet or ORC.
Not human readable.
There are two ways to create an Avro table. The first way shown here is creating an Avro
table with an in-line schema. You need to specify the Avro SerDE as well as the INPUTFORMAT
and the OUTPUTFORMAT. You also need to specify TBLPROPERTIES to define the table.
Note that the column definitions can be inferred from the schema. Big SQL infers the schema
at table creation time. Hive infers it every time the table is read. If you want to change
the schema in Big SQL, you must drop and recreate it.
The second method is to create a table using
a file. You will still need to specify the SerDe and the input and output format.
When no column list is specified, then the
table columns will be inferred as shown in the table.

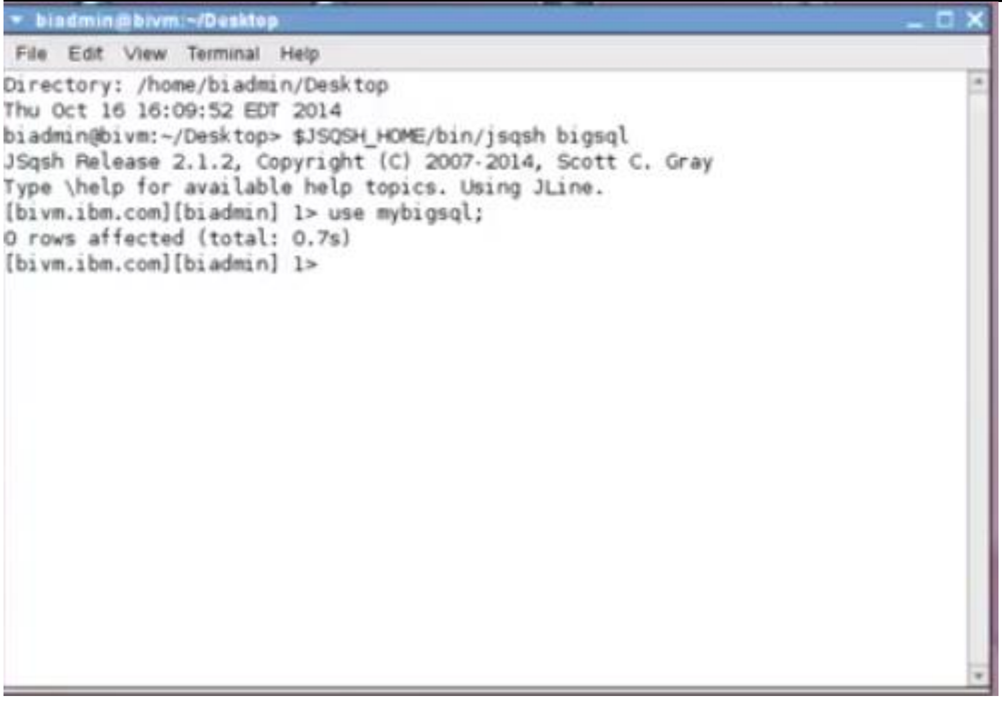
This can lead to a particular case where your table ends up using a STRING, which is essentially the maximum VARCHAR with 32K size. As stated before, this can be bad for performance so you should either use the bigsql.string.size to lower the default size or explicitly declare your columns.
Having completed this lesson, you should now be able to list and explain the Big SQL data types. Create Big SQL schemas. Create Big SQL tables. Understand and use the file formats supported by Big SQL.
Congratulations, you have completed this course.


with Big SQL data - Lab solution - Part 1
Ver más ta
File Edit View History Bookmarks Tools Help
IBM InfoSphere BigInsights Quickst...
bivm:8080/login.jsp
Google
IBM® InfoSphere® BigInsights™ Quickstart Edition
Please enter your information
User name:
biadmin
Password:
••••
Login
Cancel
Licensed Materials - Property of IBM Corp. © Copyright 2010, 2014. IBM Corporation. IBM, InfoSphere and BigInsights are trademarks of IBM Corporation, registered in many jurisdictions worldwide.

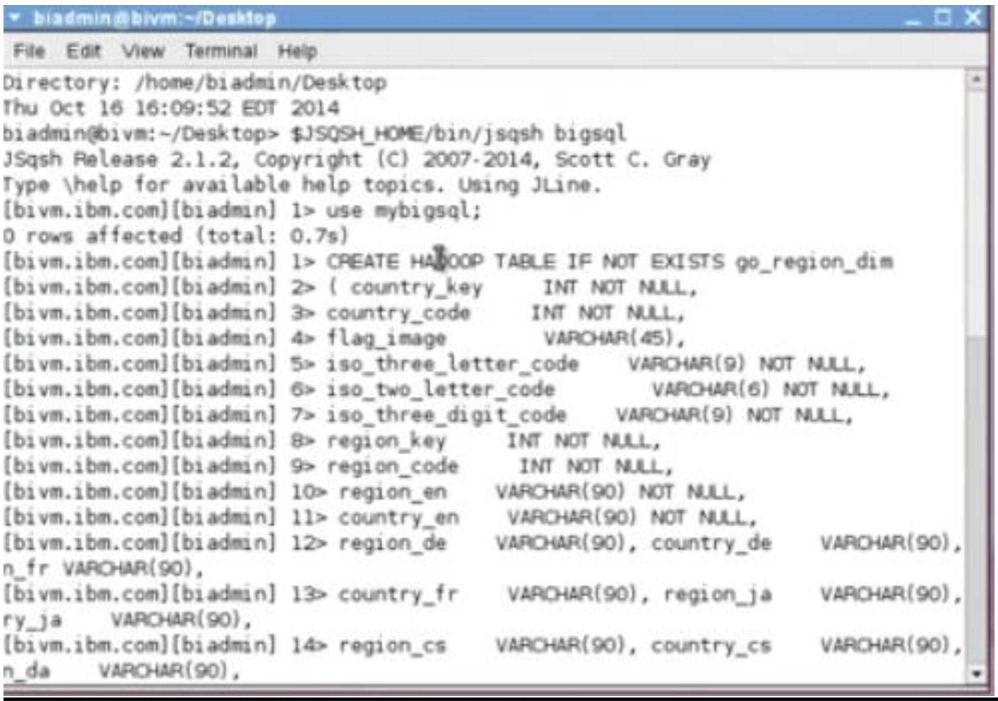

with Big SQL data - Lab solution - Part 1
IBM InfoSphere BigInsights Quick Start Edition (for Non-Production Environment)
Welcome | Dashboard | Cluster Status | Files | Applications | Application Status | BigSheets
Monitor Services | Manage Alerts | Log Settings | Backup and Restore HBase
Add nodes | Add services | Remove services | Remove nodes

Nodes	1
Map/Reduce	Running
HDFS	Running
Alert	Running
Big SQL	Running
Catalog	Running
HBase	Running
Hive	Running
HttpFS	Running
Monitoring	Unavailable
Oozie	Running
Zookeeper	Running

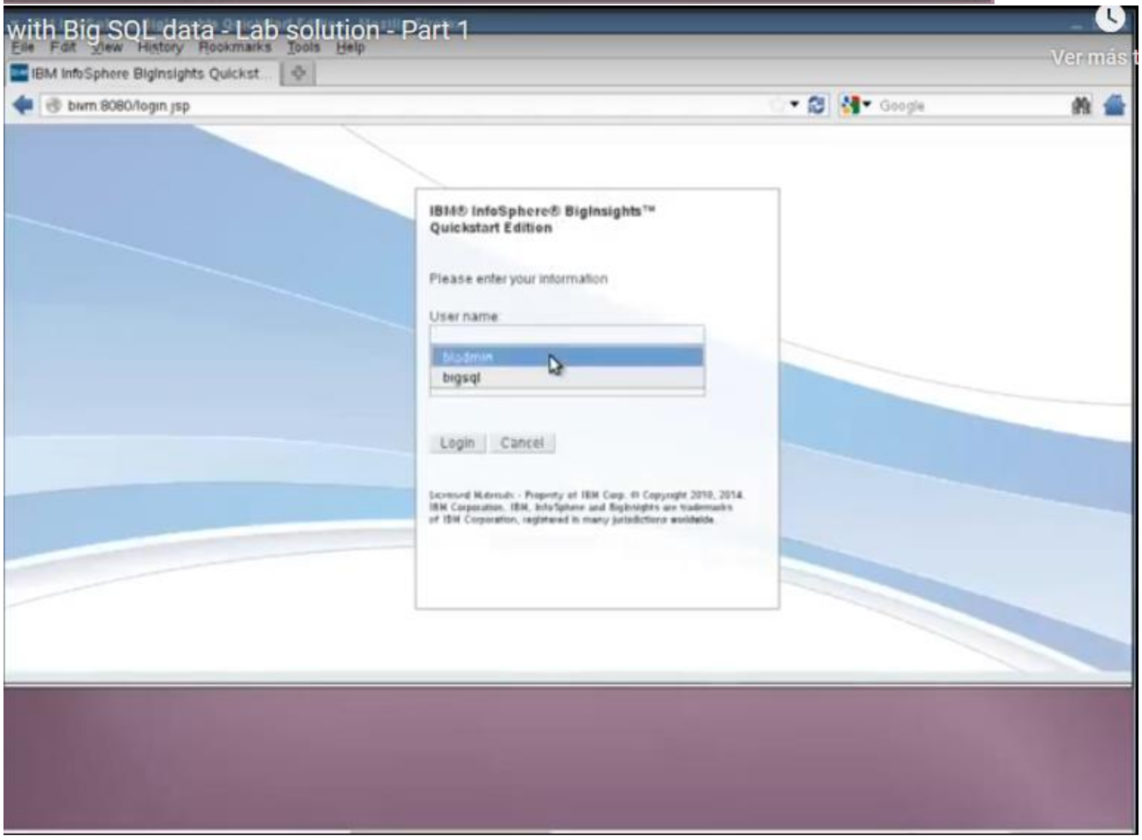
Host	Status	Roles
bivm.ibm.com	Host is running	namenode, monitoring, scheduler, catalog, t



```
Directory: /home/biadmin/Desktop
Thu Oct 16 16:09:52 EDT 2014
biadmin@bivm:~/Desktop> $JSQSH_HOME/bin/jsqsh bigsql
JSqsh Release 2.1.2, Copyright (C) 2007-2014, Scott C. Gray
Type \help for available help topics. Using JLine.
[bivm.ibm.com][biadmin] 1> use mybigsql;
0 rows affected (total: 0.7s)
[bivm.ibm.com][biadmin] 1>
```



```
Directory: /home/biadmin/Desktop
Thu Oct 16 16:09:52 EDT 2014
biadmin@bivm:~/Desktop> $JSQSH_HOME/bin/jsqsh bigsql
JSqsh Release 2.1.2, Copyright (C) 2007-2014, Scott C. Gray
Type \help for available help topics. Using JLine.
[bivm.ibm.com][biadmin] 1> use mybigsql;
0 rows affected (total: 0.7s)
[bivm.ibm.com][biadmin] 1> CREATE HADOOP TABLE IF NOT EXISTS go_region_dim
[bivm.ibm.com][biadmin] 2> ( country_key      INT NOT NULL,
[bivm.ibm.com][biadmin] 3> country_code      INT NOT NULL,
[bivm.ibm.com][biadmin] 4> flag_image         VARCHAR(45),
[bivm.ibm.com][biadmin] 5> iso_three_letter_code    VARCHAR(9) NOT NULL,
[bivm.ibm.com][biadmin] 6> iso_two_letter_code       VARCHAR(6) NOT NULL,
[bivm.ibm.com][biadmin] 7> iso_three_digit_code    VARCHAR(9) NOT NULL,
[bivm.ibm.com][biadmin] 8> region_key      INT NOT NULL,
[bivm.ibm.com][biadmin] 9> region_code      INT NOT NULL,
[bivm.ibm.com][biadmin] 10> region_en     VARCHAR(90) NOT NULL,
[bivm.ibm.com][biadmin] 11> country_en     VARCHAR(90) NOT NULL,
[bivm.ibm.com][biadmin] 12> region_de     VARCHAR(90), country_de     VARCHAR(90),
n_fr VARCHAR(90),
[bivm.ibm.com][biadmin] 13> country_fr     VARCHAR(90), region_ja     VARCHAR(90),
ry_ja     VARCHAR(90),
[bivm.ibm.com][biadmin] 14> region_cs     VARCHAR(90), country_cs     VARCHAR(90),
n_da     VARCHAR(90),
```


with Big SQL data - Lab solution - Part 1
File Edit View History Bookmarks Tools Help
IBM InfoSphere BigInsights Quickst...
bivm:8080/login.jsp
Google
Ver más

IBM® InfoSphere® BigInsights™
Quickstart Edition

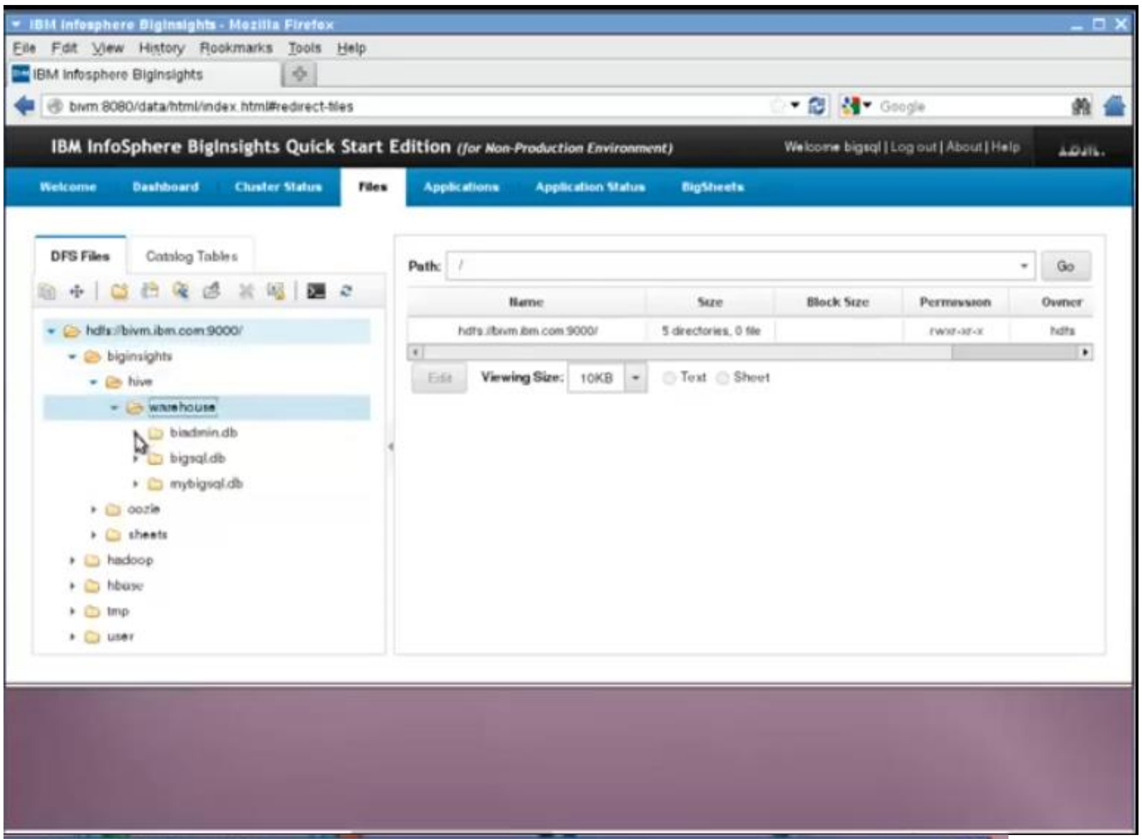
Please enter your information

User name

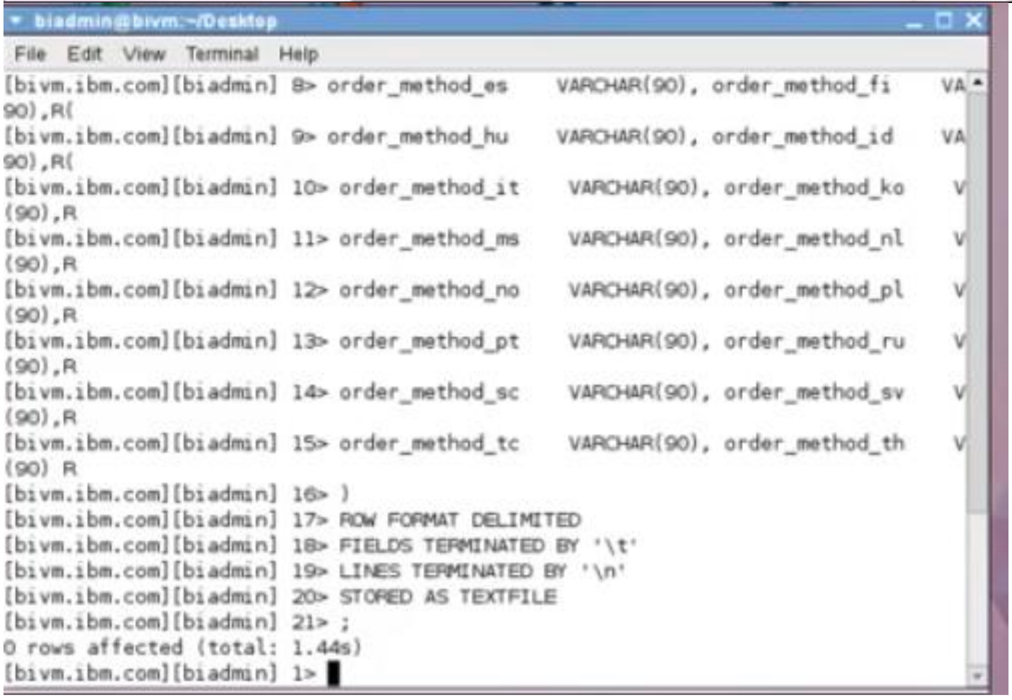
biadmin
bigsql

Login Cancel

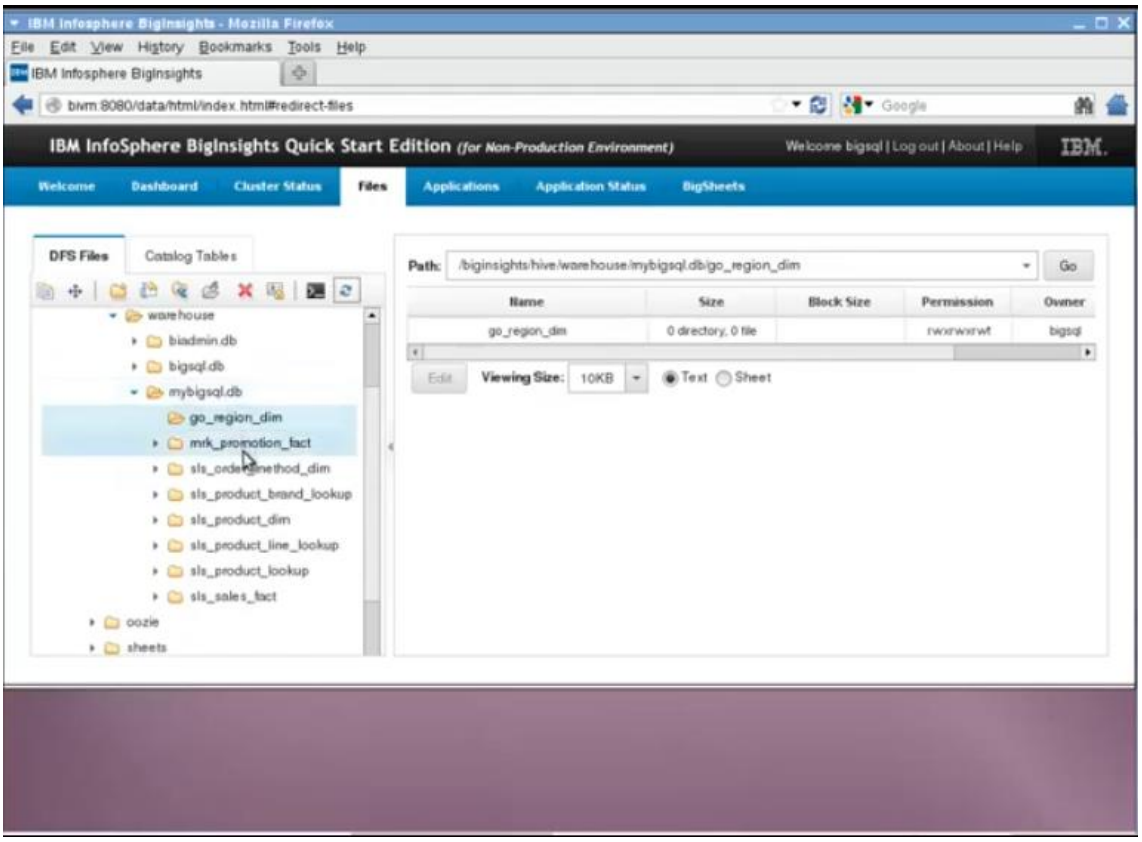
Licensed Materials - Property of IBM Corp. © Copyright 2010, 2014 IBM Corporation. IBM, InfoSphere and BigInsights are trademarks of IBM Corporation, registered in many jurisdictions worldwide.


IBM InfoSphere BigInsights - Mozilla Firefox
File Edit View History Bookmarks Tools Help
IBM InfoSphere BigInsights
bivm:8080/data/html/index.html#redirect-files
IBM InfoSphere BigInsights Quick Start Edition (for Non-Production Environment)
Welcome bigsql | Log out | About | Help
Welcome
Dashboard
Cluster Status
Files
Applications
Application Status
BigSheets
DFS Files
Catalog Tables
hdfs://bivm.ibm.com:9000/
biginsights
hive
warehouse
biadmin.db
bigsql.db
mybigsql.db
oozie
sheets
hadoop
hbase
tmp
user
Path: /
Go
Edit
Viewing Size: 10KB
Text
Sheet

Name	Size	Block Size	Permission	Owner
hdfs://bivm.ibm.com:9000/	5 directories, 0 file		rwxr-xr-x	hdfs



```
[bivm.ibm.com][biadmin] 8> order_method_es    VARCHAR(90), order_method_fi    VA
90),R(
[bivm.ibm.com][biadmin] 9> order_method_hu    VARCHAR(90), order_method_id    VA
90),R(
[bivm.ibm.com][biadmin] 10> order_method_it    VARCHAR(90), order_method_ko    V
(90),R
[bivm.ibm.com][biadmin] 11> order_method_ms    VARCHAR(90), order_method_nl    V
(90),R
[bivm.ibm.com][biadmin] 12> order_method_no    VARCHAR(90), order_method_pl    V
(90),R
[bivm.ibm.com][biadmin] 13> order_method_pt    VARCHAR(90), order_method_ru    V
(90),R
[bivm.ibm.com][biadmin] 14> order_method_sc    VARCHAR(90), order_method_sv    V
(90),R
[bivm.ibm.com][biadmin] 15> order_method_tc    VARCHAR(90), order_method_th    V
(90) R
[bivm.ibm.com][biadmin] 16> )
[bivm.ibm.com][biadmin] 17> ROW FORMAT DELIMITED
[bivm.ibm.com][biadmin] 18> FIELDS TERMINATED BY '\t'
[bivm.ibm.com][biadmin] 19> LINES TERMINATED BY '\n'
[bivm.ibm.com][biadmin] 20> STORED AS TEXTFILE
[bivm.ibm.com][biadmin] 21> ;
0 rows affected (total: 1.44s)
[bivm.ibm.com][biadmin] 1>
```


IBM InfoSphere BigInsights Quick Start Edition (for Non-Production Environment)
Welcome Dashboard Cluster Status Files Applications Application Status BigSheets
DFS Files Catalog Tables
warehouse
biadmin.db
bigsql.db
mybigsql.db
go_region_dim
mrk_promotion_fact
sls_order_method_dim
sls_product_brand_lookup
sls_product_dim
sls_product_line_lookup
sls_product_lookup
sls_sales_fact
oozie
sheets
Path: /biginsights/hive/warehouse/mybigsql.db/go_region_dim
Go
Name Size Block Size Permission Owner
go_region_dim 0 directory, 0 file rwxrwxrwt bigsql
Edit Viewing Size: 10KB Text Sheet

In this exercise, you will get to create Big SQL schemas and tables and then work with operations to load data into those tables. You will also write SQL queries to get the data from tables and views. The exercise will also cover some additional load operations as well show you how to create and use partitioned tables.
Since we are starting up a new exercise, let's make sure that BigInsights is still up and running. The thing is, with these VM, sometimes the instances does not stay on so I like to check to make sure they are still running. Log in to the Web Console and check the cluster status to make sure everything is still running. If not, start up BigInsights before continuing on with the lab.
Start up a new terminal.
Start JSqsh and connect to the bigsql connection.
Create a schema that you will use for this
lab. The use mybigsql command will set the default schema as mybigsql. It will also create that schema for you if it does not exist.
The examples that you will see in this lab
exercise assumes that your sample data is on the VMWare image. You will use sales data from a fictional company that sells and distributes outdoor products to third-party retailers as well as direct to consumers online. There are a series of FACT and DIMENSION tables. You will use few than 10 of these in the GOSALESDW database in this lab exercise.
You will be creating these tables in Big SQL. In the JSqsh shell, copy and paste the table definition to create the dimension table for the region info.
Notice the Hadoop keyword. You need to specify this keyword in order to create tables for the Hadoop environment. However, you can omit this if you enable the SYSHADOOP.COMPATIBILITY_MODE.
Also, notice that you did not explicitly specify the table's schema. The table will be created in the default schema, which we have specified as mybigsql.
The table's data will be row format delimited and have its fields terminated by tabs (\t) and the lines terminated by new line (\n). The data will be stored as TEXTFILE format,

making it easy for a wide range of applications to work with.
Launch the BigInsights console. Log in as bigsql with the password bigsql.
Go to the Files tab to check out the table definition.
Drill down to biginsights --> hive --> warehouse --> mybigsql.db to see the go_region_dim table that you created.
Go back to the JSqsh shell and copy and paste
the next query to create the tracking method of the order of sale.
Go through and copy and paste the remaining queries for the tables to create them in Big SQL. The video will fast forward to once all eight tables have been created
Once all eight tables
have been created, go to the web console to see their directories. You may need to refresh the view to see the updates. Notice that the directories are empty and that is because
we have only created the table and have not loaded any data into them yet. You will see how to do this in the next video.