Course Guide

IBM BigInsights Big SQL v4

Course code DW633 ERC 1.0

IBM Training

September 2015

NOTICES

This information was developed for products and services offered in the USA.

IBM may not offer the products, services, or features discussed in this document in other countries. Consult your local IBM representative for information on the products and services currently available in your area. Any reference to an IBM product, program, or service is not intended to state or imply that only that IBM product, program, or service may be used. Any functionally equivalent product, program, or service that does not infringe any IBM intellectual property right may be used instead. However, it is the user's responsibility to evaluate and verify the operation of any non-IBM product, program, or service. IBM may have patents or pending patent applications covering subject matter described in this document. The furnishing of this document does not grant you any license to these patents. You can send license inquiries, in writing, to:

IBM Director of Licensing
IBM Corporation
North Castle Drive, MD-NC119
Armonk, NY 10504-1785
United States of America

The following paragraph does not apply to the United Kingdom or any other country where such provisions are inconsistent with local law: INTERNATIONAL BUSINESS MACHINES CORPORATION PROVIDES THIS PUBLICATION "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF NON-INFRINGEMENT, MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. Some states do not allow disclaimer of express or implied warranties in certain transactions, therefore, this statement may not apply to you.

This information could include technical inaccuracies or typographical errors. Changes are periodically made to the information herein; these changes will be incorporated in new editions of the publication. IBM may make improvements and/or changes in the product(s) and/or the program(s) described in this publication at any time without notice.

Any references in this information to non-IBM websites are provided for convenience only and do not in any manner serve as an endorsement of those websites. The materials at those websites are not part of the materials for this IBM product and use of those websites is at your own risk.

IBM may use or distribute any of the information you supply in any way it believes appropriate without incurring any obligation to you. Information concerning non-IBM products was obtained from the suppliers of those products, their published announcements or other publicly available sources. IBM has not tested those products and cannot confirm the accuracy of performance, compatibility or any other claims related to non-IBM products. Questions on the capabilities of non-IBM products should be addressed to the suppliers of those products.

This information contains examples of data and reports used in daily business operations. To illustrate them as completely as possible, the examples include the names of individuals, companies, brands, and products. All of these names are fictitious and any similarity to the names and addresses used by an actual business enterprise is entirely coincidental.

TRADEMARKS

IBM, the IBM logo, ibm.com and BigInsights are trademarks or registered trademarks of International Business Machines Corp., registered in many jurisdictions worldwide. Other product and service names might be trademarks of IBM or other companies. A current list of IBM trademarks is available on the web at “Copyright and trademark information” at www.ibm.com/legal/copytrade.shtml.

Java and all Java-based trademarks and logos are trademarks or registered trademarks of Oracle and/or its affiliates.

Linux is a registered trademark of Linus Torvalds in the United States, other countries, or both.

Microsoft, Windows, and the Windows logo are trademarks of Microsoft Corporation in the United States, other countries, or both.

UNIX is a registered trademark of The Open Group in the United States and other countries.

Contents

Course overview

Preface overview

This course is designed to introduce the student to the capabilities of Big SQL. Big SQL is part of IBM BigInsights that allows you to access your HDFS data by providing a logical view to it. You can use the same SQL that was developed for your data warehouse data on your HDFS data. This course will provide some context on why you would use Big SQL followed by how to use Big SQL to access your data. It will also cover Big SQL federation allowing you to join various data sources with Big SQL. Big SQL also integrates with a number of other components including, Spark, HBase and BigSheets.

Intended audience

The course is designed for developers and administrators that want to use Big SQL to access and administer their Hadoop data.

Topics covered

Topics covered in this course include:

- Understand how Big SQL fit in the Hadoop architecture
- Creating Big SQL schemas and tables
- Loading data into Big SQL tables
- List and understand the Big SQL data types
- Querying Big SQL tables
- Understanding Big SQL data access plans
- Controlling data access using column masking and row-based access control
- Using Big SQL statistics to improve query performance
- Setting up and using Big SQL federation
- Using Big SQL operations on tables managed by HBase
- Using Spark operations on data managed by Big SQL

Course prerequisites

Participants should have:

- Students should be familiar with Hadoop and the Linux file system.
- Although not required, it would also be helpful for students to take the DW613 - IBM BigInsights Overview to have a better understanding of how BigSheets fit into everything.
- Student can attend many free courses at www.bigdatauniversity.com to acquire the necessary requirements.

Document conventions

Conventions used in this guide follow Microsoft Windows application standards, where applicable. As well, the following conventions are observed:

- **Bold**: Bold style is used in demonstration and exercise step-by-step solutions to indicate a user interface element that is actively selected or text that must be typed by the participant.
- *Italic*: Used to reference book titles.
- CAPITALIZATION: All file names, table names, column names, and folder names appear in this guide exactly as they appear in the application.
  To keep capitalization consistent with this guide, type text exactly as shown.

Additional training resources

- Visit IBM Analytics Product Training and Certification on the IBM website for details on:
  - Instructor-led training in a classroom or online
  - Self-paced training that fits your needs and schedule
  - Comprehensive curricula and training paths that help you identify the courses that are right for you
  - IBM Analytics Certification program
  - Other resources that will enhance your success with IBM Analytics Software
- For the URL relevant to your training requirements outlined above, bookmark:
  - Information Management portfolio: http://www-01.ibm.com/software/data/education/
  - Predictive and BI/Performance Management/Risk portfolio: http://www-01.ibm.com/software/analytics/training-and-certification/

IBM product help

Help type	When to use	Location
Task-oriented	You are working in the product and you need specific task-oriented help.	*IBM Product* - Help link
Books for Printing (.pdf)	You want to use search engines to find information. You can then print out selected pages, a section, or the whole book. Use Step-by-Step online books (.pdf) if you want to know how to complete a task but prefer to read about it in a book. The Step-by-Step online books contain the same information as the online help, but the method of presentation is different.	Start/Programs/*IBM Product*/Documentation
IBM on the Web	You want to access any of the following:	
	• IBM - Training and Certification	• http://www-01.ibm.com/software/analytics/training-and-certification/
	• Online support	• http://www-947.ibm.com/support/entry/portal/Overview/Software
	• IBM Web site	• http://www.ibm.com

IBM Training

Using Big SQL to access HDFS data

IBM BigInsights v4.0

IBM Training

Unit objectives

- Understand how Big SQL fits in the Hadoop architecture
- Accessing and using Big SQL
- Creating Big SQL schemas and tables

Unit objectives

IBM Training

Big SQL on Hadoop

- Multiple tools available to work with data on Hadoop
- MapReduce is highly scalable, but difficult to use
- Other tools requires specific expertise
- Big SQL has a common and familiar syntax

Using Big SQL to access HDFS data

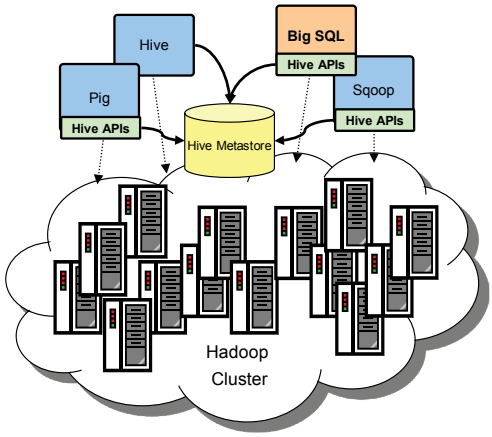
Big SQL on Hadoop

There are a number of tools available to work with data on Hadoop. Tools such as MapReduce, Pig, Hive, and many more. MapReduce, being one of the most common tool is highly scalable, but it is difficult to use. There are a lot of coding required for running batch jobs. There are other tools as well, but those all require a specific set of expertise. What Big SQL offers is a common and familiar query syntax that everybody understands.

IBM Training

What is Big SQL?

- Simply put, it is just a view on your data residing in the Hadoop FileSystem.
- No proprietary storage format
- Modern SQL:2011 capabilities
- Same SQL can be used on your warehouse data with little or no modifications



Using Big SQL to access HDFS data

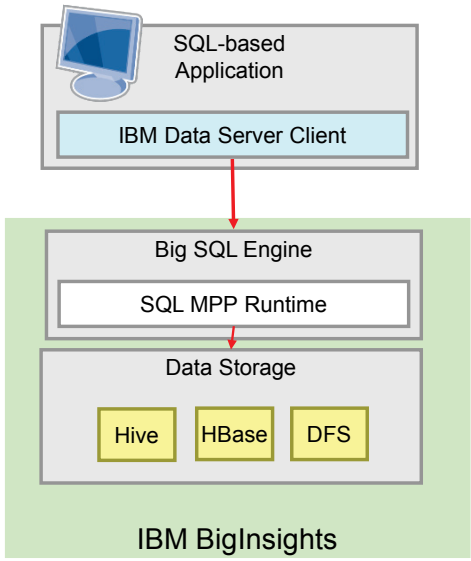
What is Big SQL?

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 can use your existing queries.

IBM Training

Architected for performance

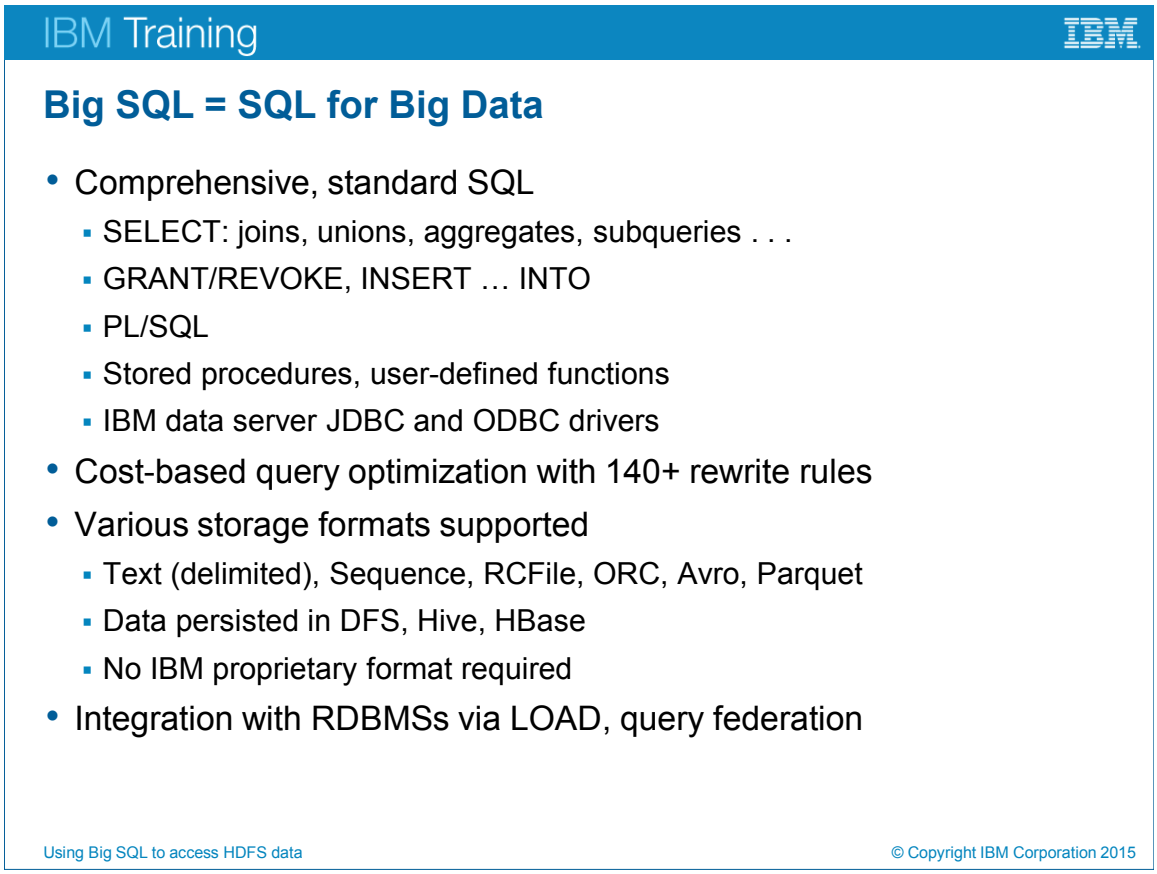
- MapReduce is replaced with a modern, massively parallel processing (MPP) architecture
  - Compiler and runtime are native node (not Java)
  - Big SQL workers live directly on the cluster
    - Processing happens locally at the data source
- Operations occur in memory with the ability to spill to disk
  - Supports aggregations and sorts larger than available RAM



Using Big SQL to access HDFS data

Architected for performance

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 available RAM.



Big SQL = SQL for Big Data

Big SQL is designed to provide SQL developers with an easy on-ramp for querying data managed by Hadoop. It enables data administrators to create new tables for data stored in DFS, HBase, or the Hive warehouse. In addition, a LOAD command enables administrators to populate Big SQL tables with data from various sources. And Big SQL's JDBC and ODBC drivers enable many existing tools to use Big SQL to query this distributed data.

Big SQL's runtime execution engine is all native code (C/C++)

For common table formats a native I/O engine is utilized

e.g. text (delimited), RC, SEQ, Parquet, Avro

For all others, a java I/O engine is used

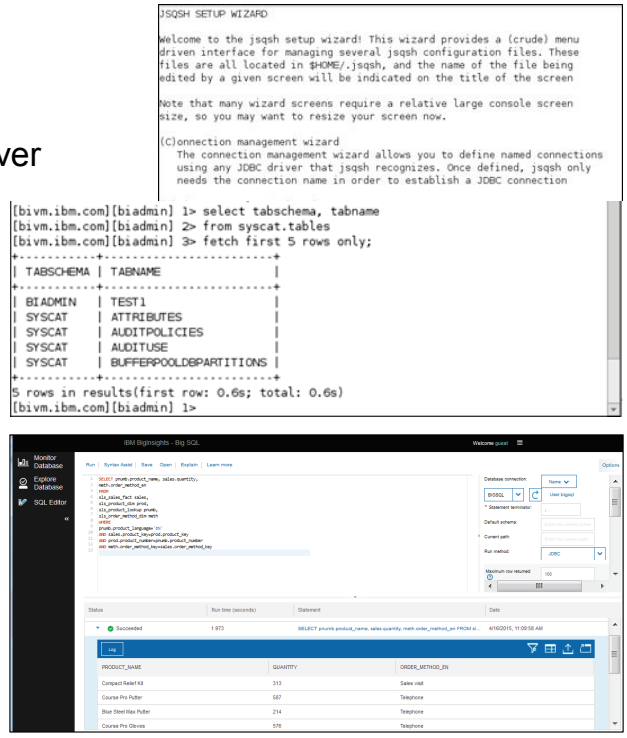
Maximizes compatibility with existing tables

Allows for custom file formats and SerDe's

IBM Training

Accessing Big SQL

- Java SQL Shell (JSqsh)
- Web tooling using Data Server Manager (DMS)
- Tools that support IBM JDBC/ODBC driver



Using Big SQL to access HDFS data

Accessing Big SQL

Big SQL includes a command-line interface called JSqsh. JSqsh (pronounced J-skwish) - short for Java SQshelL (pronounced s-q-shell) - is an open source database query tool featuring much of the functionality provided by a good shell, such as variables, redirection, history, command line editing, and so on. As shown on this chart, it includes built-in help information and a wizard for establishing new database connections.

In addition, when Big SQL is installed, administrators can also install IBM Data Server Manager on the Big SQL Head Node. This Web-based tool includes a SQL editor that runs statements and returns results, as shown here. DSM also includes facilities are also available for monitoring your Big SQL database. For more on DSM, visit http://www-03.ibm.com/software/products/en/ibm-data-server-manager.

Tools that support IBM's JDBC / ODBC driver are also options.

IBM Training

JSqsh (1 of 3)

- Big SQL comes with a CLI pronounced as "jay-skwish" – Java SQL Shell
  - Open source command client
  - Query history and query recall
  - Multiple result set display styles
  - Multiple active sessions
- Started under $JSQSH_HOME/bin/jsqsh
  - where $JSQSH_HOME is the installation path of JSqsh

```
[biadmin@ibmclass bin]$ pwd
/usr/ibmpacks/common-utils/jsqsh/2.14/bin
[biadmin@ibmclass bin]$ ls
jsqsh  jsqsh.bat
[biadmin@ibmclass bin]$
```

Using Big SQL to access HDFS data

JSqsh (1 of 3)

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 have 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.

IBM Training

JSqsh (2 of 3)

- Run the JSqsh connection wizard to supply connection information

```
The following configuration properties are supported by this driver.

    Connection name : bigsql
             Driver : IBM Data Server (DB2, Informix, Big SQL)
           JDBC URL : jdbc:db2://${server}:${port}/${db}

Connection URL Variables
------------------------
1                db : BIGSQL
2              port : 51000
3            server : ibmclass.localdomain
4              user : bigsql
5          password : ********
6       Autoconnect : false
```

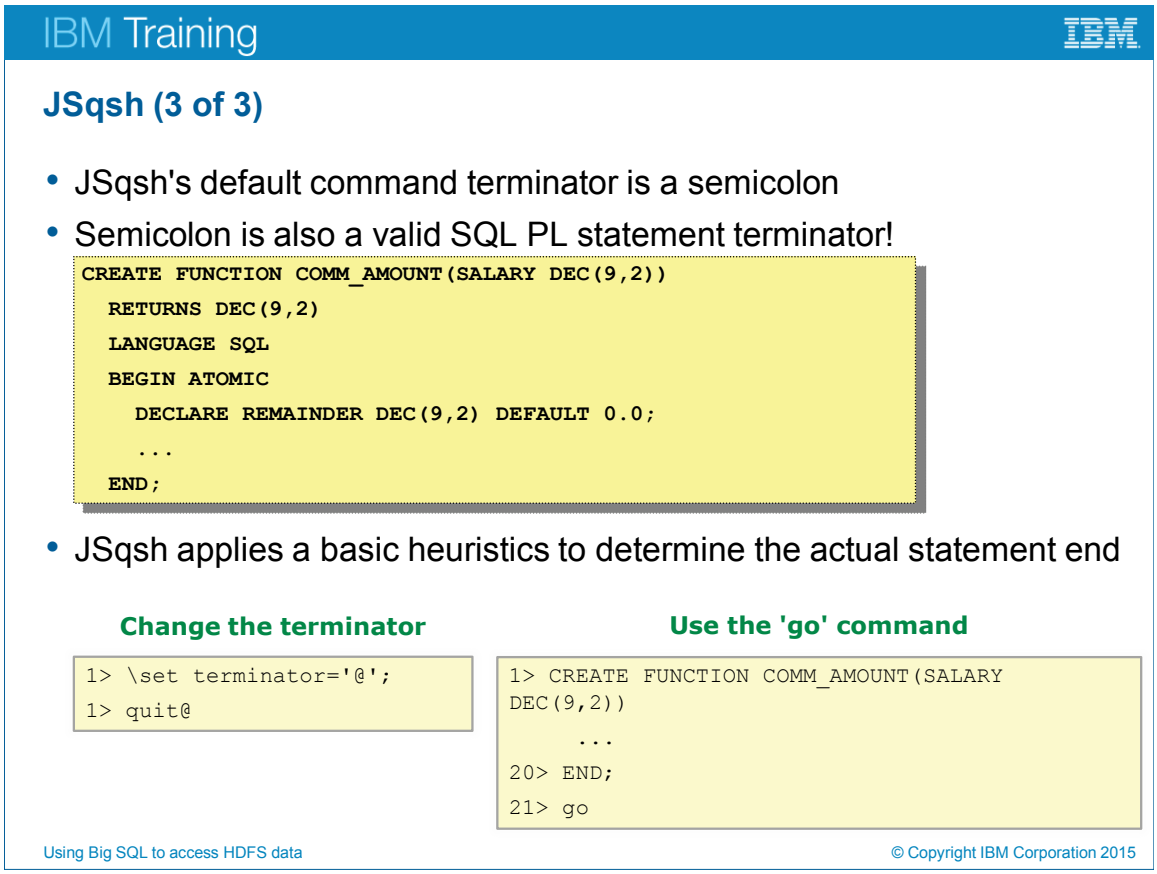
- Connect to the *bigsql* database:
  - `./jsqsh bigsql`

```
[biadmin@ibmclass bin]$ ./jsqsh bigsql
JSqsh Release 2.14, Copyright (C) 2007-2015, Scott C. Gray
Type \help for available help topics. Using JLine.
[ibmclass.localdomain][bigsql] 1>
```

Using Big SQL to access HDFS data

JSqsh (2 of 3)

When you first use jsqsh, it is recommended that you set up the connection using the wizard to supply the information. Once that has been set up, you can connect to the default bigsql database.

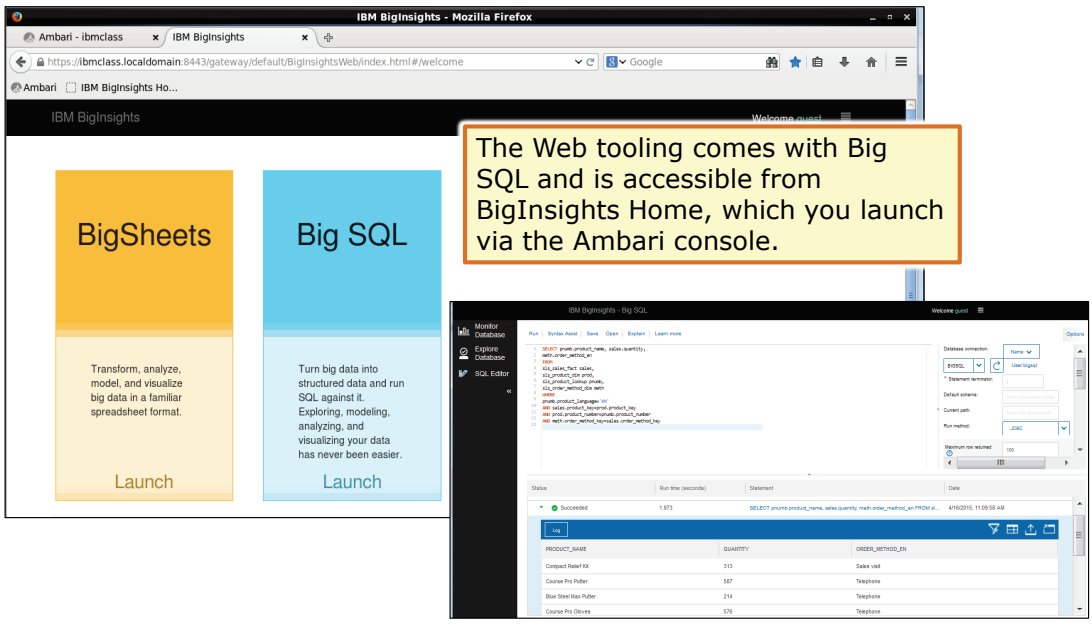


JSqsh (3 of 3)

If you have used JSqsh before, you will know that the default command terminator is a semicolon. With the new Big SQL, 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 @ symbol.

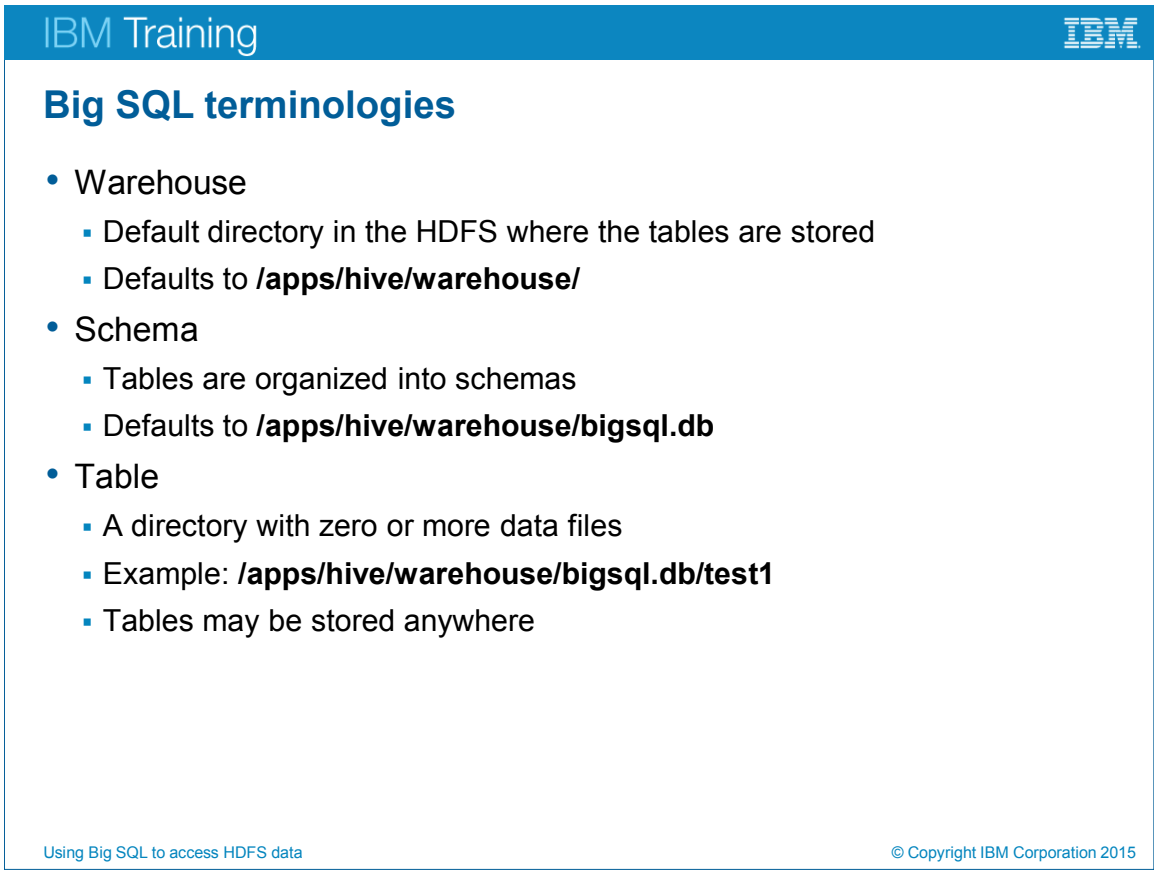
IBM Training

Web tooling using Data Server Manager (DSM)



Web tooling using Data Server Manager (DSM)

A new addition to the v4 release of BigInsights is the web tooling provided by Data Server Manager. You start the web tooling by accessing the BigInsights Home page and then clicking the Big SQL link to open up the web tooling page. From there, you can work with Big SQL.



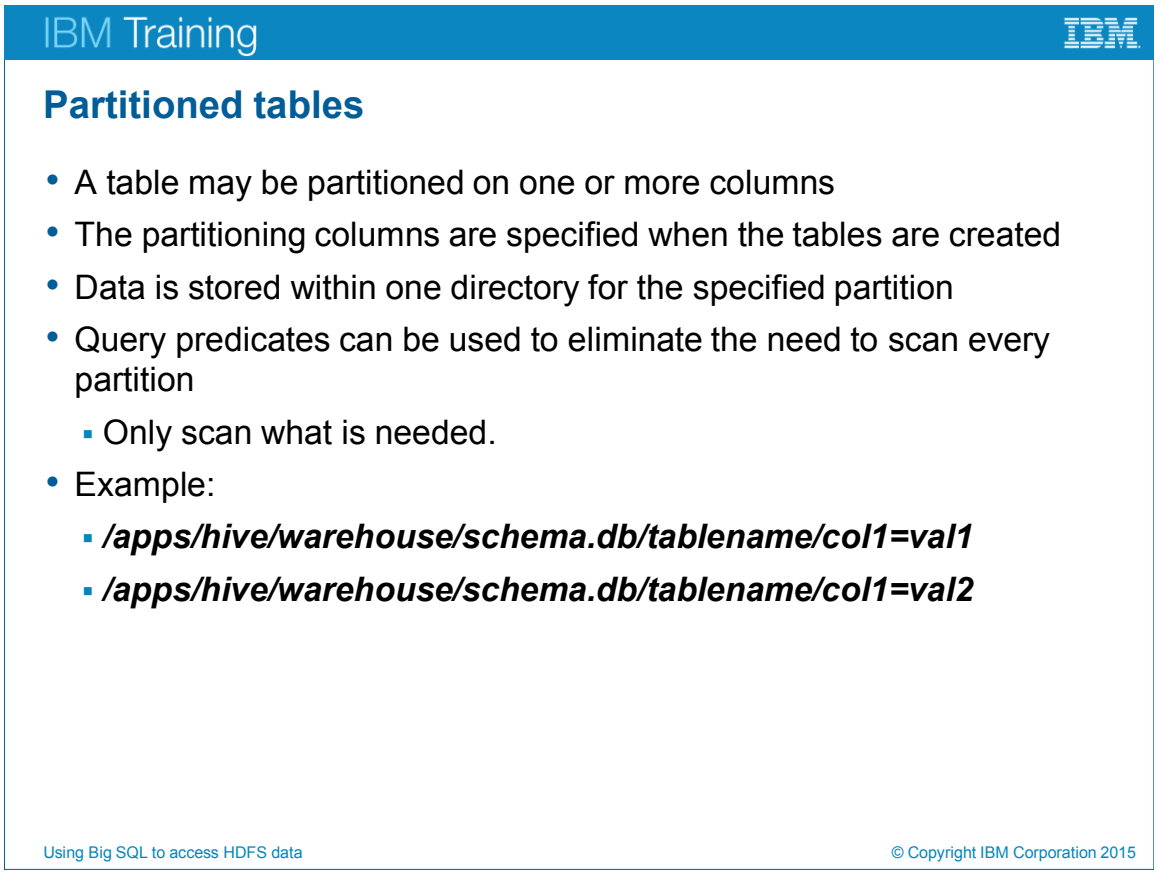
Big SQL terminologies

Here are some Big SQL terminologies.

Warehouse is the default directory in the DFS in which the tables are stored. The default location is */apps/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 */apps/hive/warehouse/bigsql.db*

Table is a directory with zero or more data files. An example of a table directory within DFS is */apps/hive/warehouse/bigsql.db/test1*



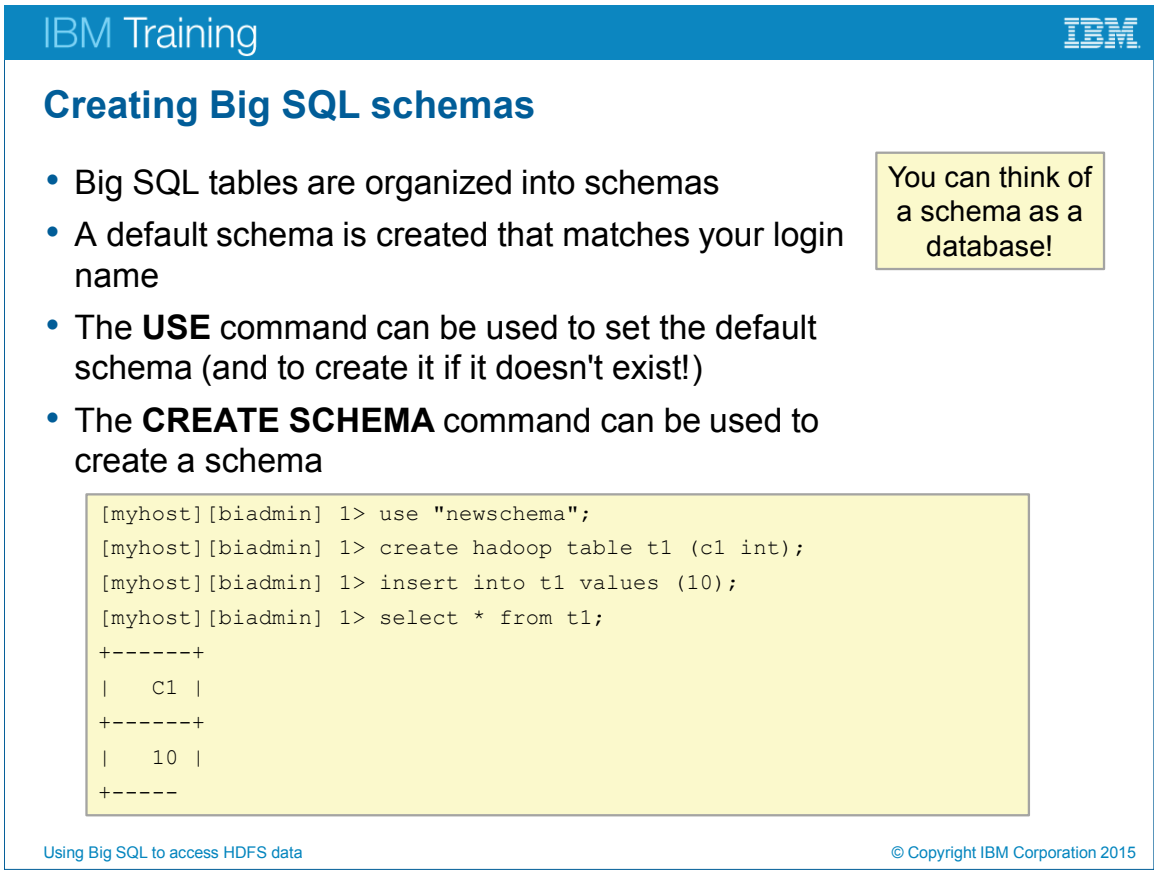
Partitioned tables

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.

/apps/hive/warehouse/schema.db/tablename/col=val1

/apps/hive/warehouse/schema.db/tablename/col=val2

The name of the two directories are, in fact, “col=val1” and “col=val2” for each respective partitions.



Creating Big SQL schemas

Schemas are a way to organize tables. The default schema matches your login name. If you want to specify another default schema, you use the USE command. If you execute the USE statement on a schema that does not currently exist, it gets created. Once the USE command has been invoked, the specified schema becomes the default.

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.

┌ Reminder ──────────────────

When a schema is created, by default, a directory is defined in /apps/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 /apps/hive/warehouse/demo.db.

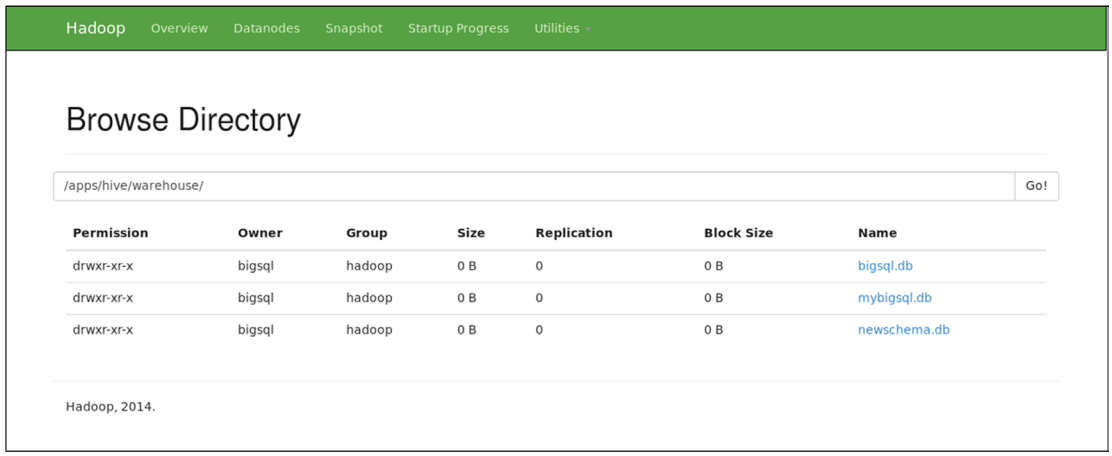
It is possible to define a schema and explicitly place the schema directory elsewhere in HDFS besides the Hive warehouse.

└──────────────────────────

IBM Training

Using the NameNode UI to browse the HDFS

- Access via the HDFS service in Ambari.
  - Quick Links → NameNode UI → Utilities → Browse the file system



Using Big SQL to access HDFS data

Using the NameNode UI to browse the HDFS

You use the **NameNode UI** to browse the HDFS and check out schemas and tables that you create. By default, the directories are located under */apps/hive/warehouse*. You can specify a different directory if you wish.

IBM Training

Creating a Big SQL table

- Standard CREATE TABLE DDL with **extensions**

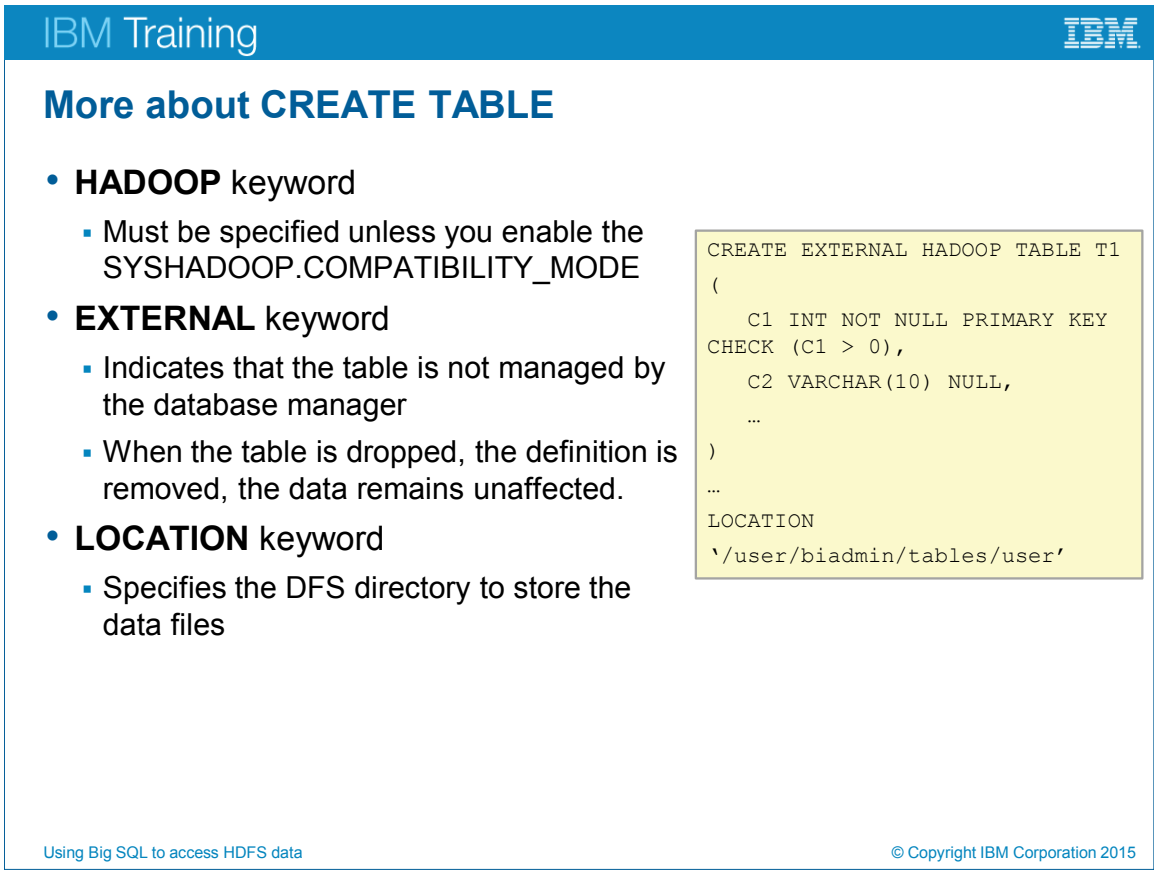
```
create hadoop table users
(
  id        int           not null primary key,
  office_id int           null,
  fname     varchar(30)   not null,
  lname     varchar(30)   not null)
row format delimited
   fields terminated by '|'
stored as textfile;
```

- The "hadoop" keyword creates the table in the HDFS
- Row format delimited and textfile formats are default
- Constraints not enforced (but useful for query optimization)

Using Big SQL to access HDFS data

Creating a Big SQL table

At top, you'll see the syntax for creating a Big SQL table called USERS in the default schema (the user's ID). If you're familiar with SQL, most of this statement will look familiar to you. The items shown in blue are Big SQL DDL extensions for Hadoop. For example, the Hadoop keyword indicates that the data is to be stored on the cluster in the distributed file system – not as a "local" table on the Big SQL head node only. Other clauses specify the storage format of the data – in this case, a row-based text file, with fields delimited by a vertical bar ("|"). Other options & clauses are supported. However, what's important to take away from this example is that it's standard SQL plus some Hadoop-specific extensions.



More about CREATE TABLE

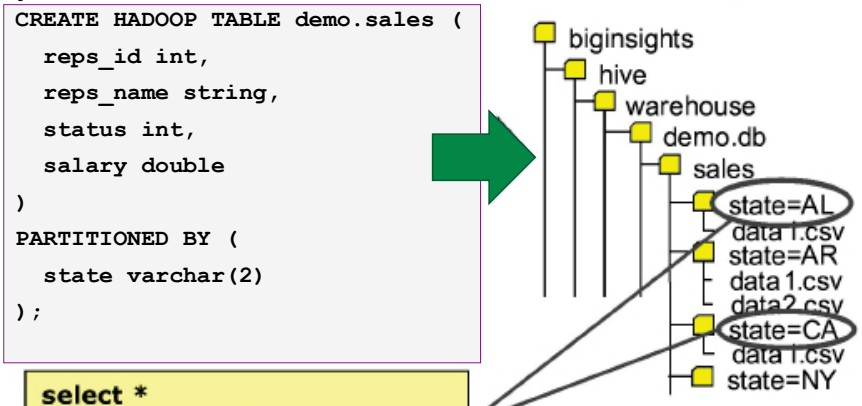
You will need to specify the HADOOP keyword to create a table for the Hadoop environment. You can omit that if you enable the environment variable SYSHADOOP.COMPATIBILITY_MODE. Optionally, you can specify the EXTERNAL keyword to create an external table. This indicates that the table is not managed by 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.

IBM Training

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');
```

Using Big SQL to access HDFS data

CREATE TABLE - partitioned tables

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.

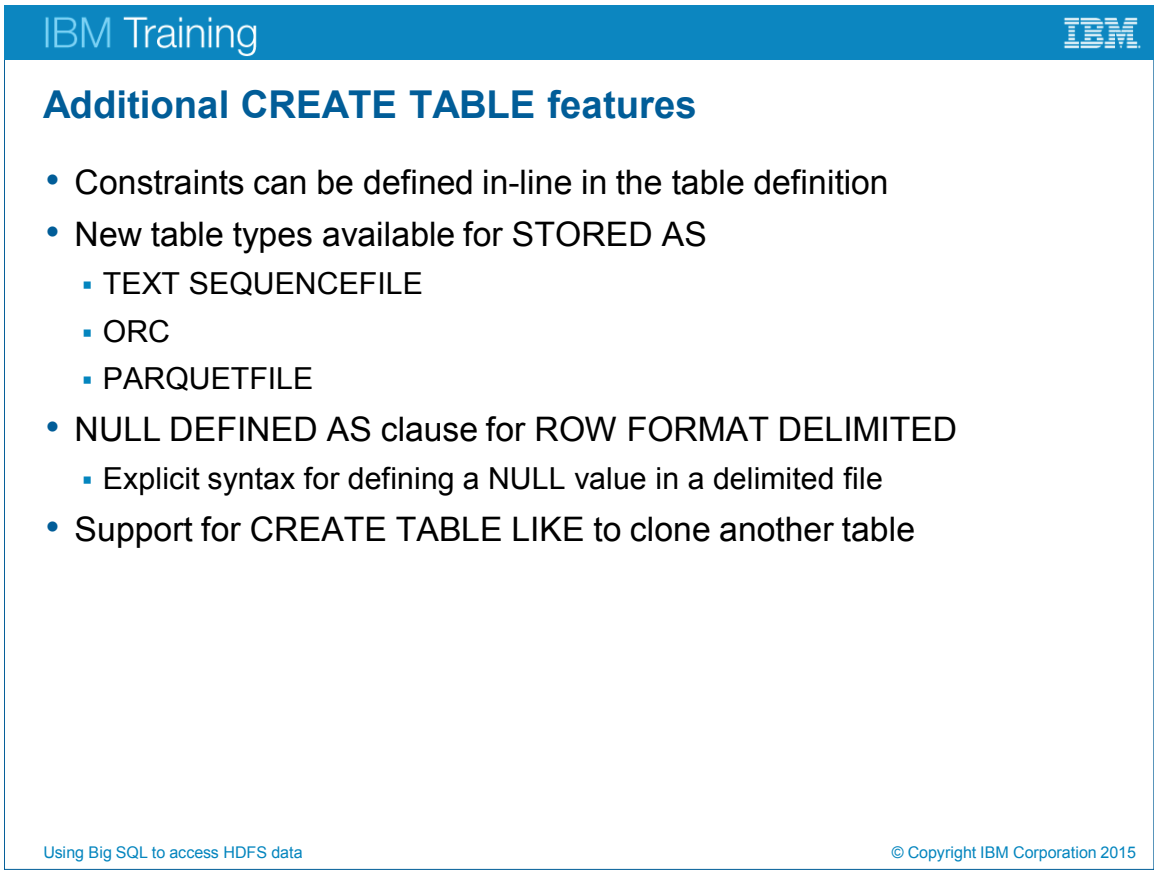
Let us look at the sample code here:

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

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.



Additional CREATE TABLE features

There are some new CREATE TABLE features in Big SQL.

Constraints may now be defined in-line in the table definition.

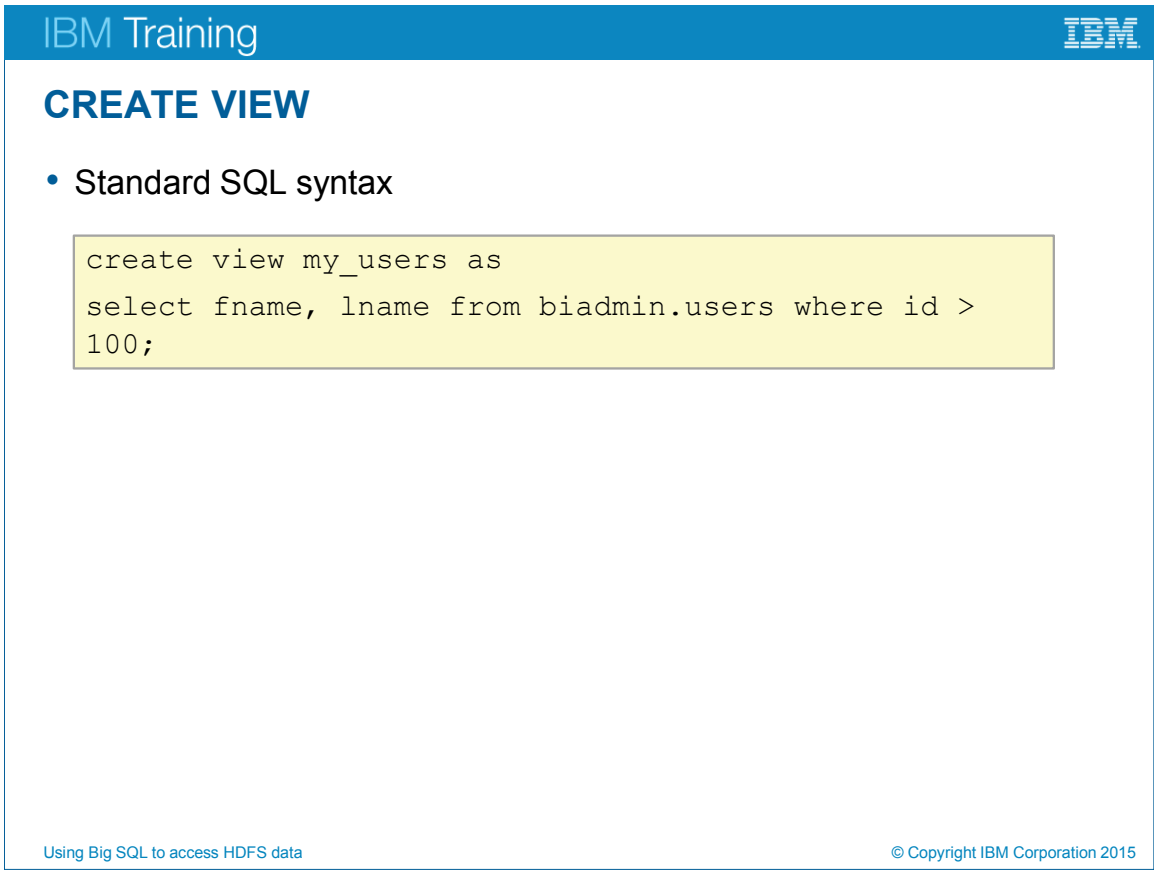
- PRIMARY KEY/UNIQUE
- FOREIGN KEY
- CHECK (new to Big SQL 3.0)
- Constraints are advisory only
- Query optimizer can take advantage

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.

There is support for CREATE TABLE LIKE to clone another table.



CREATE VIEW

You can create Big SQL views just like you'd create a view in a relational DBMS. Here's a simple example.

IBM Training

Checkpoint

1. What is one of the many reasons to use Big SQL?
2. List the two ways you can access and use Big SQL.
3. When creating a Big SQL table, what is the keyword that you need to use to ensure that Big SQL creates the table within the HDFS?

Checkpoint

IBM Training

Checkpoint solutions

1. What is one of the reasons to use Big SQL?
   - Want to access your Hadoop data without using MapReduce
   - Do not want to learn new languages like Hive or JAQL
   - No deep learning curve because Big SQL uses standard 2011 query structure.
2. List the two ways you can work with Big SQL.
   - JSqsh
   - Web tooling from DSM
3. When creating a Big SQL table, what is the keyword that you need to use to ensure that Big SQL creates the table within the HDFS?
   - Hadoop

Checkpoint solutions

IBM Training

Demonstration 1

Connecting to the IBM Big SQL Server

At the end of this demonstration, you will be able to:

- Connect to the IBM Big SQL Server
- Use JSqsh or the Web Tooling to run SQL queries
- Create Big SQL schemas and tables

Demonstration 1: Connecting to the IBM Big SQL Server

Demonstration 1: Connecting to the IBM Big SQL Server

Purpose:

You will see how to connect to the Big SQL Server, a component of IBM BigInsights. In particular, you will connect to the Big SQL server using JSqsh, a CLI for Big SQL. JSqsh, pronounce "jay-skwish" is an open source project for querying JDBC databases, such as Big SQL. In this demonstration, you will see how to get up and running with JSqsh. This demonstration will also show you how to explore the Big SQL service through Apache Ambari, another open source project for cluster manager.

Estimated time: 60 minutes

User/Password: **biadmin/biadmin**

root/dalvm3

Services Password: **ibm2blue**

Task 1. Configure your image.

Important: Occasionally, when you suspend and resume the VM image, the network may assign a different IP address than the one you had configured. In these instances, the Ambari console and the services will not run. You will need to update /etc/hosts file with the newly assigned IP address to continue working with the image. No restart of the VM image is necessary - just give it a couple of minutes, at most. In some cases, you may need to restart the Ambari server, using *ambari-server restart* from the command line.

1. To open a new terminal, right-click the desktop, and then click **Open in Terminal**.
2. Type `ifconfig` to check for the current assigned IP address.
3. Take note of the IP address next to **inet**.

   Next, you need to edit the /etc/hosts file to map the hostname to the IP address.
4. To switch to root user, type `su -` and if prompted for a password, type `dalvm3`.
5. To open the */etc/hosts* file, type `gedit /etc/hosts`.
6. Ensure that the contents of the file are similar to the following:

   ```
   10.0.0.118 ibmclass.localdomain ibmclass
   127.0.0.1 localhost.localdomain localhost
   ```
7. Update with the IP address for ibmclass from step 3.
8. Save and exit the file, and then close the terminal.

Task 2. Start the BigInsights components.

If all of the components have been started already, you may skip this task. Otherwise, follow the steps in this task to start up all of the components.

You will start up all the services via the Ambari console to ensure that everything is ready for the lab. You may stop what you don't need later, but for now, you will start everything.

1. Launch **Firefox**, and then if necessary, navigate to the **Ambari** login page, **http://ibmclass.localdomain:8080**.
2. Log in to the **Ambari** console as **admin/admin**.

   On the left side of the browser are the statuses of all the services. If any are currently yellow, wait for several minutes for them to become red before you start them up.
3. Once all the statuses are red, at the bottom of the left side, click **Actions** and then click **Start All** to start up the services.
4. In the Confirmation dialog, click **OK**.

   This will take a while to complete to complete.
5. When the services have started successfully, click **OK**.

   If your Web browser returns an error instead of a sign in screen, verify that the Ambari server has been started and that you have the correct URL for it. If needed, launch Ambari manually: log into the node containing the Ambari server as `root` and issue this command: `ambari-server start`

Task 3. Start up the Big SQL service.

1. Inside the Ambari console, ensure that **BigInsights - Big SQL** has been started.

   Big SQL requires that the monitoring utility package is started as well.
2. Open a terminal and switch to the root user.

   `su -`

   If prompted for a password: `dalvm3`
3. Change directory to the following path:

   `cd /usr/ibmpacks/bigsql/4.0/dsm/1.1/ibm-datasrvrmgr/bin/`
4. Run the script `/dsmKnoxSetup.sh -knoxHost ibmclass.localdomain`

   ibmclass.localdomain is the host where the Knox gateway is running.

   Remember, whenever you restart the Knox server, you will have to run the dsmKnoxSetup script again.

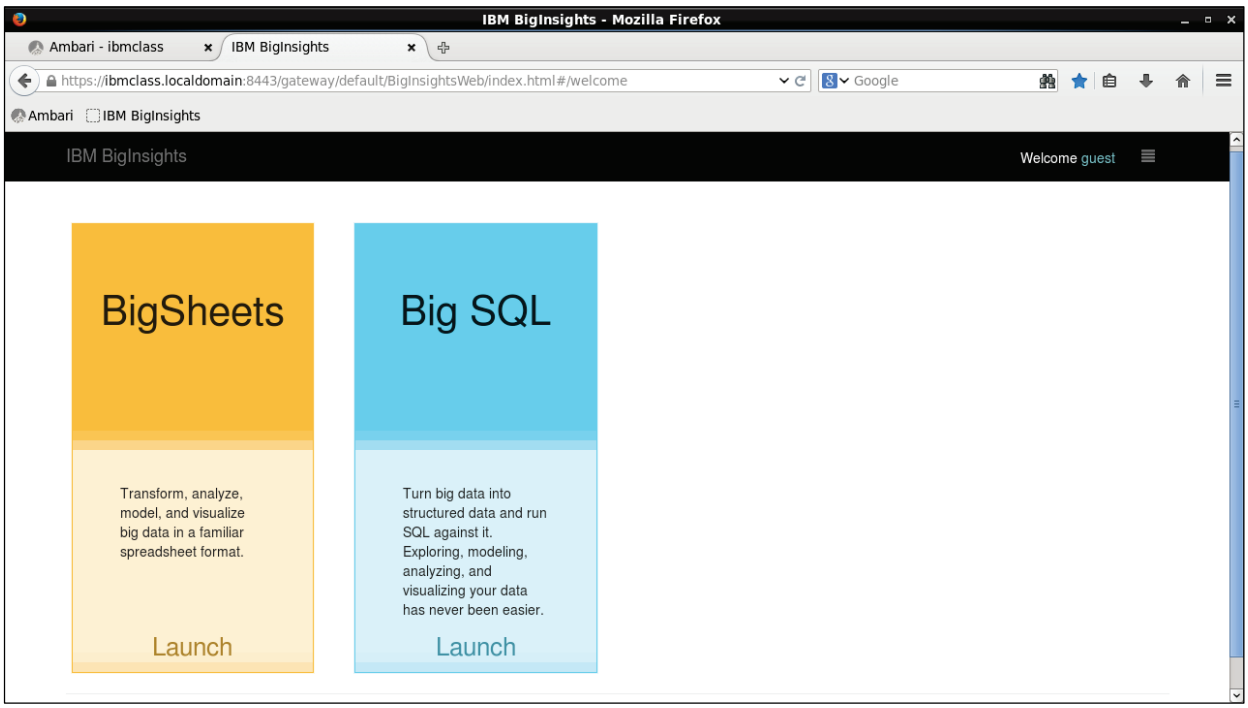
5. If prompted with the message, Do you wish to continue running with the above value (select 1 or 2), select **1**.

   To use Big SQL, you will need to access BigInsights Home. The BigInsights Home requires the LDAP server to be started.
6. Click the **Knox** component.
7. Under the **Service Actions** dropdown on the upper right, select **Start Demo LDAP**.
8. Click **OK** to close the confirmation window.
9. In Firefox, open a new tab, and navigate to the **Web UI (BigInsights Home)** page with the following URL.

   https://ibmclass.localdomain:8443/gateway/default/BigInsightsWeb/index.html
10. If promoted to login, enter **guest / guest-password**.

    It should be saved in the Firefox browser so you can click ok to continue with the login.

    The results appear as follows:



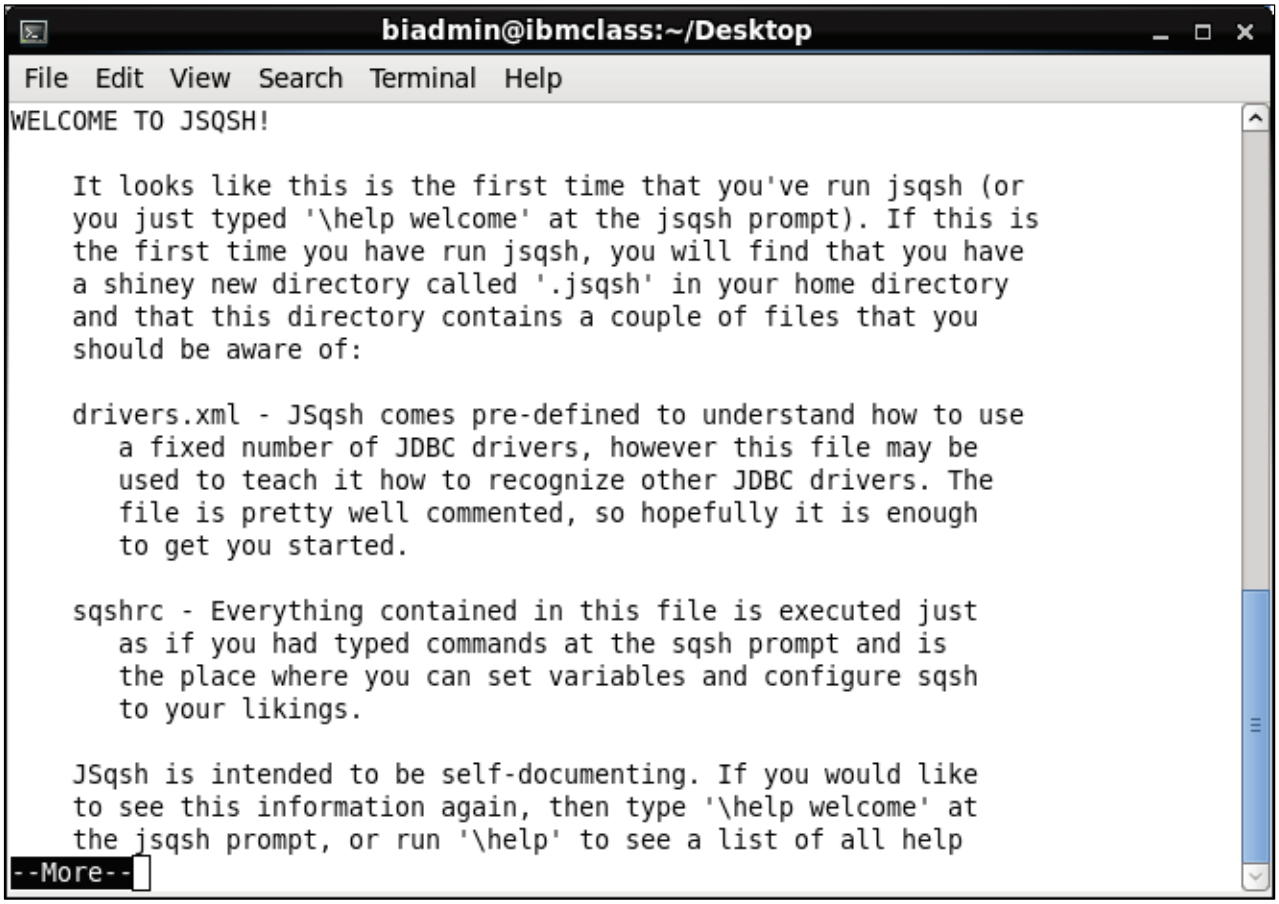
You will now be able to access Big SQL via the **BigInsights Home** page. Check to see that it is available.

Task 4. Connecting to Big SQL using JSqsh.

1. Right-click on the desktop and select **Open in Terminal**.
2. Launch the JSqsh shell:

   `/usr/ibmpacks/common-utils/jsqsh/2.14/bin/jsqsh`

   If it is your first time, you will see a welcome screen for a setup.
3. Press **Enter** to continue and enter **c** to launch the connection wizard.

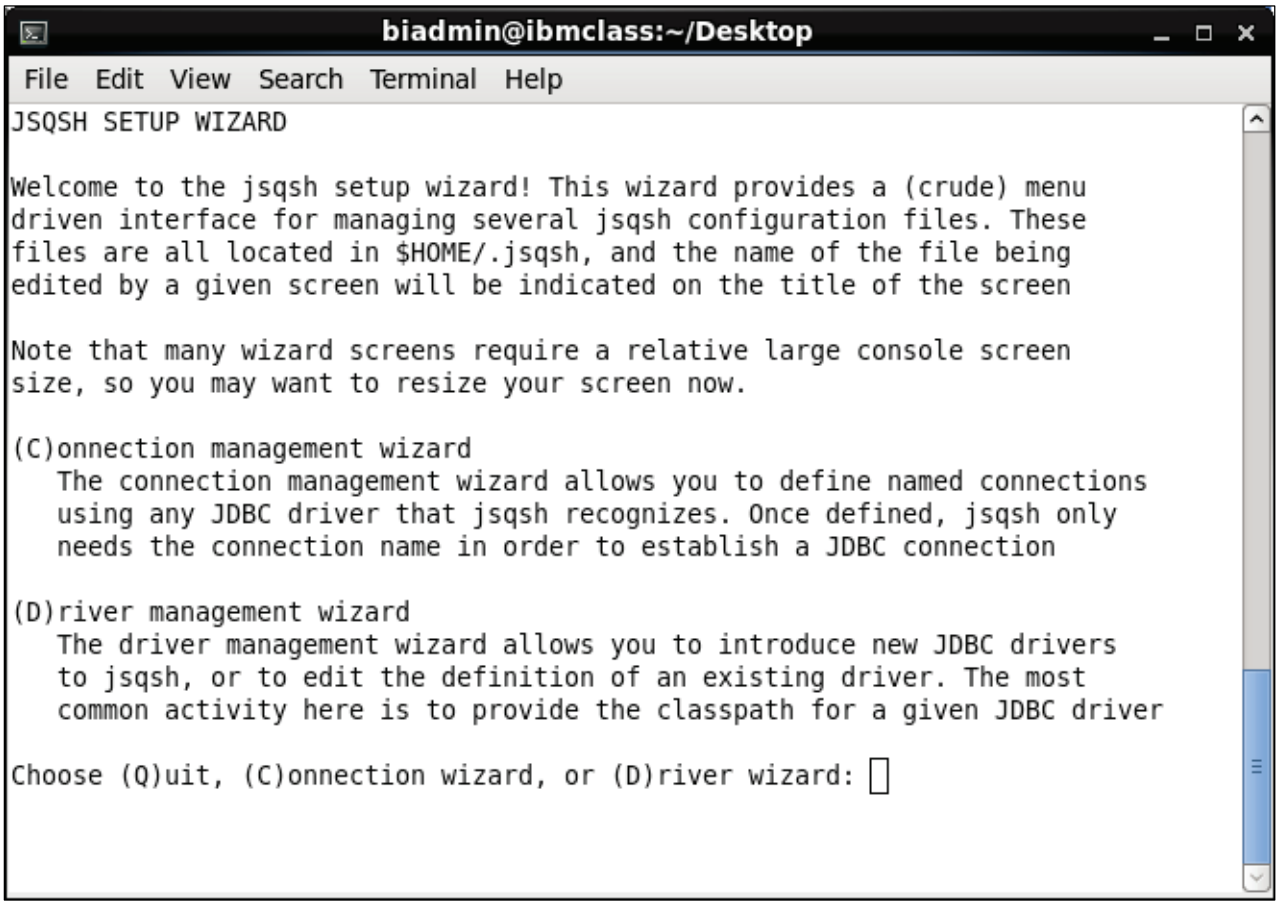


If this is not your first time, you will get a prompt that looks like this:

You want to run the setup to make sure you have the right connection information.

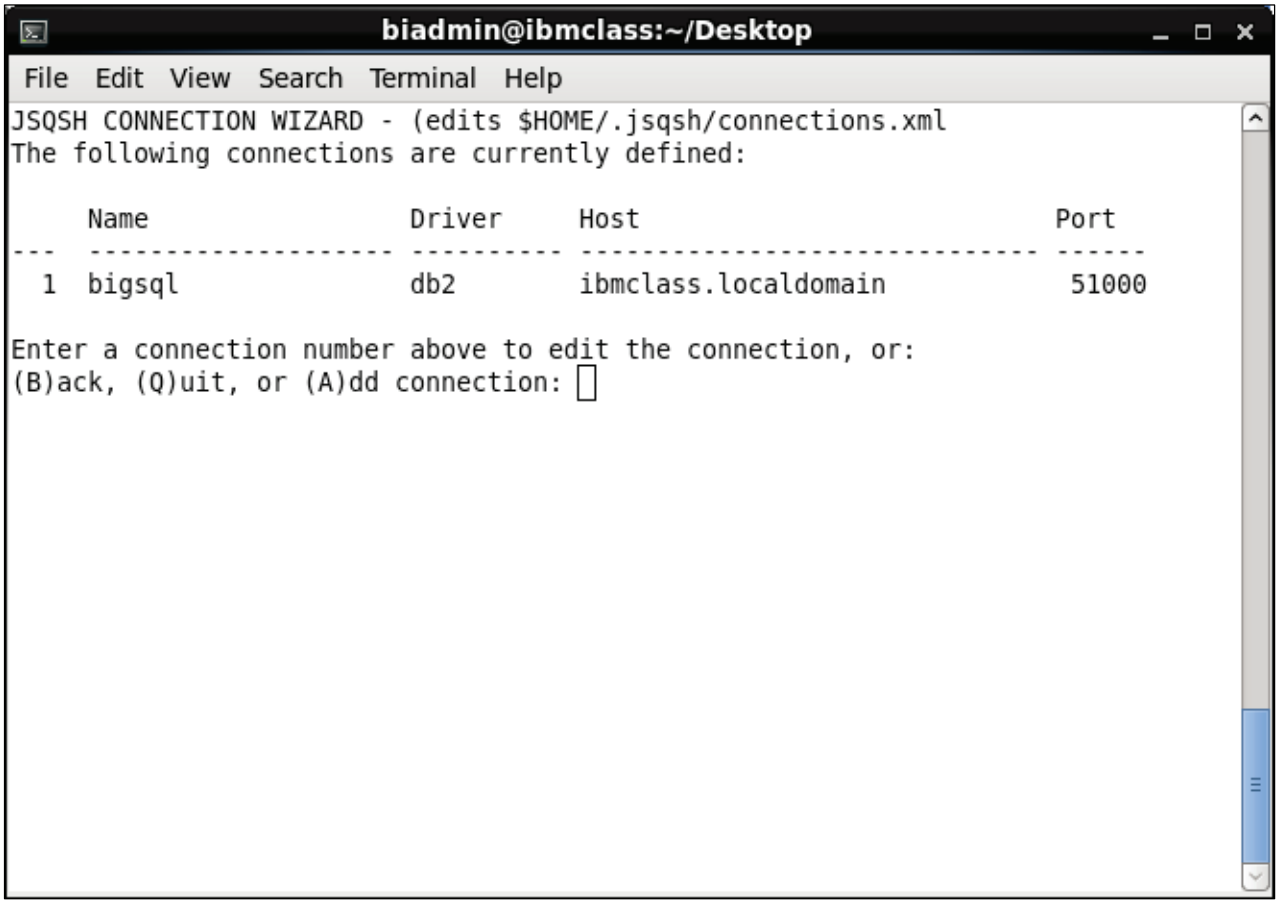
4. Type:

```
\setup
```

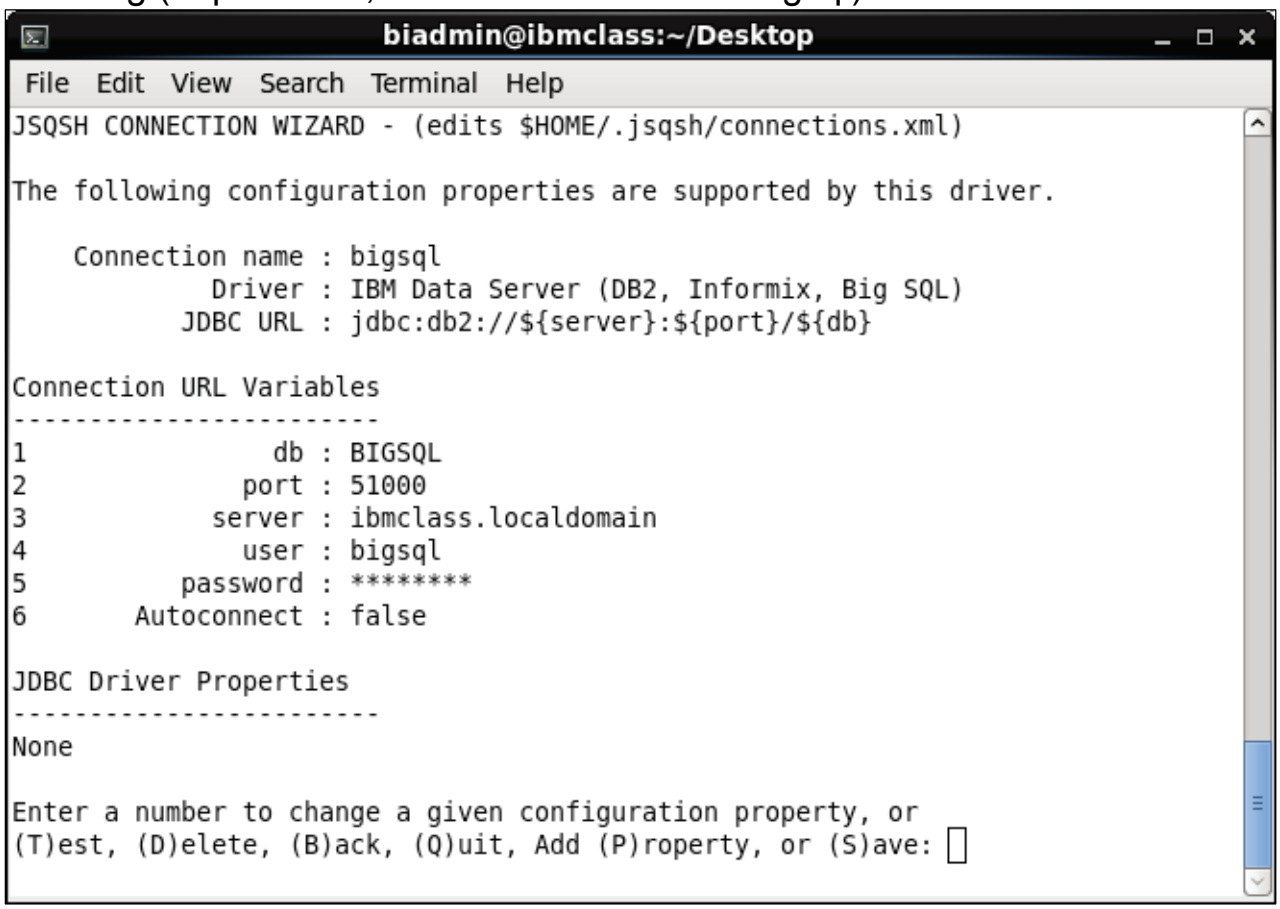


5. Enter **C** to start the connection wizard.

6. Enter **1**, to select the bigsql connection:



For each of the Connection URL variables, ensure yours matches the following (In particular, make sure the user is bigsql):



You want to enter in and save the password so that you can connect automatically for the purposes of this demonstration.

7. Enter **5**, and type in the bigsql service password: **ibm2blue**.
8. Enter **T** to test the connection.
9. Enter **S** to save your configurations.
10. Enter **Q** to quit.

    At this point, you have successfully connected to the *bigsql* schema/database.

Task 5. Getting help for JSqsh.

1. From JSqsh, type **\help** to display a list of available help categories.

```
1> \help
Available help categories. Use "\help <category>" to display topics within that
category
+----------+----------------------------------------------+
| Category | Description                                  |
+----------+----------------------------------------------+
| commands | Help on all avaiable commands                |
| vars     | Help on all avaiable configuration variables |
| topics   | General help topics for jsqsh               |
+----------+----------------------------------------------+
1>
```

2. Type **\help commands** to display help for supported commands. A partial list of supported commands is displayed on the initial screen.

```
1> \help commands
Available commands. Use "\help <command>" to display detailed help for a given c
ommand
+------------+-----------------------------------------------------------------+
| Command    | Description                                                     |
+------------+-----------------------------------------------------------------+
| \alias     | Creates an alias                                                |
| \buf-appen | Appends the contents of one SQL buffer into another             |
| d          |                                                                 |
| \buf-copy  | Copies the contents of one SQL buffer into another              |
| \buf-edit  | Edits a SQL buffer                                              |
| \buf-load  | Loads an external file into a SQL buffer                        |
| \call      | Call a prepared statement                                       |
| \connect   | Establishes a connection to a database.                         |
| \create    | Generates a CREATE TABLE using table definitions                |
| \databases | Displays set of available databases (catalogs)                  |
| \debug     | (Internal) Used to enable debugging                             |
| \describe  | Displays a description of a database object                     |
| \diff      | Compares results from multiple sessions                         |
| \drivers   | Displays a list of JDBC drivers known by jsqsh.                 |
| \echo      | Displays a line of text.                                        |
| \end       | Ends the current session                                        |
| \eval      | Read and execute an input file full of SQL                      |
| \globals   | Displays all global variables                                   |
--More--
```

Press the space bar to display the next page or q to quit the display of help information.

Task 6. Executing basic Big SQL statements.

In this section, you will execute simple JSqsh commands and Big SQL queries so that you can become familiar with the JSqsh shell.

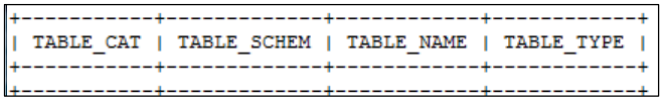
1. From the **JSqsh** shell, connect to your Big SQL server using the connection **bigsql** you created in a previous task.

```
\connect bigsql
```

If prompted for a password, type ibm2blue.

2. Type **\show tables -e | more** to display essential information about all available tables one page at a time.

   If you're working with a newly installed Big SQL server, your results will appear similar to those below.



```
+-----------+-------------+------------+------------+
| TABLE_CAT | TABLE_SCHEM | TABLE_NAME | TABLE_TYPE |
+-----------+-------------+------------+------------+
+-----------+-------------+------------+------------+
```

3. Type the following command into JSqsh to create a simple Hadoop table:

```
create hadoop table test1 (col1 int, col2 varchar(5));
```

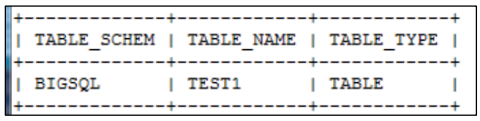
   Because you didn't specify a schema name for the table it was created in your default schema, which is the user name specified in your JDBC connection. This is equivalent to

```
create hadoop table yourID.test1 (col1 int, col2
varchar(5));
```

   Where *yourID* is the user name for your connection. In a previous task, you created a connection using the bigsql user ID, so your table is BIGSQL.TEST1.

4. Display all tables in the current schema with the **\tables** command.

```
\tables
```



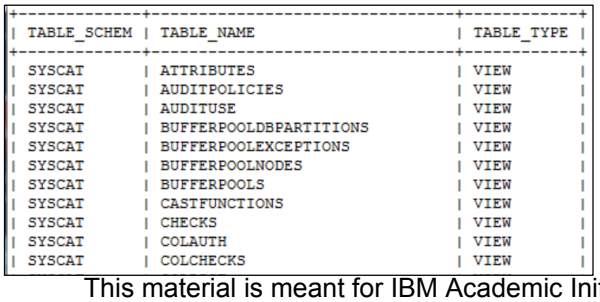
```
+-------------+------------+------------+
| TABLE_SCHEM | TABLE_NAME | TABLE_TYPE |
+-------------+------------+------------+
| BIGSQL      | TEST1      | TABLE      |
+-------------+------------+------------+
```

   Optionally, display information about tables and views created in other schemas, such as the SYSCAT schema used for the Big SQL catalog.

5. Specify the schema name in upper case since it will be used directly to filter the list of tables.

```
\tables -s SYSCAT
```

   Partial results are shown below.



```
+-------------+-------------------------------+------------+
| TABLE_SCHEM | TABLE_NAME                    | TABLE_TYPE |
+-------------+-------------------------------+------------+
| SYSCAT      | ATTRIBUTES                    | VIEW       |
| SYSCAT      | AUDITPOLICIES                 | VIEW       |
| SYSCAT      | AUDITUSE                      | VIEW       |
| SYSCAT      | BUFFERPOOLDBPARTITIONS        | VIEW       |
| SYSCAT      | BUFFERPOOLEXCEPTIONS          | VIEW       |
| SYSCAT      | BUFFERPOOLNODES               | VIEW       |
| SYSCAT      | BUFFERPOOLS                   | VIEW       |
| SYSCAT      | CASTFUNCTIONS                 | VIEW       |
| SYSCAT      | CHECKS                        | VIEW       |
| SYSCAT      | COLAUTH                       | VIEW       |
| SYSCAT      | COLCHECKS                     | VIEW       |
```

6. Insert a row into your table.

```
insert into test1 values (1, 'one');
```

This form of the INSERT statement (INSERT INTO … VALUES …) should be used for test purposes only because the operation will not be parallelized on your cluster. To populate a table with data in a manner that exploits parallel processing, use the Big SQL LOAD command, INSERT INTO … SELECT FROM statement, or CREATE TABLE AS … SELECT statement. You'll learn more about these commands later.

7. To view the metadata about a table, use the **\describe** command with the fully qualified table name in upper case.

```
\describe BIGSQL.TEST1
```

```
+-------------+-------------+-----------+-------------+----------------+-------------+
| TABLE_SCHEM | COLUMN_NAME | TYPE_NAME | COLUMN_SIZE | DECIMAL_DIGITS | IS_NULLABLE |
+-------------+-------------+-----------+-------------+----------------+-------------+
| BIGSQL      | COL1        | INTEGER   |          10 |              0 | YES         |
| BIGSQL      | COL2        | VARCHAR   |           5 |         [NULL] | YES         |
+-------------+-------------+-----------+-------------+----------------+-------------+
```

8. Optionally, query the system for metadata about this table:

```
select tabschema, colname, colno, typename, length
from syscat.columns
where tabschema = USER and tabname= 'TEST1';
```

Once again, notice that you used the table name in upper case in these queries and \describe command. This is because table and column names are folded to upper case in the system catalog tables.

You can split the query across multiple lines in the JSqsh shell if you'd like. Whenever you press **Enter**, the shell will provide another line for you to continue your command or SQL statement. A semi-colon or **go** command causes your SQL statement to execute.

```
+-----------+---------+-------+----------+--------+
| TABSCHEMA | COLNAME | COLNO | TYPENAME | LENGTH |
+-----------+---------+-------+----------+--------+
| BIGSQL    | COL1    |     0 | INTEGER  |      4 |
| BIGSQL    | COL2    |     1 | VARCHAR  |      5 |
+-----------+---------+-------+----------+--------+
2 rows in results(first row: 0.1s; total: 0.1s)
```

SYSCAT.COLUMNS is one of a number of views supplied over system catalog tables automatically maintained for you by the Big SQL service.

Issue a query that restricts the number of rows returned to 5.

9. Select the first 5 rows from SYSCAT.TABLES:

```
select tabschema, tabname from syscat.tables fetch
first 5 rows only;
```

```
+-----------+------------------------+
| TABSCHEMA | TABNAME                |
+-----------+------------------------+
| BIGSQL    | TEST1                  |
| SYSCAT    | ATTRIBUTES             |
| SYSCAT    | AUDITPOLICIES          |
| SYSCAT    | AUDITUSE               |
| SYSCAT    | BUFFERPOOLDBPARTITIONS |
+-----------+------------------------+
5 rows in results(first row: 0.1s; total: 0.1s)
```

Restricting the number of rows returned by a query is a useful development technique when working with large volumes of data.

If you plan to use JSqsh frequently, it's worth exploring some additional features. This demonstration shows you how to recall previous commands, redirect output to local files, and execute scripts.

Review the history of commands you recently executed in the JSqsh shell.

10. Type `\history` and press **Enter**.

    Note that previously run statements are prefixed with a number in parentheses. You can reference this number in the JSqsh shell to recall that query.

11. Type `!!` (two exclamation points, without spaces) to recall the previously run statement.

    The previous statement selects the first 5 rows from SYSCAT.TABLES.

12. To run the statement, type a semi-colon on the following line.

```
5 rows in results(first row: 0.1s; total: 0.1s)
[bdvs1052.svl.ibm.com][bigsql] 1> \history
(1) create hadoop table test1 (col1 int, col2 varchar(5))
(2) insert into test1 values (1, 'one')
(3) select tabschema, colname, colno, typename, length
    from syscat.columns
    where tabschema = USER and tabname= 'TEST1'
(4) select tabschema, tabname from syscat.tables fetch first 5 rows only
[bdvs1052.svl.ibm.com][bigsql] 1> !!
[bdvs1052.svl.ibm.com][bigsql] 1> select tabschema, tabname from syscat.tables fetch first 5 rows only
[bdvs1052.svl.ibm.com][bigsql] 2> ;
+-----------+------------------------+
| TABSCHEMA | TABNAME                |
+-----------+------------------------+
| BIGSQL    | TEST1                  |
| SYSCAT    | ATTRIBUTES             |
| SYSCAT    | AUDITPOLICIES          |
| SYSCAT    | AUDITUSE               |
| SYSCAT    | BUFFERPOOLDBPARTITIONS |
+-----------+------------------------+
5 rows in results(first row: 0.1s; total: 0.2s)
```

Recall a previous SQL statement by referencing the number reported via the \history command. For example, if you wanted to recall the 4th statement, you would enter !4. After the statement is recalled, add a semi-column to the final line to run the statement.

You will experiment with JSqsh's ability to support piping of output to an external program.

13. Enter the following two lines on the command shell:

```
select tabschema, tabname from syscat.tables

go | more
```

The go statement in the second line causes the query on the first line to be executed. (Note that there is no semi-colon at the end of the SQL query on the first line. The semi-colon is a Big SQL short cut for the JSqsh go command.) The | more clause causes the output that results from running the query to be piped through the Unix/Linux more command to display one screen of content at a time.

Your results should look similar to this:

```
| TABSCHEMA | TABNAME                          |
+-----------+----------------------------------+
| BIGSQL    | TEST1                            |
| SYSCAT    | ATTRIBUTES                       |
| SYSCAT    | AUDITPOLICIES                    |
| SYSCAT    | AUDITUSE                         |
| SYSCAT    | BUFFERPOOLDBPARTITIONS           |
| SYSCAT    | BUFFERPOOLEXCEPTIONS             |
| SYSCAT    | BUFFERPOOLNODES                  |
| SYSCAT    | BUFFERPOOLS                      |
| SYSCAT    | CASTFUNCTIONS                    |
| SYSCAT    | CHECKS                           |
| SYSCAT    | COLAUTH                          |
| SYSCAT    | COLCHECKS                        |
| SYSCAT    | COLDIST                          |
| SYSCAT    | COLGROUPCOLS                     |
| SYSCAT    | COLGROUPDIST                     |
| SYSCAT    | COLGROUPDISTCOUNTS               |
| SYSCAT    | COLGROUPS                        |
| SYSCAT    | COLIDENTATTRIBUTES               |
| SYSCAT    | COLLATIONS                       |
| SYSCAT    | COLOPTIONS                       |
| SYSCAT    | COLUMNS                          |
| SYSCAT    | COLUSE                           |
--More--443 rows in results(first row: 0.1s; total: 0.8s)
[bdvs1052.svl.ibm.com][bigsql] 1>
```

14. Since there are more than 400 rows to display in this example, enter `q` to quit displaying further results and return to the JSqsh shell.

Experiment with JSqsh's ability to redirect output to a local file rather than the console display.

15. Enter the following two lines on the command shell, adjusting the path information on the final line as needed for your environment:

```
select tabschema, colname, colno, typename, length
from syscat.columns
where tabschema = USER and tabname= 'TEST1'
go > $HOME/test1.out
```

This example directs the output of the query shown on the first line to the output file test1.out in your user's home directory.

16. Exit the shell:

```
quit
```

17. From a terminal window, view the output file:

```
cat $HOME/test1.out
```

```
[bigdata@bdvs1054 ~]$ cat $HOME/test1.out
+-----------+---------+-------+----------+--------+
| TABSCHEMA | COLNAME | COLNO | TYPENAME | LENGTH |
+-----------+---------+-------+----------+--------+
| BIGSQL    | COL1    |     0 | INTEGER  |      4 |
| BIGSQL    | COL2    |     1 | VARCHAR  |      5 |
+-----------+---------+-------+----------+--------+
```

Invoke JSqsh using an input file containing Big SQL commands to be executed. Maintaining SQL script files can be quite handy for repeatedly executing various queries.

From the Unix/Linux command line, use any available editor to create a new file in your local directory named test.sql.

18. Type:

```
vi test.sql
```

a. Add the following 2 queries into your file

```
select tabschema, tabname from syscat.tables fetch
first 5 rows only;

select tabschema, colname, colno, typename, length
from syscat.columns
fetch first 10 rows only;
```

b. Save your file (select ‘esc’ to exit INSERT mode then type :wq) and return to the command line.

c. Invoke JSQSH, instructing it to connect to your Big SQL database and execute the contents of the script you just created. (You may need to adjust the path or user information shown below to match your environment.)

```
/usr/ibmpacks/common-utils/jsqsh/2.14/bin/jsqsh
bigsql < test.sql
```

In this example, bigsql is the name of the database connection you created in an earlier task.

d. Inspect the output.

As you will see, JSQSH executes each instruction and displays its output. (Partial results are shown below.)

```
[bigdata@bdvs1054 ~]$ /usr/ibmpacks/bigsql/4.0/jsqsh/bin/jsqsh bigsql -P bigsql < test.sql
JSqsh Release 2.11, Copyright (C) 2007-2015, Scott C. Gray
Type \help for available help topics. Using JLine.
[bdvs1052.svl.ibm.com][bigsql] 1> select tabschema, tabname from syscat.tables fetch first 5 rows only
+-----------+-------------------------+
| TABSCHEMA | TABNAME                 |
+-----------+-------------------------+
| BIGSQL    | TEST1                   |
| SYSCAT    | ATTRIBUTES              |
| SYSCAT    | AUDITPOLICIES           |
| SYSCAT    | AUDITUSE                |
| SYSCAT    | BUFFERPOOLDBPARTITIONS  |
+-----------+-------------------------+
5 rows in results(first row: 0.3s; total: 0.3s)
[bdvs1052.svl.ibm.com][bigsql] 1>
[bdvs1052.svl.ibm.com][bigsql] 2> select tabschema, colname, colno, typename, length
[bdvs1052.svl.ibm.com][bigsql] 3> from syscat.columns
[bdvs1052.svl.ibm.com][bigsql] 4> fetch first 10 rows only;
+-----------+-------------+-------+-----------+-----------+
| TABSCHEMA | COLNAME     | COLNO | TYPENAME  |    LENGTH |
+-----------+-------------+-------+-----------+-----------+
| SYSIBM    | NAME        |     0 | VARCHAR   |       128 |
| SYSIBM    | CREATOR     |     1 | VARCHAR   |       128 |
| SYSIBM    | TYPE        |     2 | CHARACTER |         1 |
| SYSIBM    | CTIME       |     3 | TIMESTAMP |        10 |
| SYSIBM    | REMARKS     |     4 | VARCHAR   |       254 |
| SYSIBM    | PACKED_DESC |     5 | BLOB      | 133169152 |
| SYSIBM    | VIEW_DESC   |     6 | BLOB      |   4190000 |
| SYSIBM    | COLCOUNT    |     7 | SMALLINT  |         2 |
| SYSIBM    | FID         |     8 | SMALLINT  |         2 |
| SYSIBM    | TID         |     9 | SMALLINT  |         2 |
+-----------+-------------+-------+-----------+-----------+
10 rows in results(first row: 0.1s; total: 0.1s)
```

Next, you will clean up the database.

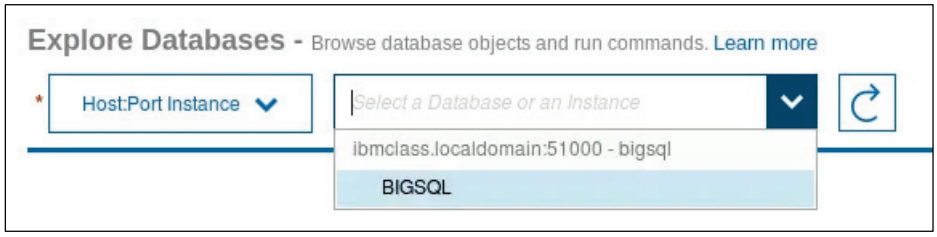
19. Launch the JSqsh shell again and issue this command:

```
drop table test1;
```

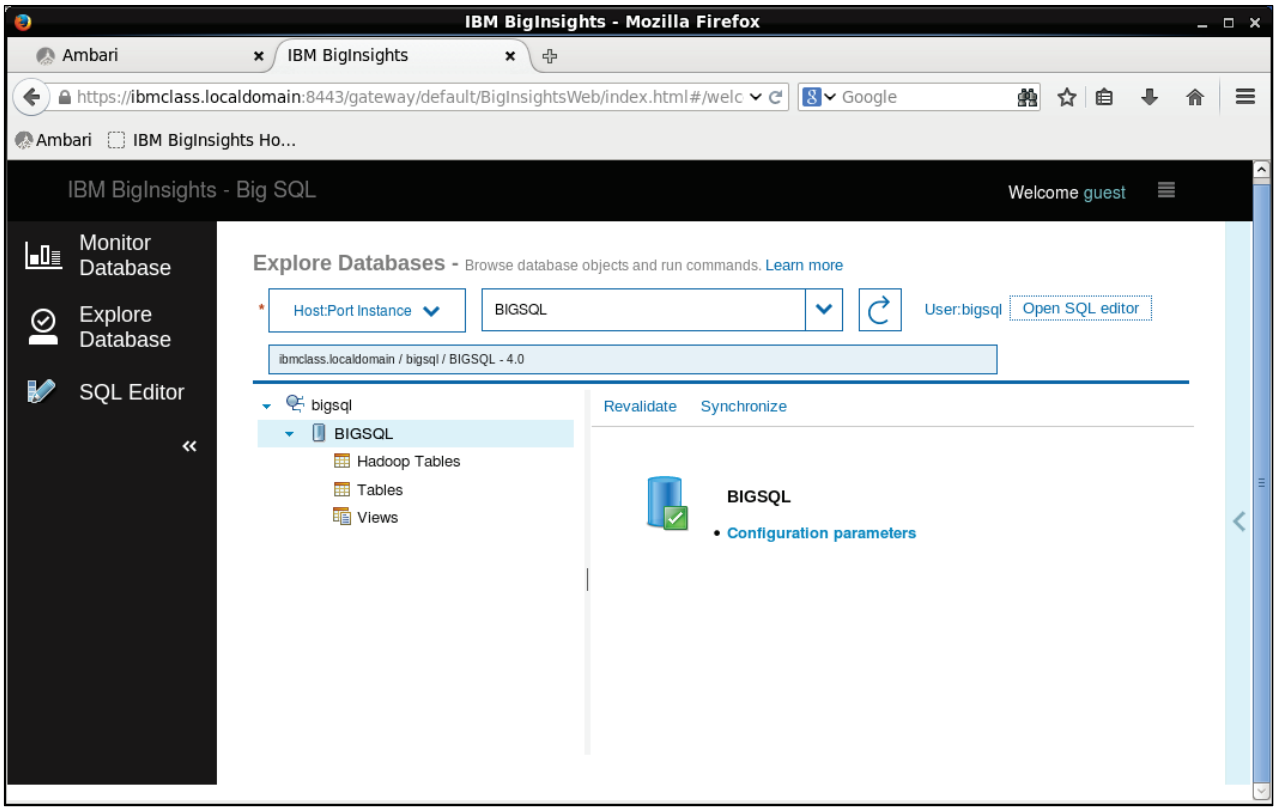
There's more to JSqsh than this short lab can cover. Visit the JSqsh wiki (https://github.com/scgray/jsqsh/wiki) to learn more.

Task 7. Exploring Big SQL through Ambari.

1. Launch BigInsights Home (https://ibmclass.localdomain:8443/gateway/default/BigInsightsWeb/index.html#/welcome) and click on the Big SQL service to open up the Big SQL page.
2. Click on **Explore Database** to see the default BIGSQL table that you created earlier.
3. Select the BIGSQL instance:



4. Login using the Big SQL credentials, use **bigsql/ibm2blue**.

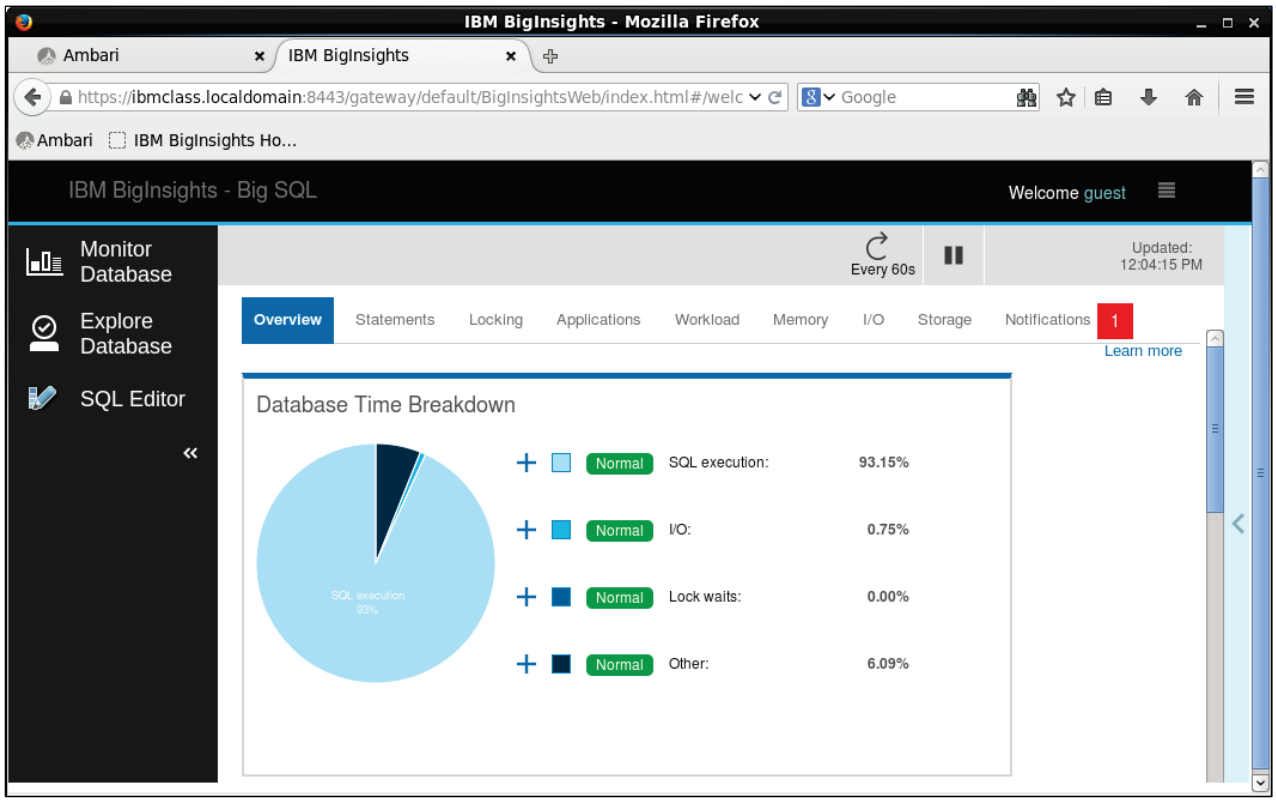


From there, you can explore your Big SQL database.

5. Click on the **Hadoop Tables** link to load the view of the tables.

   When you create Big SQL tables using the hadoop keyword, the tables would show up here. If you omit that keyword, and you did not specify the SYSHADOOP.COMPATIBILITY_MODE environment variable, then the tables would show up under Tables.

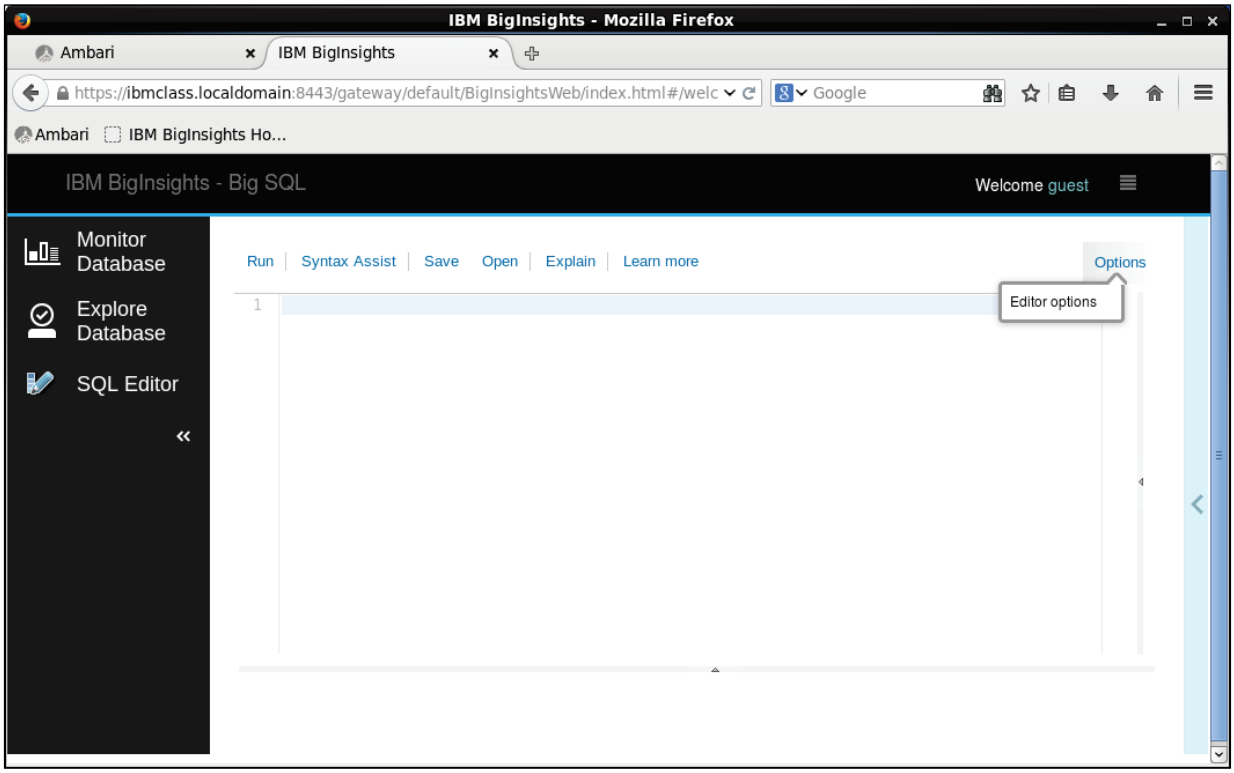
6. On the left panel, click on the **Monitor Database** to view the performance metrics for the database.



   There are number of tabs on the monitoring page that you can explore. Click on each of them to find out more.

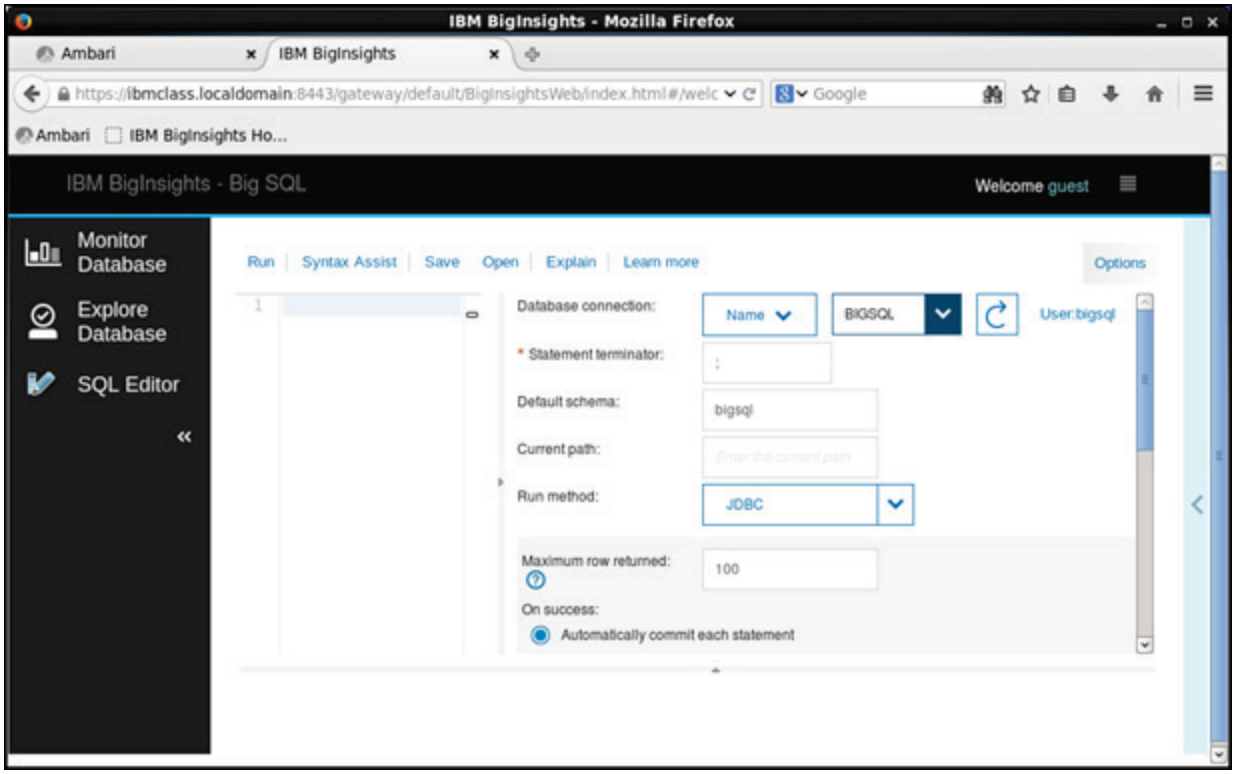
7. From the left panel, click **SQL Editor**.

8. Explore the **Options** menu to specify the database connection to use.



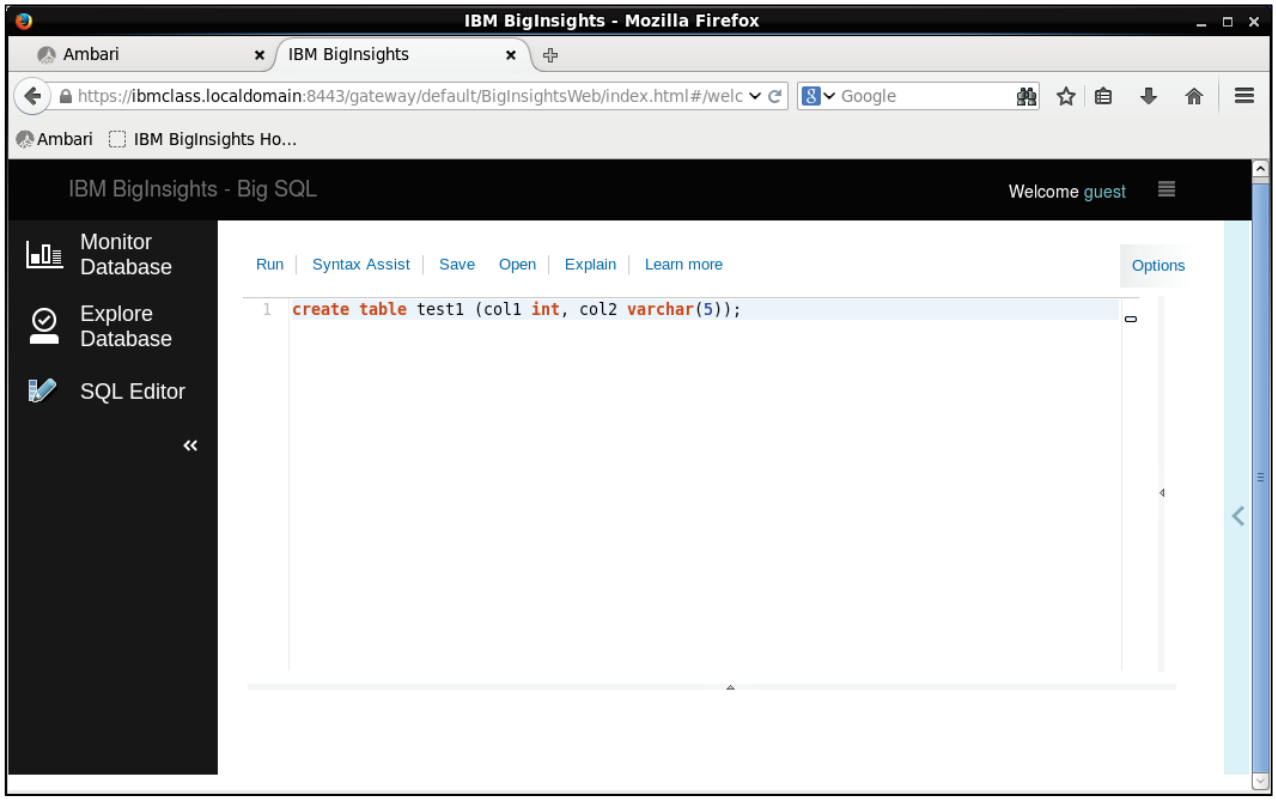
9. From the Database connection dropdown, select BIGSQL and in the Default schema field, type BIGSQL.

The results appear as follows:



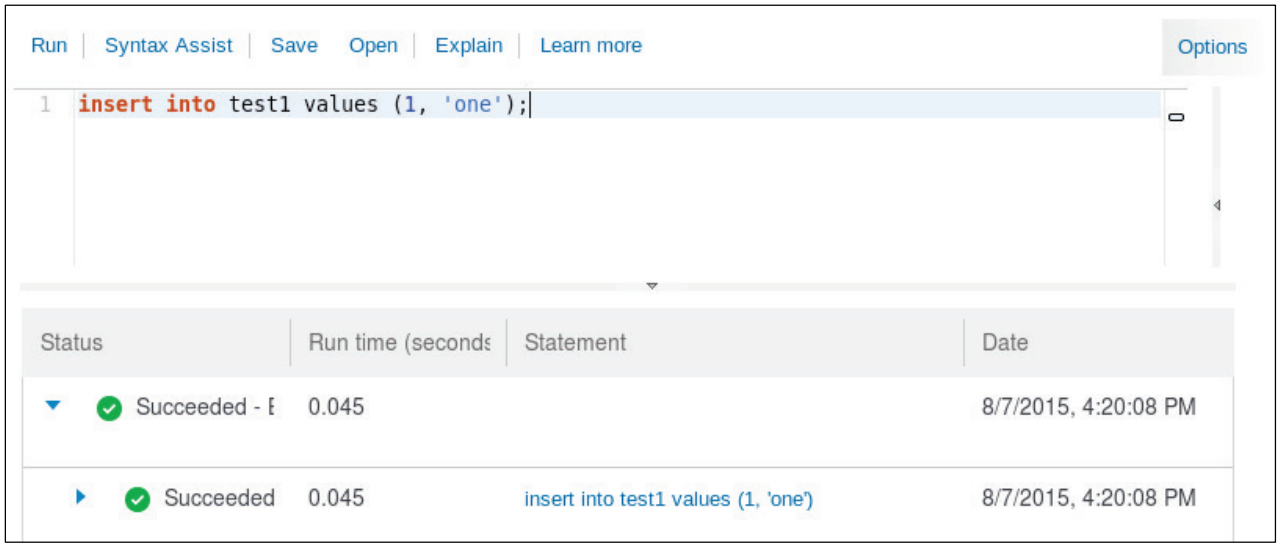
10. Collapse the Options pane after you are done.
11. In the Run query field, type:

```
create table test1 (col1 int, col2 varchar(5));
```



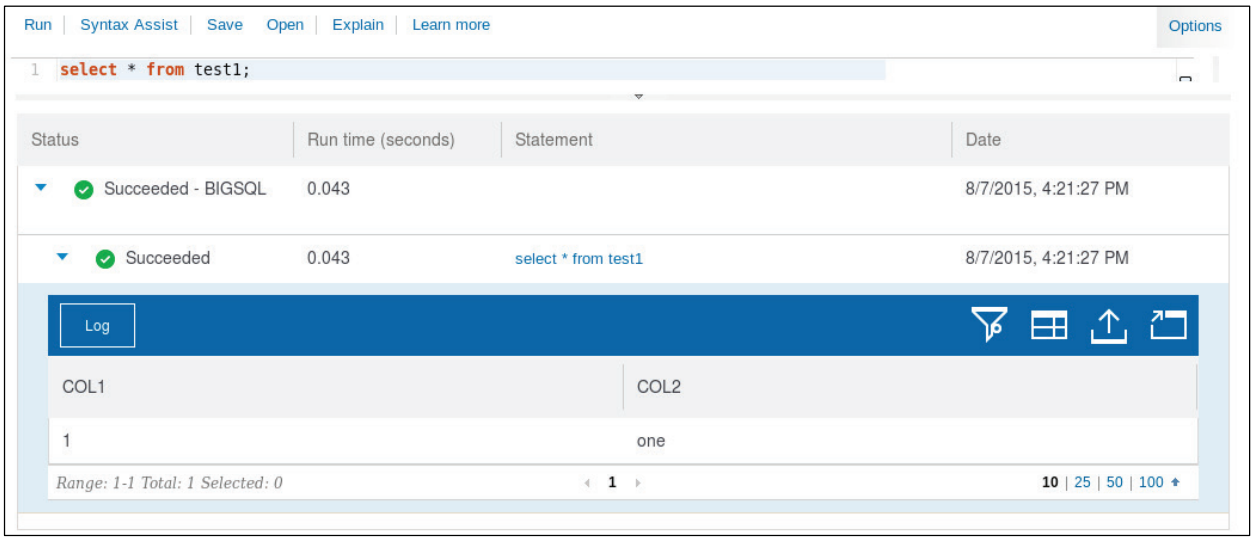
12. Click **Run** to execute the query.
13. Insert a row into the table:

```
insert into test1 values (1, 'one');
```



14. Select from the table:

```
select * from test1;
```



You are no longer require the table.

15. Drop the table:

```
drop table test1;
```

Task 8. Creating sample tables.

In this task, you will create several sample tables.

Determine the location of the sample data in your local file system and make a note of it. You will need to use this path specification when issuing LOAD commands later in this demonstration.

Subsequent examples in this section presume your sample data is in the **/usr/ibmpacks/bigsql/4.0/bigsql/samples/data** directory.

This is the location of the data in typical Big SQL installations.

Also note that the statements in this demonstration are provided in the scripts that are located in **/home/biadmin/labfiles/bigsql/BIG_SQL_Data_Analysis** for you to copy and paste the statements into your console of choice.

1. Using your choice of either JSqsh or the Big SQL service via Ambari, establish a connection to your Big SQL server following the standard process for your environment.

2. Create 8 tables in your default schema (bigsql). Issue each of the following CREATE TABLE statements one at a time, and verify that each completed successfully. If you are not able to create tables, refer to the troubleshooting section.

```
-- dimension table for region info
CREATE HADOOP TABLE IF NOT EXISTS go_region_dim
( country_key  INT NOT NULL
, country_code INT NOT NULL
, flag_image    VARCHAR(45)
, iso_three_letter_code  VARCHAR(9) NOT NULL
, iso_two_letter_code        VARCHAR(6) NOT NULL
, iso_three_digit_code   VARCHAR(9) NOT NULL
, region_key   INT NOT NULL
, region_code  INT NOT NULL
, region_enVARCHAR(90) NOT NULL
, country_en   VARCHAR(90) NOT NULL
, region_deVARCHAR(90), country_de VARCHAR(90), region_fr VARCHAR(90)
, country_fr   VARCHAR(90), region_ja   VARCHAR(90), country_ja VARCHAR(90)
, region_csVARCHAR(90), country_cs VARCHAR(90), region_da VARCHAR(90)
, country_da   VARCHAR(90), region_el   VARCHAR(90), country_el VARCHAR(90)
, region_esVARCHAR(90), country_es VARCHAR(90), region_fi VARCHAR(90)
, country_fi   VARCHAR(90), region_hu   VARCHAR(90), country_hu VARCHAR(90)
, region_idVARCHAR(90), country_id VARCHAR(90), region_it VARCHAR(90)
, country_it   VARCHAR(90), region_ko   VARCHAR(90), country_ko VARCHAR(90)
, region_msVARCHAR(90), country_ms VARCHAR(90), region_nl VARCHAR(90)
, country_nl   VARCHAR(90), region_no   VARCHAR(90), country_no VARCHAR(90)
, region_plVARCHAR(90), country_pl VARCHAR(90), region_pt VARCHAR(90)
, country_pt   VARCHAR(90), region_ru   VARCHAR(90), country_ru VARCHAR(90)
, region_scVARCHAR(90), country_sc VARCHAR(90), region_sv VARCHAR(90)
, country_sv   VARCHAR(90), region_tc   VARCHAR(90), country_tc VARCHAR(90)
, region_thVARCHAR(90), country_th VARCHAR(90)
)
ROW FORMAT DELIMITED
FIELDS TERMINATED BY '\t'
LINES TERMINATED BY '\n'
STORED AS TEXTFILE
;

-- dimension table tracking method of order for the sale (e.g., Web, fax)
CREATE HADOOP TABLE IF NOT EXISTS sls_order_method_dim
( order_method_key   INT NOT NULL
, order_method_code  INT NOT NULL
, order_method_en    VARCHAR(90) NOT NULL
, order_method_de    VARCHAR(90), order_method_fr   VARCHAR(90)
, order_method_ja    VARCHAR(90), order_method_cs   VARCHAR(90)
, order_method_da    VARCHAR(90), order_method_el   VARCHAR(90)
, order_method_es    VARCHAR(90), order_method_fi   VARCHAR(90)
, order_method_hu    VARCHAR(90), order_method_id   VARCHAR(90)
, order_method_it    VARCHAR(90), order_method_ko   VARCHAR(90)
, order_method_ms    VARCHAR(90), order_method_nl   VARCHAR(90)
, order_method_no    VARCHAR(90), order_method_pl   VARCHAR(90)
, order_method_pt    VARCHAR(90), order_method_ru   VARCHAR(90)
, order_method_sc    VARCHAR(90), order_method_sv   VARCHAR(90)
, order_method_tc    VARCHAR(90), order_method_th   VARCHAR(90)
)
ROW FORMAT DELIMITED
FIELDS TERMINATED BY '\t'
```

```
LINES TERMINATED BY '\n'
STORED AS TEXTFILE
;

-- look up table with product brand info in various languages
CREATE HADOOP TABLE IF NOT EXISTS sls_product_brand_lookup
 ( product_brand_code     INT NOT NULL
 , product_brand_en  VARCHAR(90) NOT NULL
 , product_brand_de  VARCHAR(90), product_brand_fr  VARCHAR(90)
 , product_brand_ja  VARCHAR(90), product_brand_cs  VARCHAR(90)
 , product_brand_da  VARCHAR(90), product_brand_el  VARCHAR(90)
 , product_brand_es  VARCHAR(90), product_brand_fi  VARCHAR(90)
 , product_brand_hu  VARCHAR(90), product_brand_id  VARCHAR(90)
 , product_brand_it  VARCHAR(90), product_brand_ko  VARCHAR(90)
 , product_brand_ms  VARCHAR(90), product_brand_nl  VARCHAR(90)
 , product_brand_no  VARCHAR(90), product_brand_pl  VARCHAR(90)
 , product_brand_pt  VARCHAR(90), product_brand_ru  VARCHAR(90)
 , product_brand_sc  VARCHAR(90), product_brand_sv  VARCHAR(90)
 , product_brand_tc  VARCHAR(90), product_brand_th  VARCHAR(90)
 )
 ROW FORMAT DELIMITED
 FIELDS TERMINATED BY '\t'
 LINES TERMINATED BY '\n'
 STORED AS TEXTFILE
 ;

 -- product dimension table
 CREATE HADOOP TABLE IF NOT EXISTS sls_product_dim
 ( product_key  INT NOT NULL
 , product_line_code      INT NOT NULL
 , product_type_key  INT NOT NULL
 , product_type_code      INT NOT NULL
 , product_number    INT NOT NULL
 , base_product_key  INT NOT NULL
 , base_product_number    INT NOT NULL
 , product_color_code     INT
 , product_size_code      INT
 , product_brand_key      INT NOT NULL
 , product_brand_code     INT NOT NULL
 , product_image          VARCHAR(60)
 , introduction_date      TIMESTAMP
 , discontinued_date      TIMESTAMP
 )
 ROW FORMAT DELIMITED
 FIELDS TERMINATED BY '\t'
 LINES TERMINATED BY '\n'
 STORED AS TEXTFILE
 ;

 -- look up table with product line info in various languages
 CREATE HADOOP TABLE IF NOT EXISTS sls_product_line_lookup
 ( product_line_code      INT NOT NULL
 , product_line_en   VARCHAR(90) NOT NULL
 , product_line_de   VARCHAR(90), product_line_fr   VARCHAR(90)
 , product_line_ja   VARCHAR(90), product_line_cs   VARCHAR(90)
 , product_line_da   VARCHAR(90), product_line_el   VARCHAR(90)
 , product_line_es   VARCHAR(90), product_line_fi   VARCHAR(90)
 , product_line_hu   VARCHAR(90), product_line_id   VARCHAR(90)
 , product_line_it   VARCHAR(90), product_line_ko   VARCHAR(90)
```

```
, product_line_ms   VARCHAR(90), product_line_nl   VARCHAR(90)
, product_line_no   VARCHAR(90), product_line_pl   VARCHAR(90)
, product_line_pt   VARCHAR(90), product_line_ru   VARCHAR(90)
, product_line_sc   VARCHAR(90), product_line_sv   VARCHAR(90)
, product_line_tc   VARCHAR(90), product_line_th   VARCHAR(90)
)
ROW FORMAT DELIMITED
FIELDS TERMINATED BY '\t'
LINES TERMINATED BY '\n'
STORED AS TEXTFILE;

-- look up table for products
CREATE HADOOP TABLE IF NOT EXISTS sls_product_lookup
( product_number    INT NOT NULL
, product_language  VARCHAR(30) NOT NULL
, product_name VARCHAR(150) NOT NULL
, product_description    VARCHAR(765)
)
ROW FORMAT DELIMITED
FIELDS TERMINATED BY '\t'
LINES TERMINATED BY '\n'
STORED AS TEXTFILE;

-- fact table for sales
CREATE HADOOP TABLE IF NOT EXISTS sls_sales_fact
( order_day_key     INT NOT NULL
, organization_key  INT NOT NULL
, employee_key           INT NOT NULL
, retailer_key           INT NOT NULL
, retailer_site_key      INT NOT NULL
, product_key  INT NOT NULL
, promotion_key     INT NOT NULL
, order_method_key  INT NOT NULL
, sales_order_key   INT NOT NULL
, ship_day_key           INT NOT NULL
, close_day_key     INT NOT NULL
, quantity     INT
, unit_cost    DOUBLE
, unit_price   DOUBLE
, unit_sale_price   DOUBLE
, gross_margin           DOUBLE
, sale_total   DOUBLE
, gross_profit           DOUBLE
)
ROW FORMAT DELIMITED
FIELDS TERMINATED BY '\t'
LINES TERMINATED BY '\n'
STORED AS TEXTFILE
;

-- fact table for marketing promotions
CREATE HADOOP TABLE IF NOT EXISTS mrk_promotion_fact
( organization_key  INT NOT NULL
, order_day_key     INT NOT NULL
, rtl_country_key   INT NOT NULL
, employee_key           INT NOT NULL
, retailer_key           INT NOT NULL
, product_key  INT NOT NULL
, promotion_key     INT NOT NULL
```

```
, sales_order_key   INT NOT NULL
, quantity     SMALLINT
, unit_cost    DOUBLE
, unit_price   DOUBLE
, unit_sale_price   DOUBLE
, gross_margin          DOUBLE
, sale_total   DOUBLE
, gross_profit          DOUBLE
)
ROW FORMAT DELIMITED
FIELDS TERMINATED BY '\t'
LINES TERMINATED BY '\n'
STORED AS TEXTFILE;
```

Let's briefly explore some aspects of the CREATE TABLE statements shown here. If you have a SQL background, the majority of these statements should be familiar to you. However, after the column specification, there are some additional clauses unique to Big SQL – clauses that enable it to exploit Hadoop storage mechanisms (in this case, Hive). The ROW FORMAT clause specifies that fields are to be terminated by tabs ("\t") and lines are to be terminated by new line characters ("\n"). The table will be stored in a TEXTFILE format, making it easy for a wide range of applications to work with. For details on these clauses, refer to the Apache Hive documentation.

Task 9. Troubleshooting - Optional.

Big SQL has some limitations running on a single node cluster (actually, anything less than a 3 node cluster) and for good reasons too. You do not ever want to have a single node cluster for your production environment.

In the training world, simpler is better. Your lab environment should have been configured with this fix to allow **creating** Big SQL tables, but if it wasn't, you can run this yourself:

1. Open up a new terminal.
2. Switch to the **bigsql** user.
3. Run this command to connect to the bigsql database:

```
db2 connect to bigsql

Database Connection Information
Database server        = DB2/LINUXX8664 10.6.3
SQL authorization ID   = BIGSQL
Local database alias   = BIGSQL
```


4. Run this command for the fix:

```
db2set DB2_DYNAMIC_PMAP=INCLUDE_HEAD_NODE
```

5. Restart the **Big SQL** service from **Ambari**.
6. This will allow you to create Big SQL tables (if you were not able to before).

Results:

You connected to the Big SQL Server, a component of IBM BigInsights. In particular, you connected to the Big SQL server using JSqsh, a CLI for Big SQL. You also so how to get up and running with JSqsh. This demonstration will also showed you how to explore the Big SQL service through Apache Ambari, another open source project for cluster manager.

IBM Training

Unit summary

- Understand how Big SQL fits in the Hadoop architecture
- Accessing and using Big SQL
- Creating Big SQL schemas and tables

Unit Summary

IBM Training

Querying Hadoop data using Big SQL

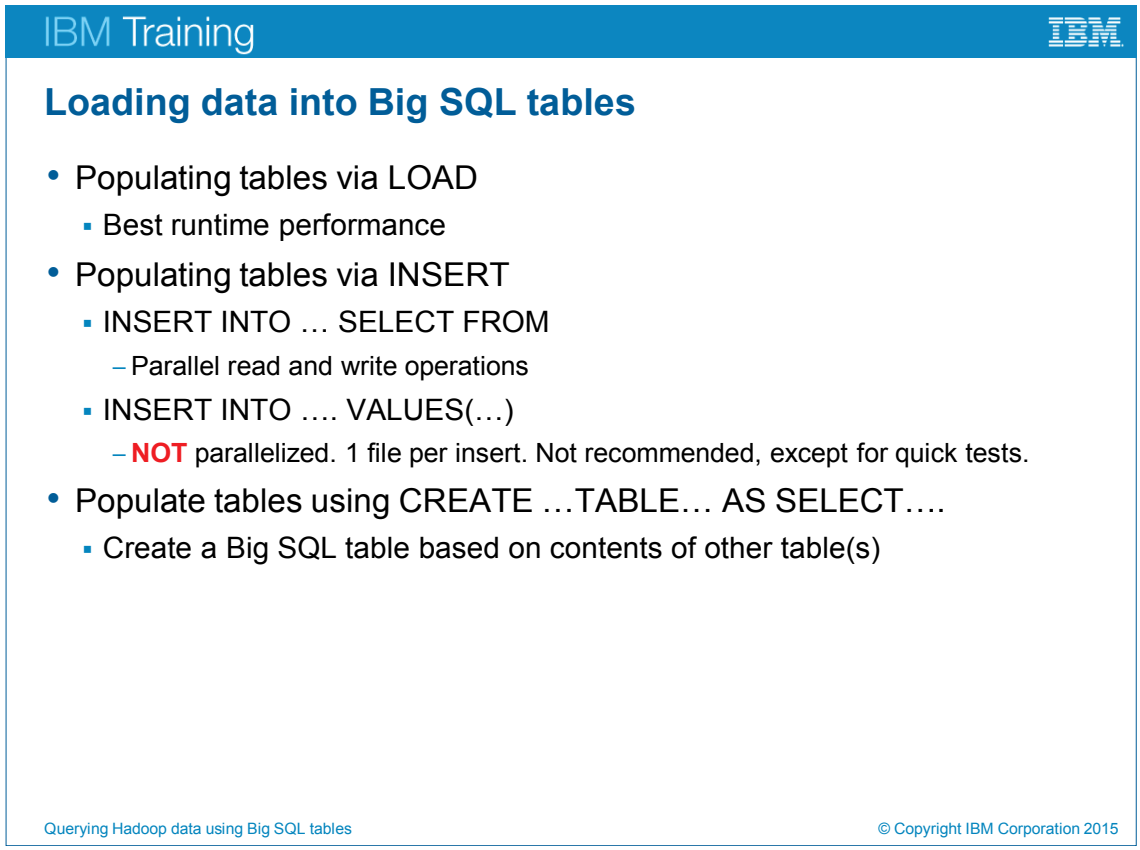
IBM BigInsights v4.0

IBM Training

Unit objectives

- Loading data into Big SQL tables
- List and understand the Big SQL data types
- Understand and use Big SQL supported file formats
- Querying Big SQL tables

Unit objectives



Loading data into Big SQL tables

There are two ways to get your HDFS data into Big SQL tables. Later on, you will see how to create Big SQL tables on top of your Hadoop data, but for now, here is how you would load data into these tables. First, there is the LOAD operation. This is the recommended operation as it yields the best performance. Another method to get data into your Big SQL table is using INSERT operations. There are two types of INSERTs. The first one listed is the INSERT INTO … SELECT FROM. This operation performs the read and write operations in parallel. The second one, a more traditional INSERT INTO … VALUES (…) is NOT parallelized and will produce 1 file per insert, making it HIGHLY INEFFICIENT for your Hadoop data. You are not recommended to use this unless it is just for testing simple and quick operations. The third way to get data into the table is during table creation time, using the CREATE … TABLE … AS SELECT … operation.

IBM Training

Populating Big SQL tables via LOAD

- Typically the best runtime performance
- Load data from local or remote file system

```
load hadoop using file url
'sftp://myID:myPassword@myServer.ibm.com:22/install-
dir/bigsql/samples/data/GOSALESDW.GO_REGION_DIM.txt' with SOURCE PROPERTIES
('field.delimiter'='\t')  INTO TABLE gosalesdw.GO_REGION_DIM overwrite;
```

- Load data from RDBMS (DB2, Netezza, Teradata, Oracle, MS-SQL, Informix) via JDBC connection

```
load hadoop
using jdbc connection url 'jdbc:db2://some.host.com:portNum/sampledb'
with parameters (user='myID', password='myPassword')
from table MEDIA columns (ID, NAME)
where 'CONTACTDATE < ''2012-02-01'''
into table media_db2table_jan overwrite
with load properties ('num.map.tasks' = 10);
```

Querying Hadoop data using Big SQL tables

Populating Big SQL tables via LOAD

You can use the LOAD command to populate it with data from files in a remote file system, files in your distributed file system, or data in the remote RDBMS servers listed. For RDBMS data, you specify JDBC URL properties and either enter a SQL query or a table name to identify the data to be retrieved. Behind the scenes, LOAD uses Sqoop connectors to retrieve the necessary data from the source.

In general, LOAD typically offers the best runtime performance for populating tables with data.

IBM Training

Populating Big SQL tables via INSERT (1 of 2)

- INSERT INTO … SELECT FROM …

```
CREATE HADOOP TABLE IF NOT EXISTS big_sales_parquet
( product_key INT NOT NULL, product_name VARCHAR(150),
Quantity INT, order_method_en VARCHAR(90) )
STORED AS parquetfile;

-- source tables do not need to be in Parquet format
insert into big_sales_parquet
SELECT sales.product_key, pnumb.product_name, sales.quantity,
meth.order_method_en
FROM sls_sales_fact sales, sls_product_dim prod,sls_product_lookup pnumb,
sls_order_method_dim meth
WHERE
pnumb.product_language='EN'
AND sales.product_key=prod.product_key
AND prod.product_number=pnumb.product_number
AND meth.order_method_key=sales.order_method_key
and sales.quantity > 5500;
```

Querying Hadoop data using Big SQL tables

Populating Big SQL tables via INSERT (1 of 2)

In some cases, it may be more convenient to populate a table with data using an INSERT statement. The INSERT INTO . . . SELECT FROM statement supports parallel read and write operations and can be a convenient way to populate a table with data retrieved from a query, particularly if you want to change the underlying storage format used for data.

IBM Training

Populating Big SQL tables via INSERT (2 of 2)

- INSERT INTO … VALUES (…)
  - **NOT** parallelized
  - 1 file is created per insert statement
  - Not recommended, except for testing

```
Create table foo (col1 int, col2 varchar(10));
insert into foo values (1, 'hello');
```

Querying Hadoop data using Big SQL tables

Populating Big SQL tables via INSERT (2 of 2)

The traditional INSERT INTO . . . VALUES(…) format is also supported but not recommended for anything but simple test operations. Its work is never parallelized, and each INSERT results in a separate file.

IBM Training

Populating Big SQL tables via CREATE … TABLE … AS SELECT …

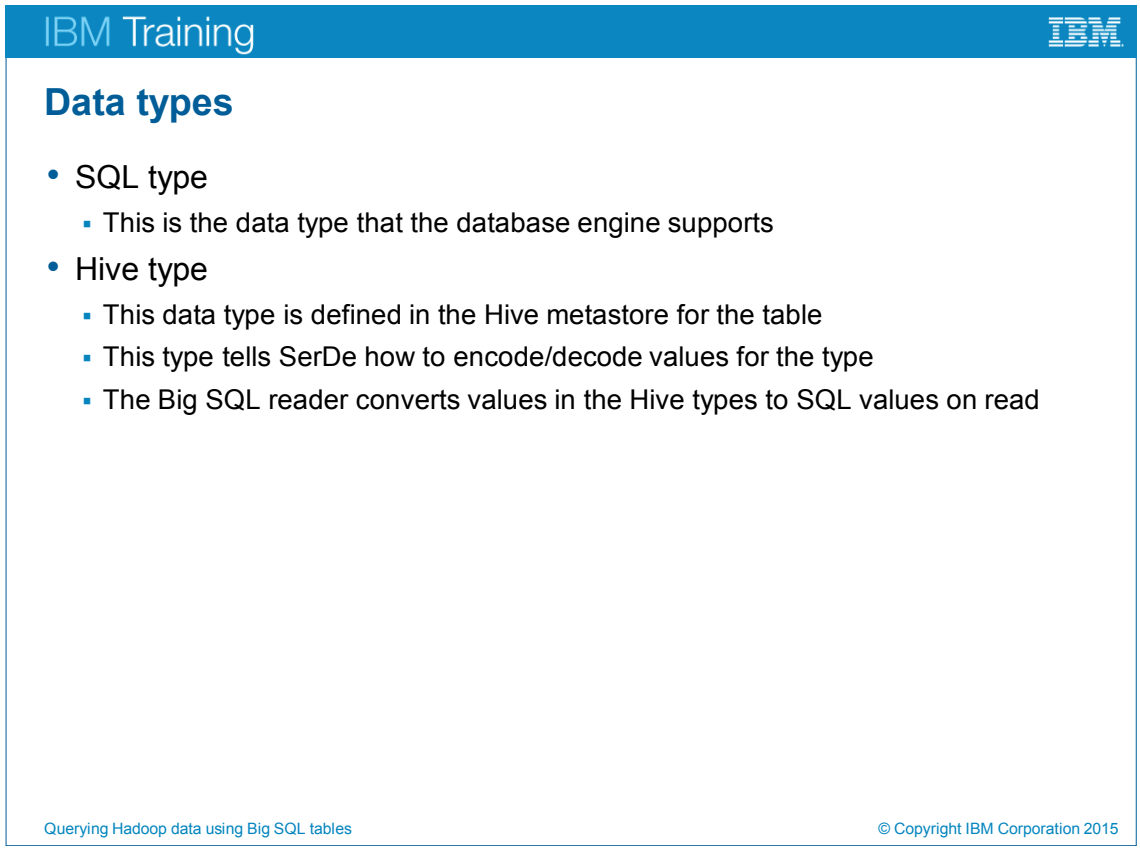
- Source tables can be in different file formats or use different underlying storage mechanism.

```
-- source tables in this example are external (just DFS files)
CREATE HADOOP TABLE IF NOT EXISTS sls_product_flat
 ( product_key INT NOT NULL
 , product_line_code INT NOT NULL
 , product_type_key INT NOT NULL
 , product_type_code INT NOT NULL
 , product_line_en VARCHAR(90)
 , product_line_de VARCHAR(90)
 )
as select product_key, d.product_line_code, product_type_key,
product_type_code, product_line_en, product_line_de
from extern.sls_product_dim d, extern.sls_product_line_lookup l
where d.product_line_code = l.product_line_code;
```

Querying Hadoop data using Big SQL tables

Populating Big SQL tables via CREATE…TABLE…AS…SELECT

Another way to load data into Big SQL table is actually to do so along with the table creation time. The source table(s) can be in different file formats or use a different underlying storage mechanism.



Data types

Big SQL uses HCatalog (and thus the Hive Metastore) as its underlying data representation and access method. Therefore, there are several important distinctions about Big SQL data types that must be understood.

Some data types that are provided by Big SQL are data types that are not available in Hive. However, when physically stored or retrieved from the underlying data source, these types are always represented as a data type that Hive does understand. These data types are referred to as extended data types.

In some cases, data types that Hive represents are not yet fully supported by Big SQL. Some of these types, such as MAP, result in an error when you try to access a column of that type. In other cases, the Hive type is mapped to the nearest Big SQL data type. For example, the Hive TINYINT data type is supported. However, it is promoted to a SMALLINT upon retrieval from Hive and treated as a SMALLINT throughout the SQL statement that accesses the column.

IBM Training

Data types (cont'd)

- Big SQL data types

Hive Type	Big SQL Type
Hive data type	Big SQL data type
TINYINT	SMALLINT
SMALLINT	SMALLINT
INT	INT
BIGINT	BIGINT
FLOAT	DOUBLE
DOUBLE	DOUBLE
STRING	VARCHAR(max)
TIMESTAMP	TIMESTAMP(9)
BOOLEAN	SMALLINT
ARRAY	(Ordinary) ARRAY
MAP	(Associative) ARRAY

Data types (cont'd)

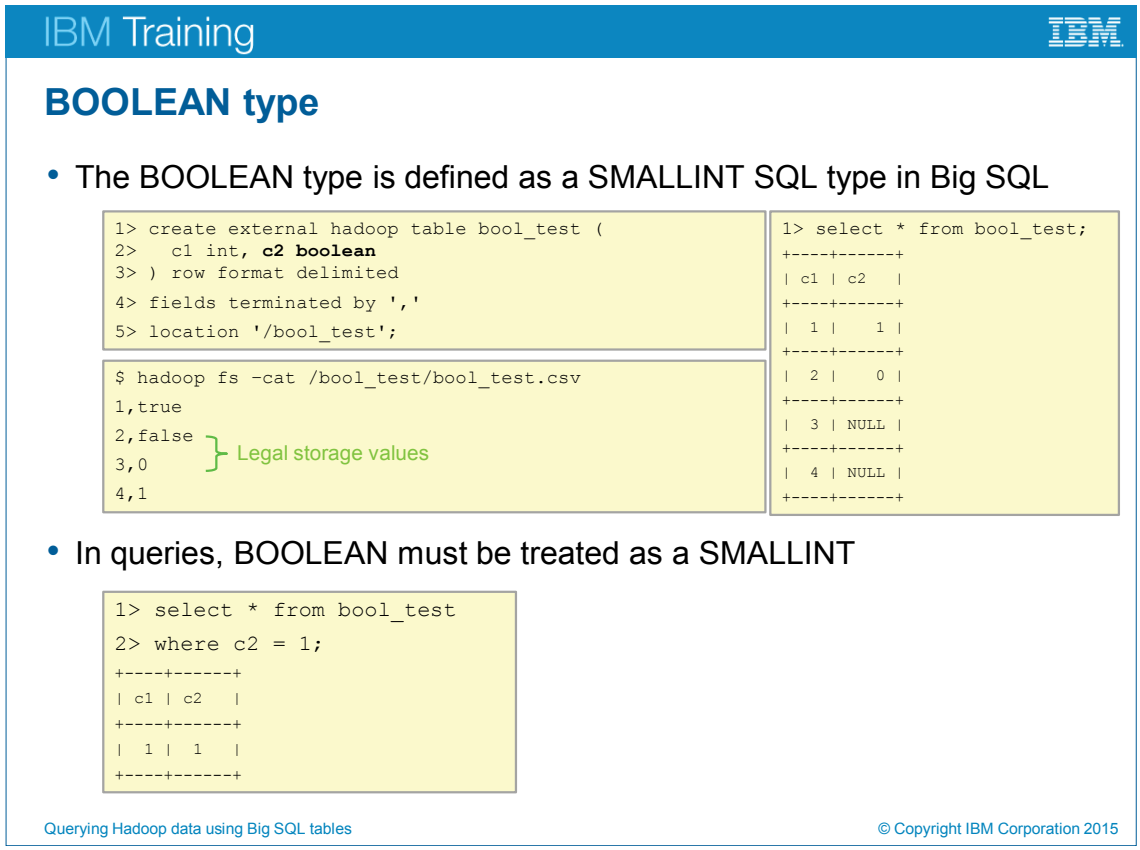
Here is a table that shows the relationship between the various data types.

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. This is just a subset of all the available Big SQL data types.

I want to point out that if you have Strings in Hive, you should explicitly define a VARCHAR with only the necessary characters that you need. If you do not define it, Big SQL will allocate the maximum value, which will be unnecessary in most cases and will cause memory overflow.

You can find out more at the Knowledge Center (http://www-01.ibm.com/support/knowledgecenter/SSPT3X_4.0.0/com.ibm.swg.im.infosphere.biginsights.dev.doc/doc/biga_numbers.html)

So if you have Hive applications, you can use the types defined in there in your Big SQL applications.



BOOLEAN type

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.

┌ Example ──────────────────

You create a table with a BOOLEAN column. The only valid values for that 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 the *true* and *false*. For the 0 and 1 value, the database engine does not recognize it (since it isn't a valid BOOLEAN), so it returns NULL.

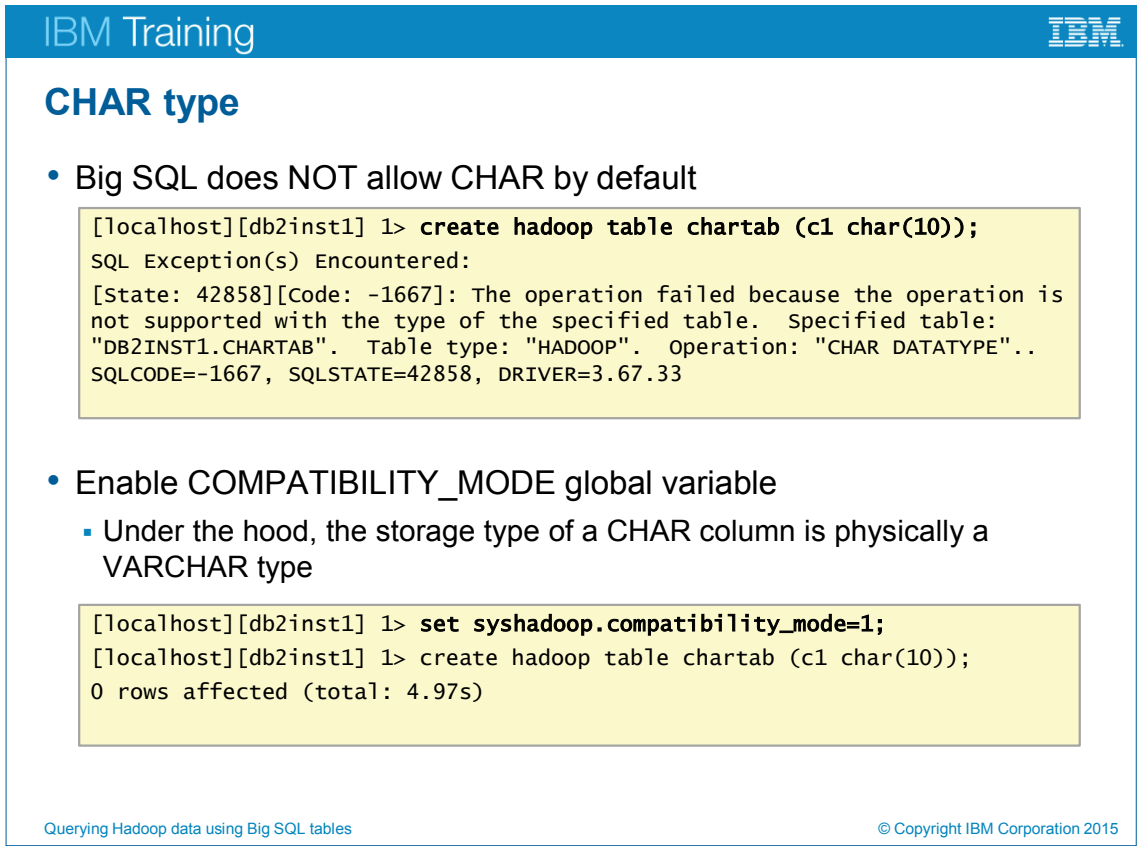
└──────────────────────────

┌ Reminder ──────────────────

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.



CHAR type

Big SQL does not allow CHAR, by default. If you try to create table with a CHAR column, you will get an error.

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.

The storage type of the CHAR column is physically a VARCHAR type.

IBM Training

DATE type

- The DATE type is defined in Hive as a TIMESTAMP
  - This means data files with DATE values must be defined with a full time

1

```
CREATE EXTERNAL HADOOP TABLE dttab
  (c1 int, c2 date)
    ROW FORMAT DELIMITED
    FIELDS TERMINATED BY ','
    LOCATION '/path/to/dttab.csv';
```

2

```
$ hadoop fs –cat
/path/to/dttab.csv
1,1997-12-15 11:32:21
2,2011-12-01 00:00:00.000
3,2014-09-10 00:00:00.000
4,2008-04-21
```

3

```
+---+---------------+
| 1 | 1997-12-15    |
+---+---------------+
| 2 | 2011-12-01    |
+---+---------------+
| 3 | 2014-09-10    |
+---+---------------+
| 4 | NULL          |
+---+---------------+
```

- During all implicit conversions to DATE, the time portion of the date is discarded

Querying Hadoop data using Big SQL tables

DATE type

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. For #1, you can see the CREATE command to create an external Hadoop table with a DATE column. On #2, you can see that you have a file with a list of dates with a blank time, except the fourth row, containing only the date. If you query it, with the results shown in #3, you get a NULL back.

The right way to do this is to append the time value to the date. When you query that, you get the appropriate date values back.

The time portion is discarded during implicit conversion of the DATE type.

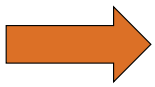
IBM Training

REAL and FLOAT types

- REAL is a 32-bit IEEE floating point
- FLOAT is a synonym for DOUBLE (64-bit IEEE floating point)
- Hive **FLOAT** ➔ Big SQL **REAL**

Hive

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



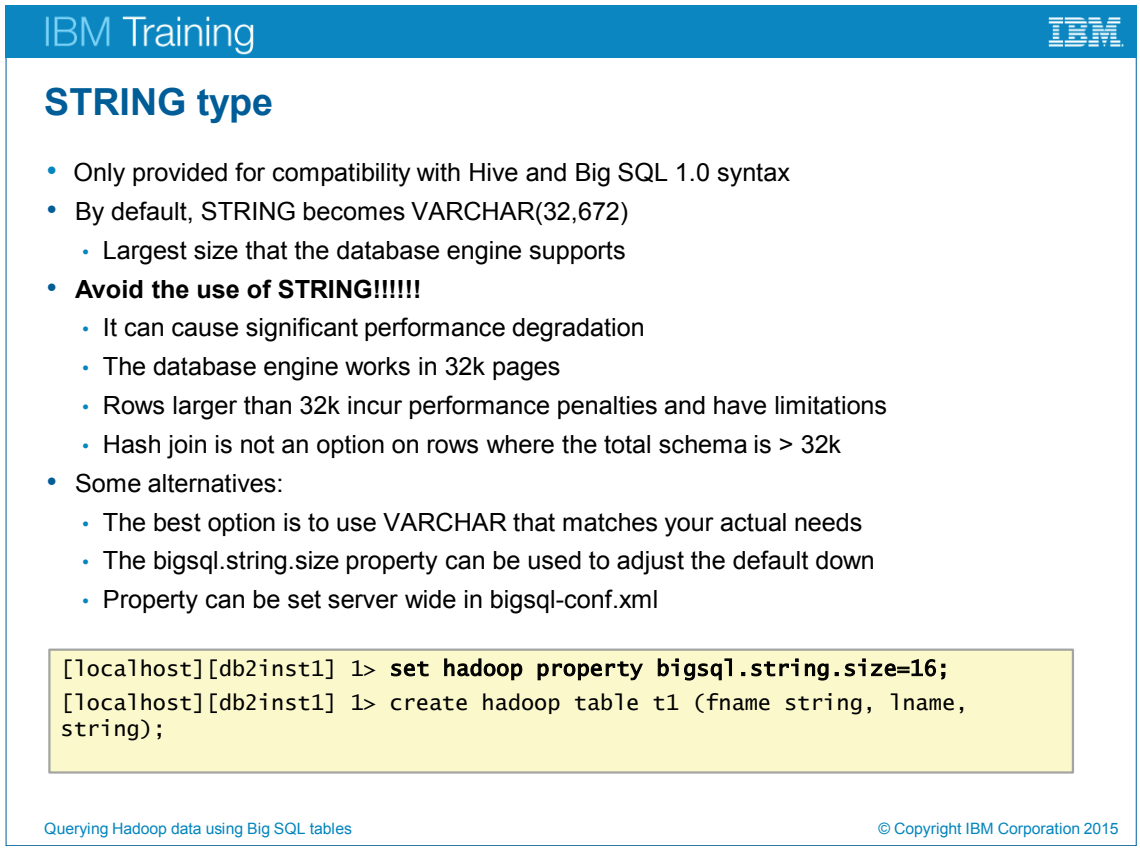
Big SQL

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

Querying Hadoop data using Big SQL tables

REAL and FLOAT types

In Big SQL, REAL is a 32-bit IEEE floating point. FLOAT is essentially a DOUBLE, a 64-bit IEEE floating point. This means that if you want to have the same data type in Big SQL from your Hive applications, you should change from FLOAT to REAL to continue to have 32-bit value. Otherwise, it will be created as DOUBLE, a 64-bit value.



STRING type

Now we get to the STRING type. 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.

IBM Training

SQL capability highlights

- Query operations
  - Projections, restrictions
  - UNION, INTERSECT, EXCEPT
  - Wide range of built-in functions (e.g. OLAP)
- 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
- Stored procedures, UDFs
  - DB2 compatible PL/SQL support
  - Cursors, flow of control (if/then/else, error handling, …), etc.

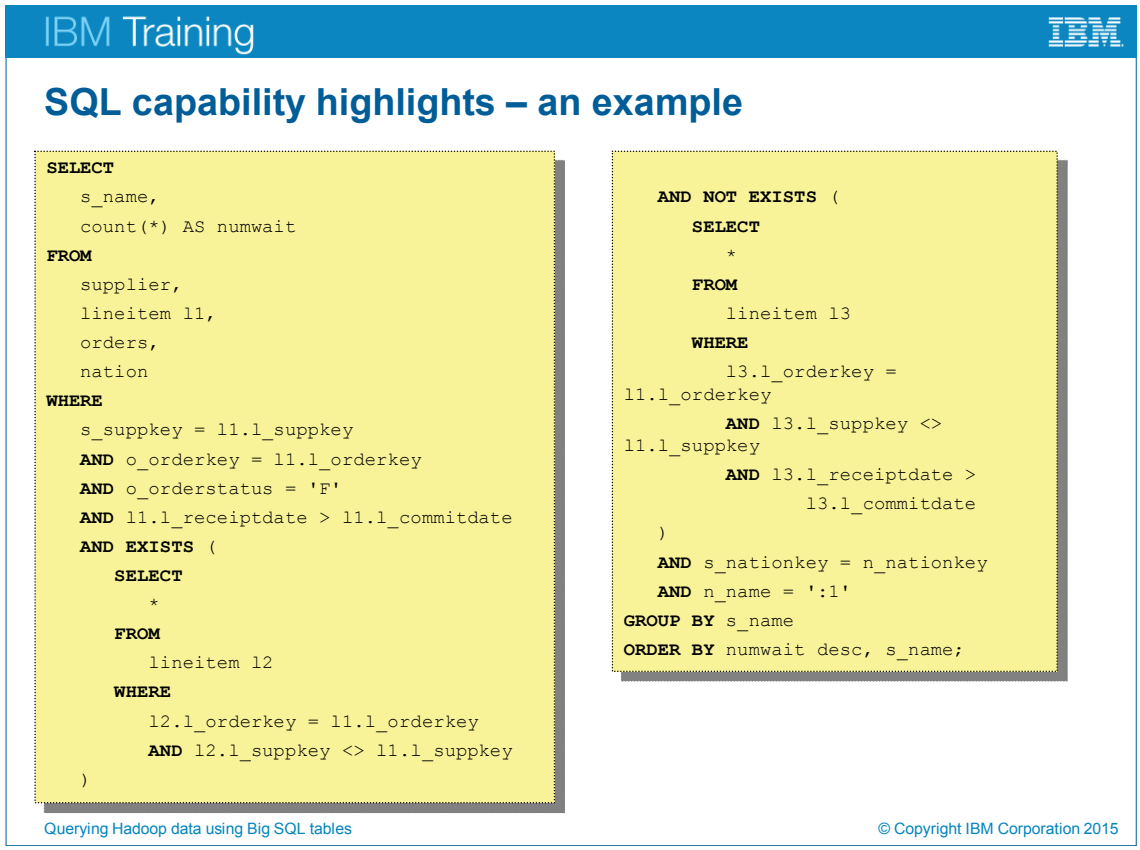
Querying Hadoop data using Big SQL tables

SQL capability highlights

The basic relational operators associated with SQL are all supported by Big SQL, as shown here on this chart. Many more sophisticated query operations are supported, too, including various types of joins, correlated and uncorrelated subqueries, PL/SQL, OLAP functions, and more. Take a look at the query at right - it contains a variety of SQL expressions that you'd expect to find in any commercial RDBMS. But some SQL-on-Hadoop implementations can't run this query because it includes SQL expressions that they don't support, such as non-equi joins, subqueries in the WHERE clause, etc.

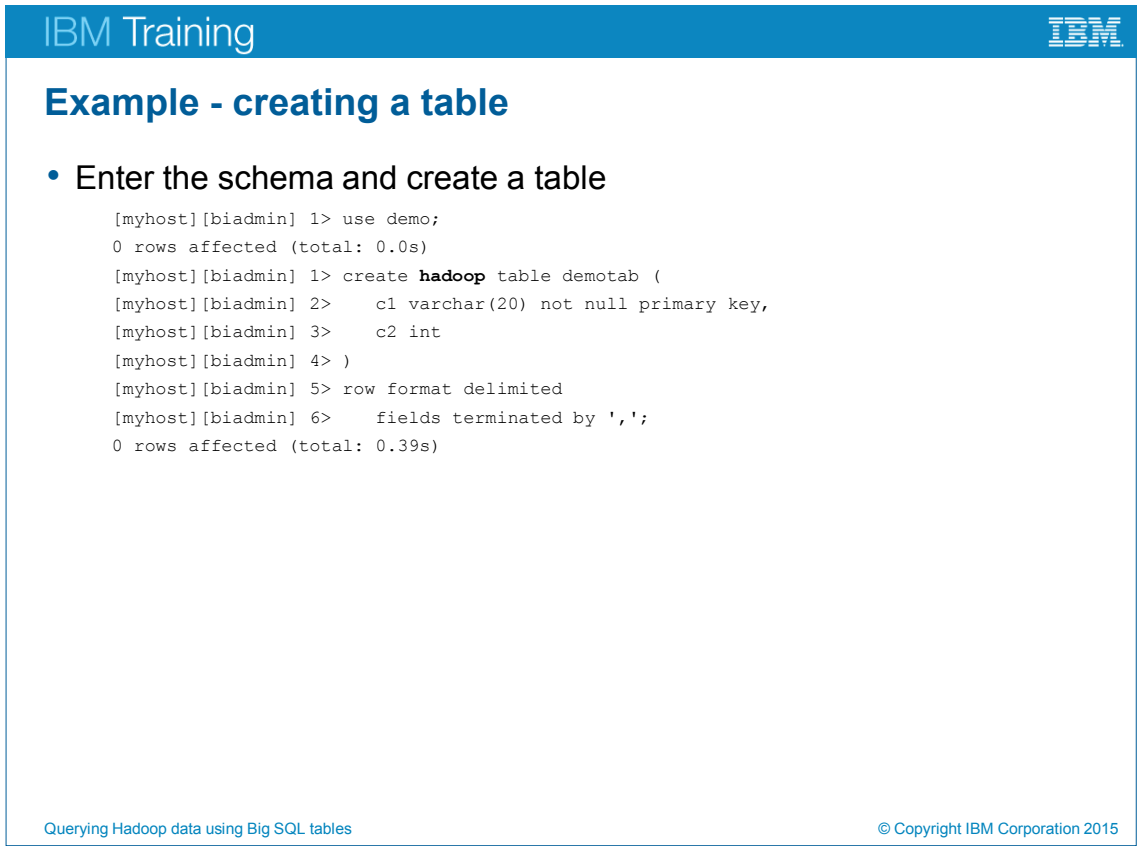
Note that Big SQL is not case sensitive. In other words, the following statements are all equivalent:

```
SELECT col1, col2 FROM t1;
select col1, col2 from t1;
SELECT COL1, COL2 FROM T1;
```



SQL capability highlights - an example

As you can see, this is very much just like any other standard SQL query. If you have experience with SQL, you can easily adopt Big SQL for your Hadoop data.



Example - creating a table

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.

┌ Important ──────────────────

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, */app/hive/warehouse/demo.db/demotab*.

Information

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 ( … )
```

IBM Training

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

```
[biadmin@myhost ~]$ :~$ hdfs dfs -cat \
   /apps/hive/warehouse/demo.db/demotab/i_1403110143227_49_201406190358948_0
foo,10
bar,20
baz,\N
```

Querying Hadoop data using Big SQL tables

Example - inserting data

Here is an example of inserting some data.

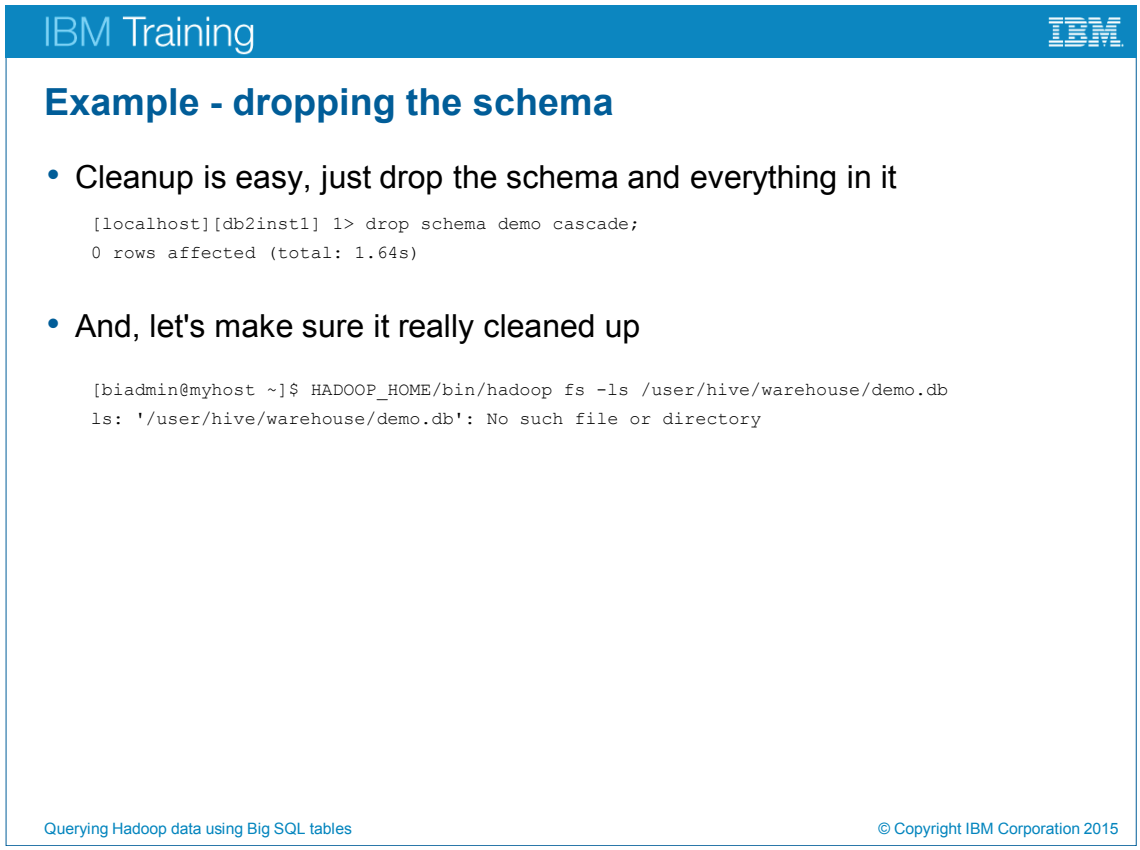
Note

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 representation if you wish.

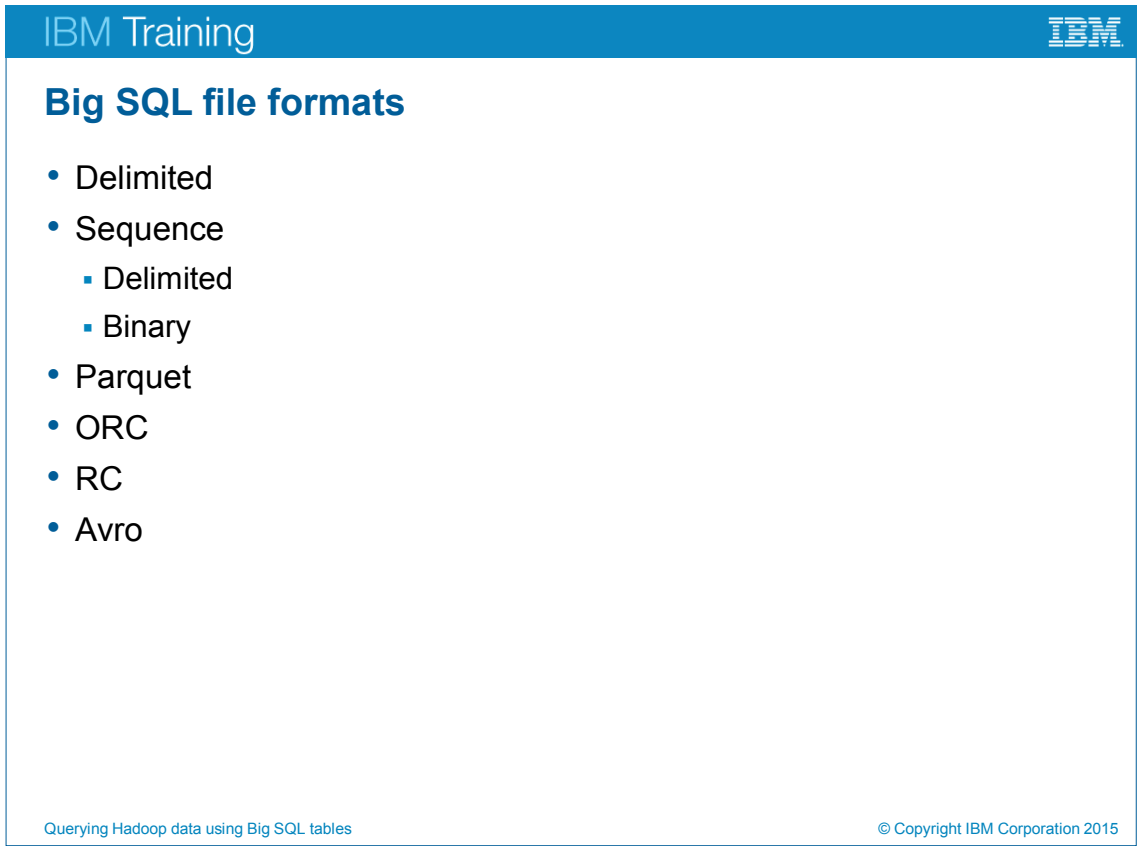


Example - dropping the schema

Cleanup is simple. This will remove the schema as well as all tables. You just execute the statement:

drop schema demo cascade;

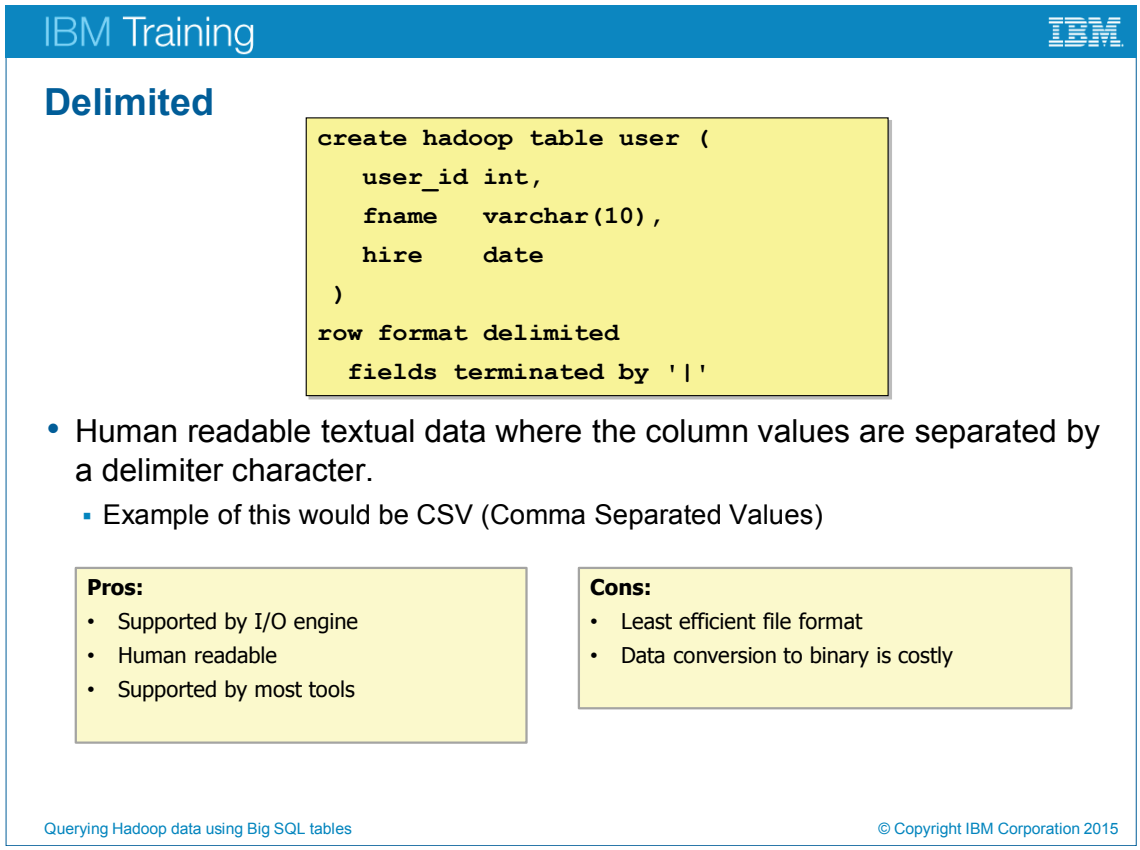
To make sure it is really gone, just go back to do a listing of the demo.db directory.



Big SQL file formats

Here are the Big SQL file formats that will be covered in detail in the following slides.

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



Delimited

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

IBM Training

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'

Querying Hadoop data using Big SQL tables

Delimited table syntax

The TERMINATED BY clause allows you to specify the type of delimiter your file uses. The default delimiter is the ASCII 0x01 (CRTL-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.

IBM Training

Delimited file format (1 of 3)

- 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

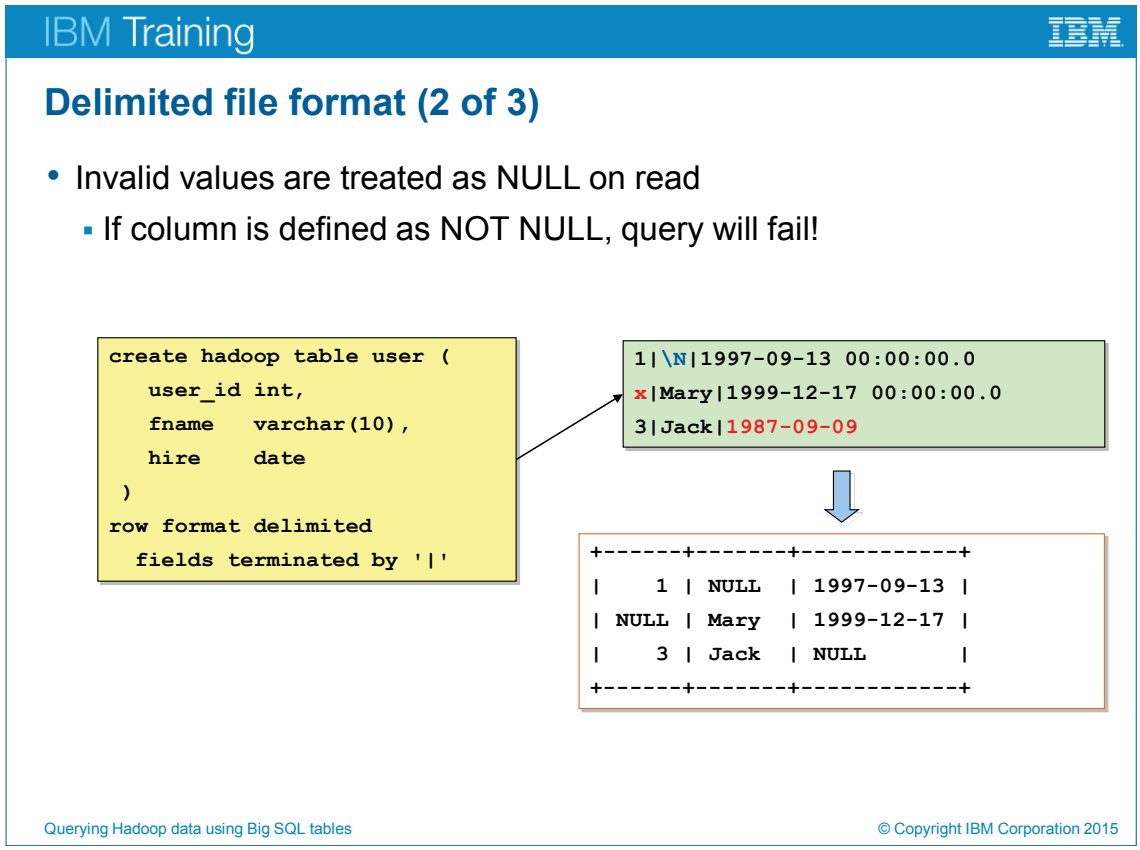
Type	Format	Type	Format
tinyint-bigint	-12, 12, +12	**boolean**	true, false, TRUE, FALSE
float, real, double	-12.12, +12.12, 12.12, 12e-3, etc.	**timestamp**	yyyy-mm-dd hh:mm:ss.fffff
string, [var]char	any	**date**	yyyy-mm-dd 00:00:00.0

Remember: Hive is told our DATE is a TIMESTAMP!

Querying Hadoop data using Big SQL tables

Delimited file format (1 of 3)

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.



Delimited file format (2 of 3)

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 as 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.

IBM Training

Delimited file format (3 of 3)

- 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 |
+---+--------------+---+
```

Querying Hadoop data using Big SQL tables

Delimited file format (3 of 3)

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 quotations 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.

┌ Example ──────────────────

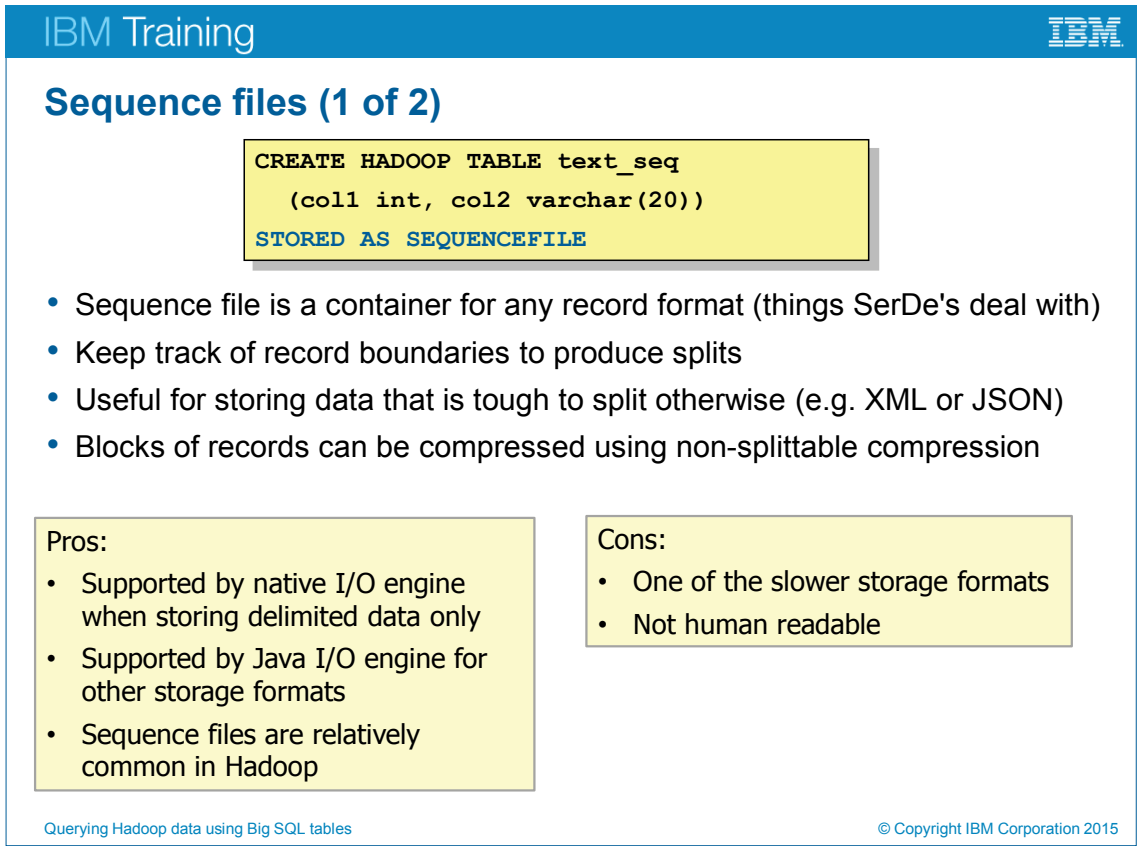
```
create hadoop table messages
(
  msg_id  int,
  msg     varchar(200),
  msg_sev int
)
row format delimited
  fields terminated by ',' escaped by '\\'
```

└──────────────────────────

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.

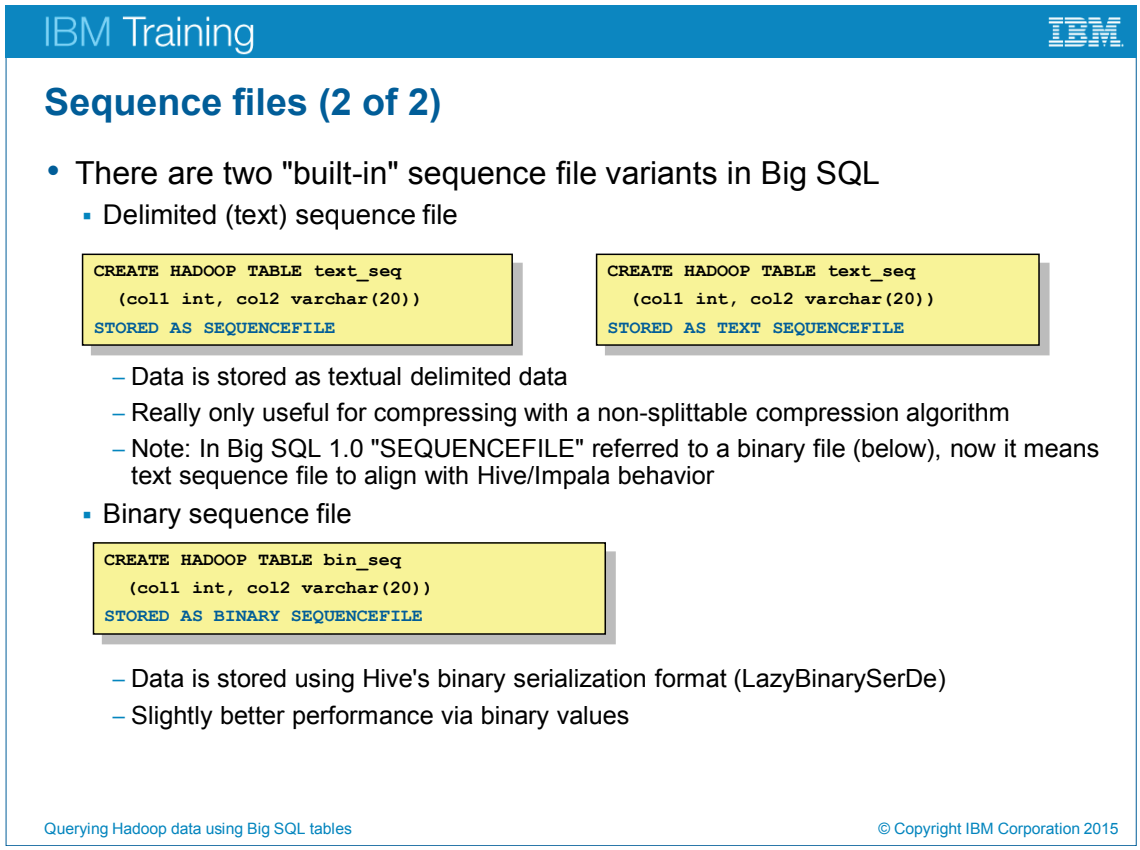


Sequence files (1 of 2)

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. 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 (2 of 2)

Sequence files comes in two flavors. You have the delimited (text) sequence file and the binary sequence file.

Example

```
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;
```

The delimited sequence file stores the data as textual delimited data. This is only useful for compressing with non-splittable compression algorithms.

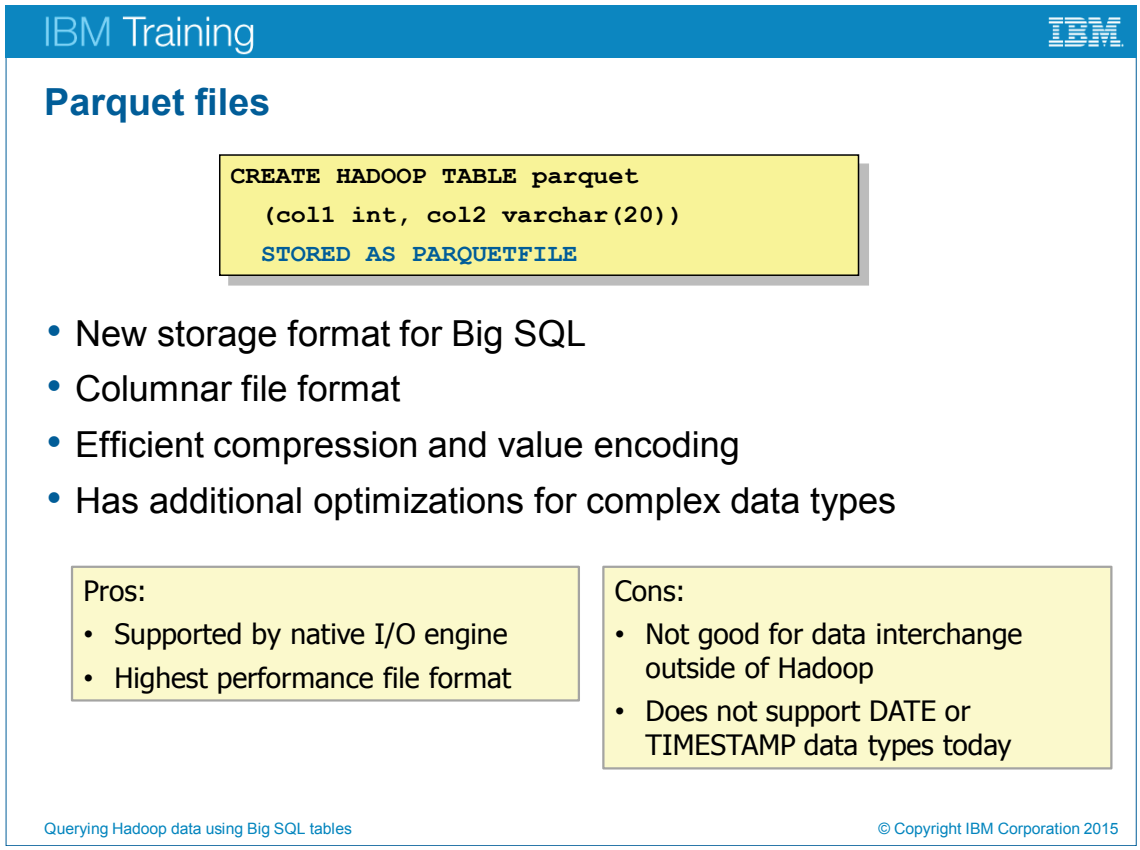
Note

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.

Example

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



Parquet files

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.

Example

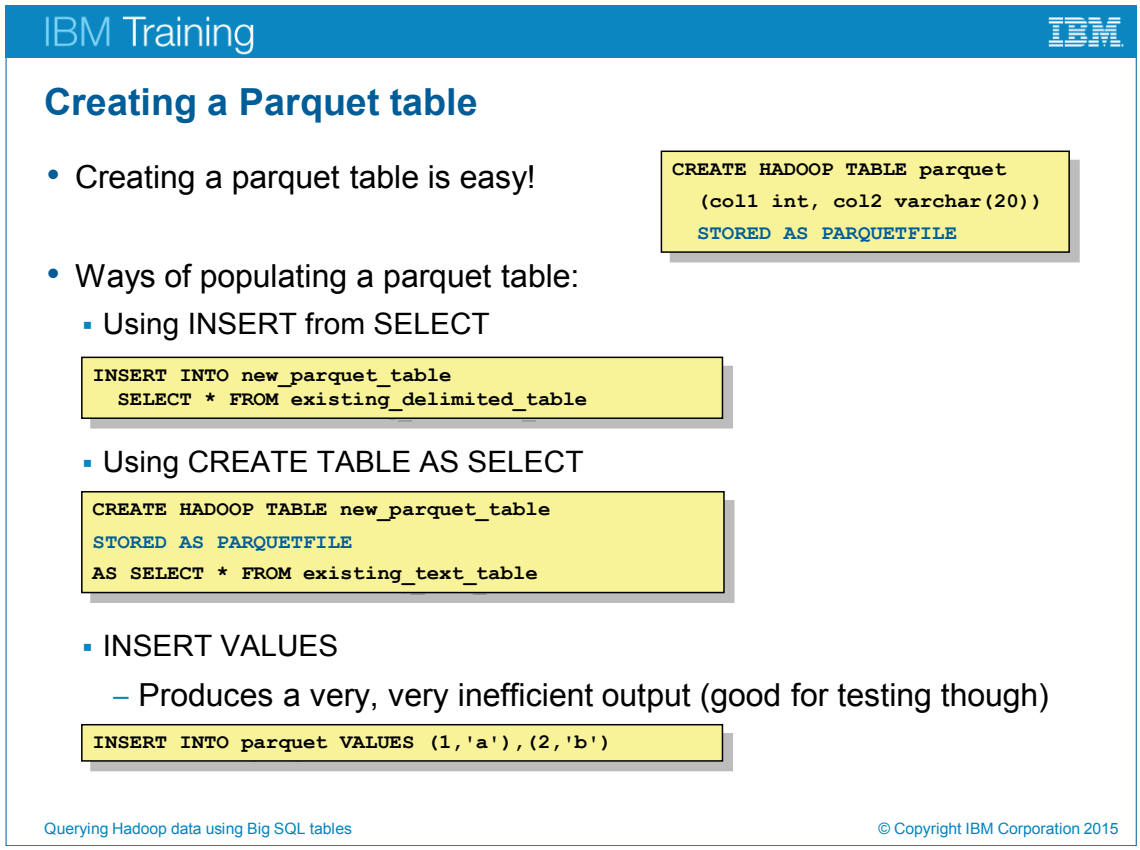
```
CREATE HADOOP TABLE parquet (col1 int, col2 varchar(20))
STORED AS PARQUETFILE;
```

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.



Creating a Parquet table

Here are some examples of how to work with Parquet tables. You saw how to create a parquet table in the previous slide.

┌ Example ──────────────────

```
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.

┌ Example ──────────────────

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

└──────────────────────────

2. Using a CREATE TABLE AS SELECT

┌ Example ──────────────────

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

└──────────────────────────

3. Using INSERT to test out your queries.

┌ Example ──────────────────

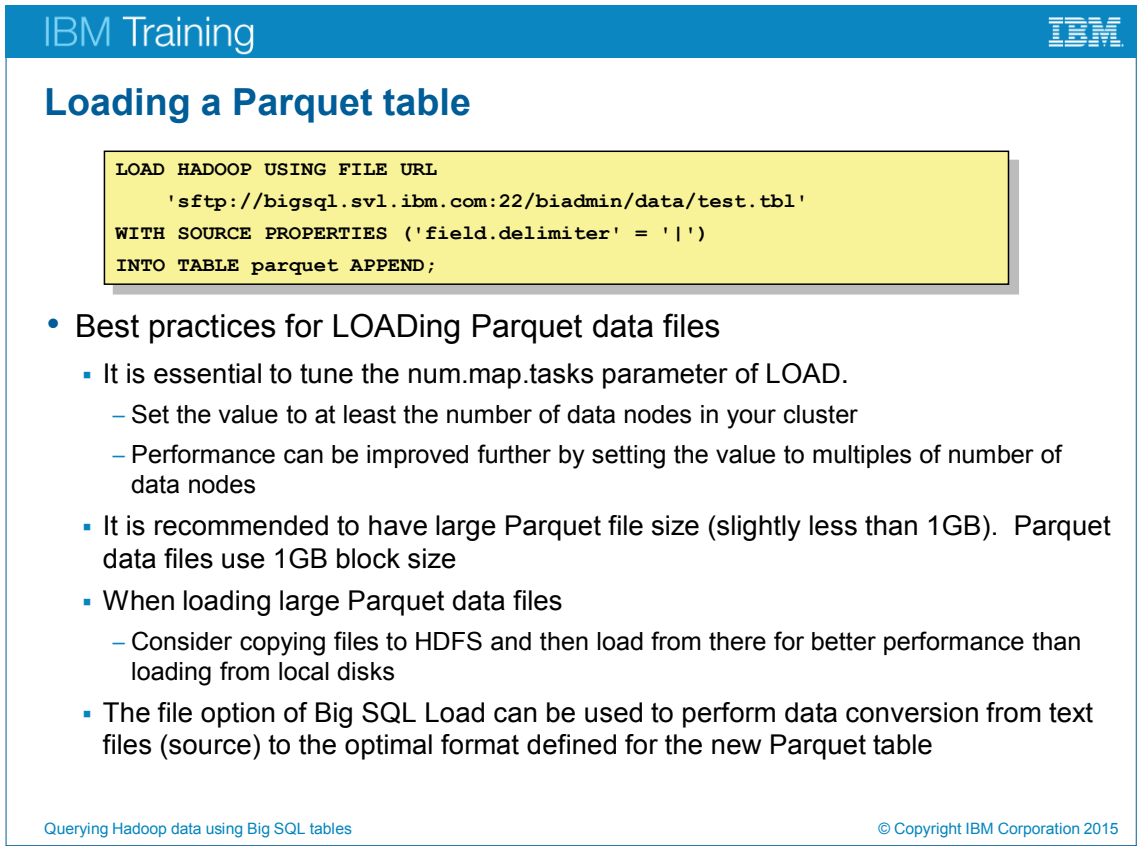
```
INSERT INTO parquet VALUES (1, 'a'), (2, 'b');
```

└──────────────────────────

┌ Reminder ──────────────────

INSERT is only good for testing. It is highly inefficient to be using for production.

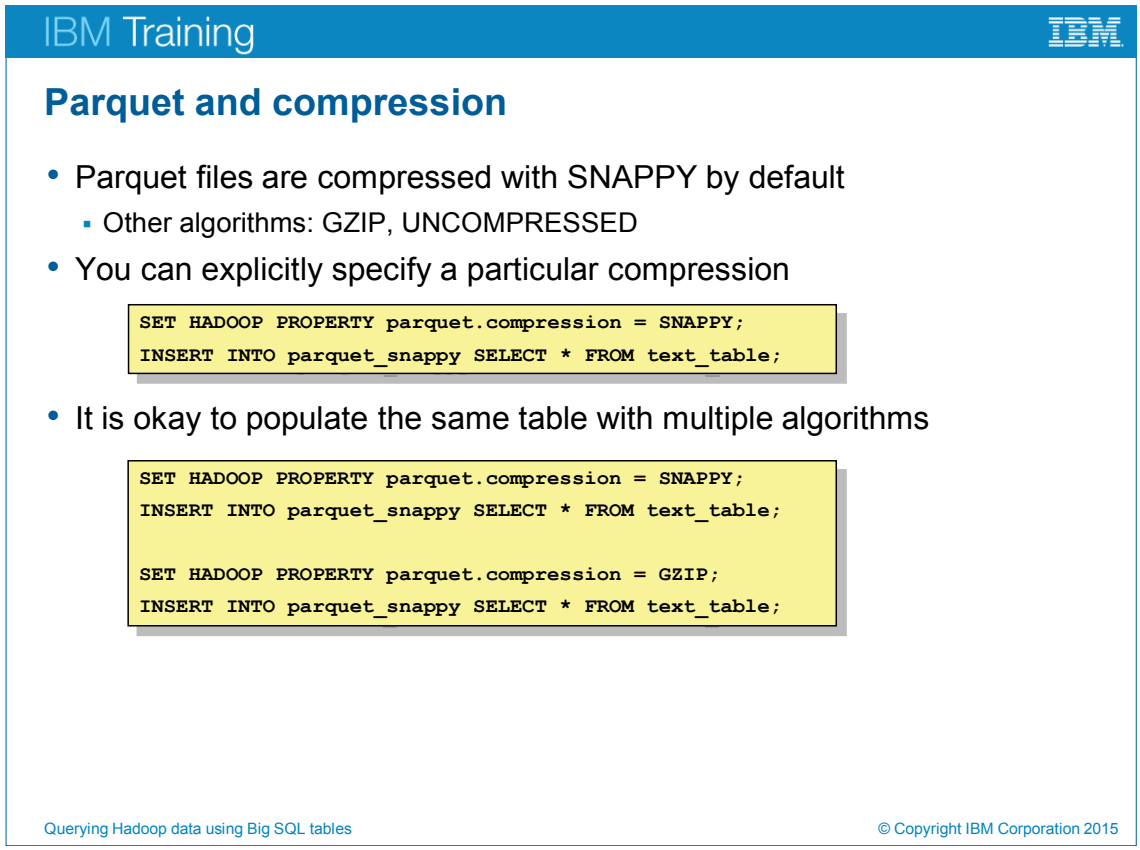
└──────────────────────────



Loading a Parquet table

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.



Parquet and compression

A few notes of 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.

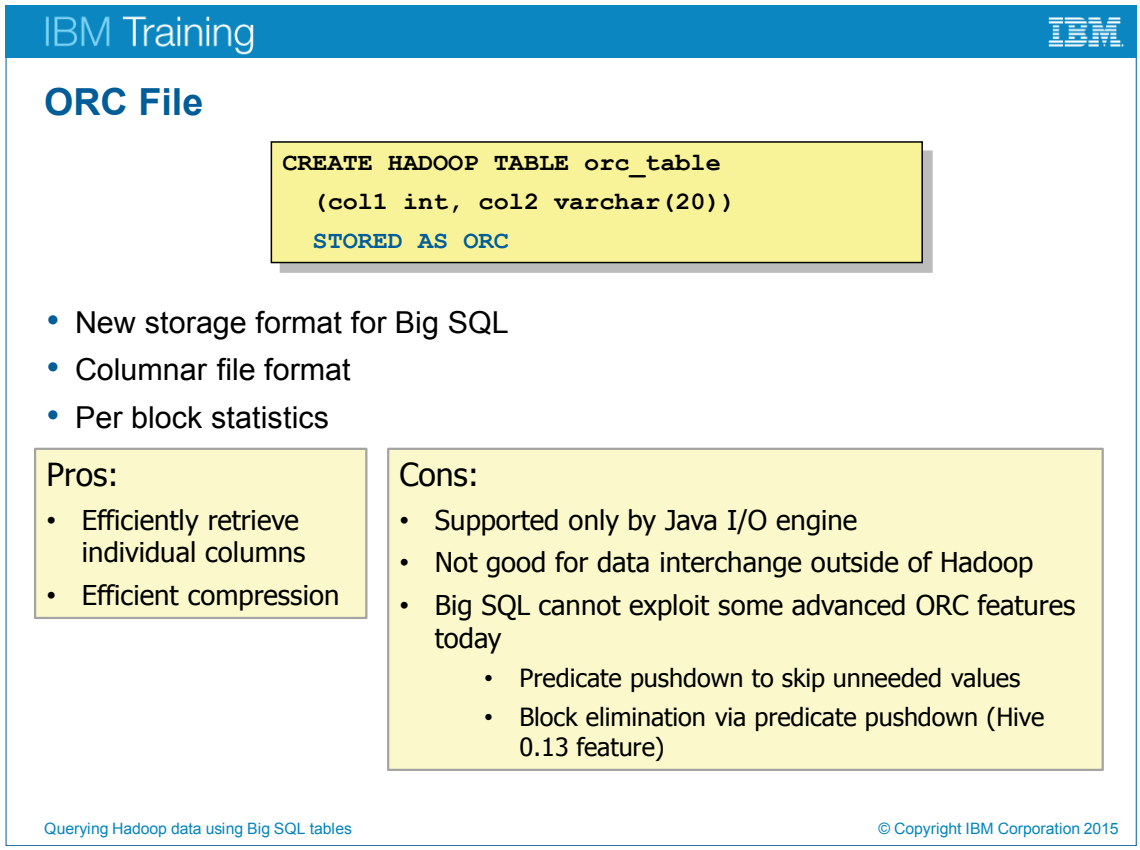
Example

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

It is also valid to populate the same table with multiple algorithms. Just set the property before each insert or load operation.

Example

```
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

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.

Example

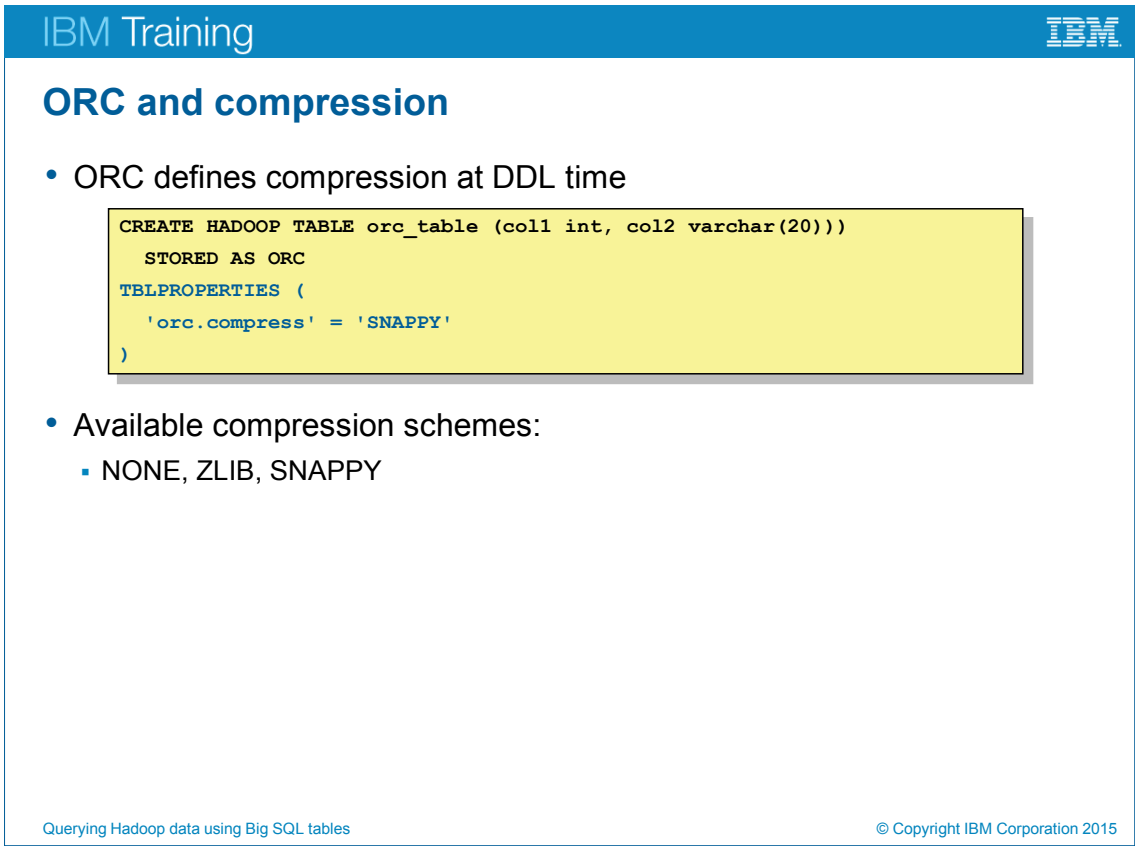
```
CREATE HADOOP TABLE orc_table  (col1 int, col2 varchar(20))
STORED AS ORC;
```

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.

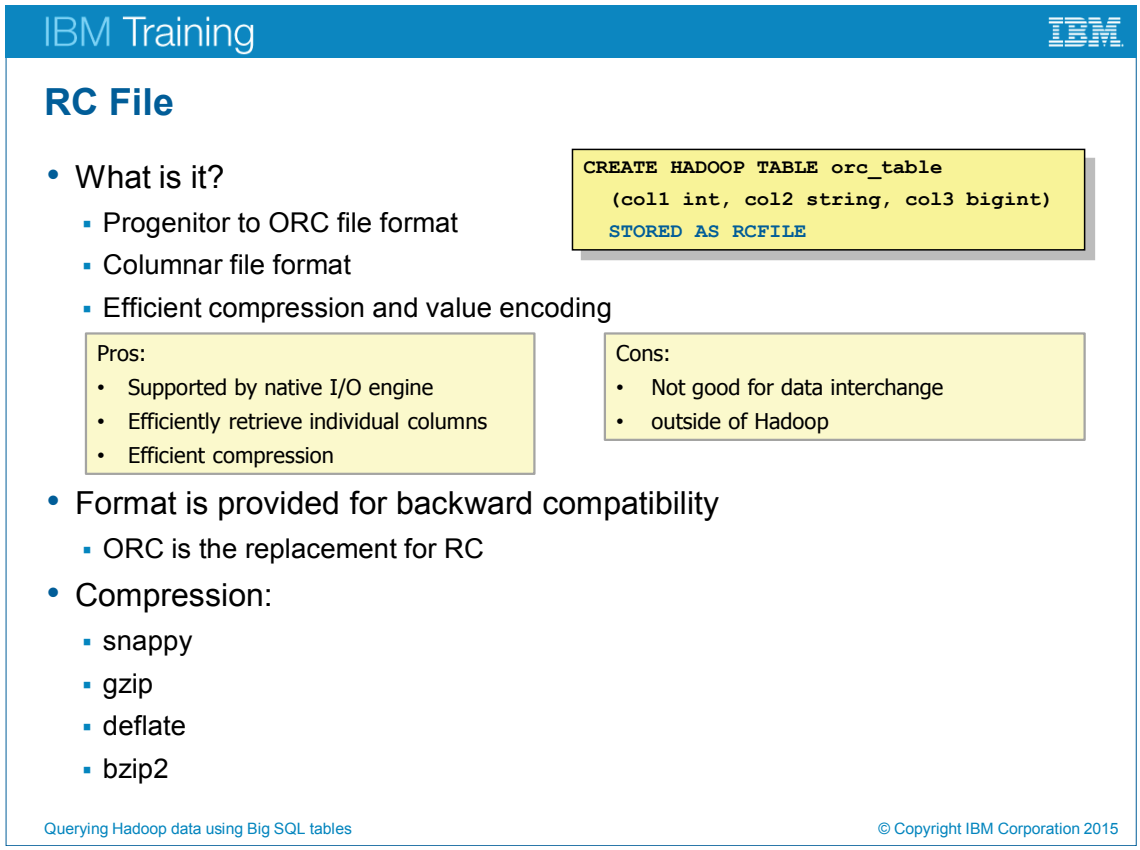


ORC and compression

ORC defines compression at DDL time. You specify it in the TBLPROPERTIES clause. The available compression schemes are NONE, ZLIB, or SNAPPY.

Example

```
CREATE HADOOP TABLE orc_table (col1 int, col2 varchar(20)))
STORED AS ORC TBLPROPERTIES ( 'orc.compress' = 'SNAPPY');
```



RC File

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:

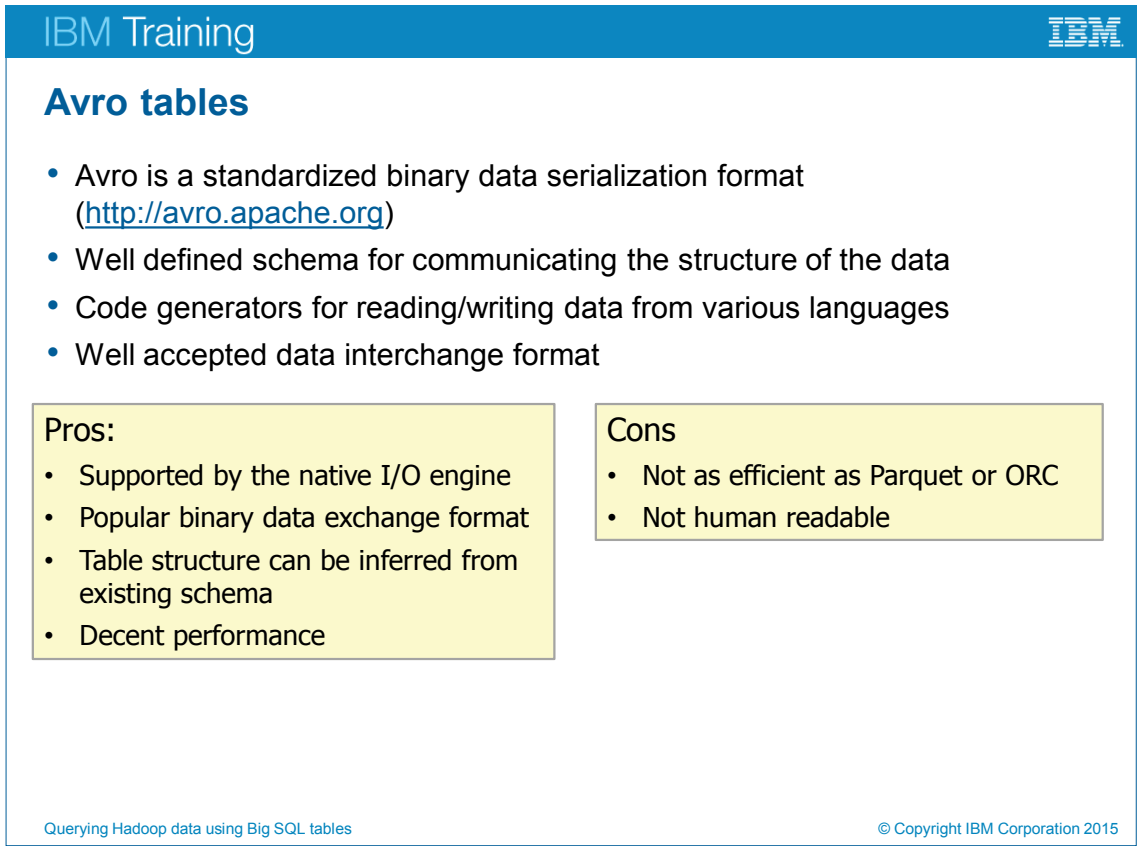
- Not good for data exchange outside of Hadoop.

There is a backward compatibility for RC with ORC.

The following compression algorithms are supported: snappy, gzip, deflate, bzip2.

Example

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



Avro tables

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.

IBM Training

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

Querying Hadoop data using Big SQL tables

Creating an Avro Table - inline schema

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.

Example

```
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" },
    ] }'
);
```

IBM Training

Creating an Avro Table – external schema

- Or, you can point to an external Avro schema file:

```
CREATE HADOOP TABLE avro_external
  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'
    );
```

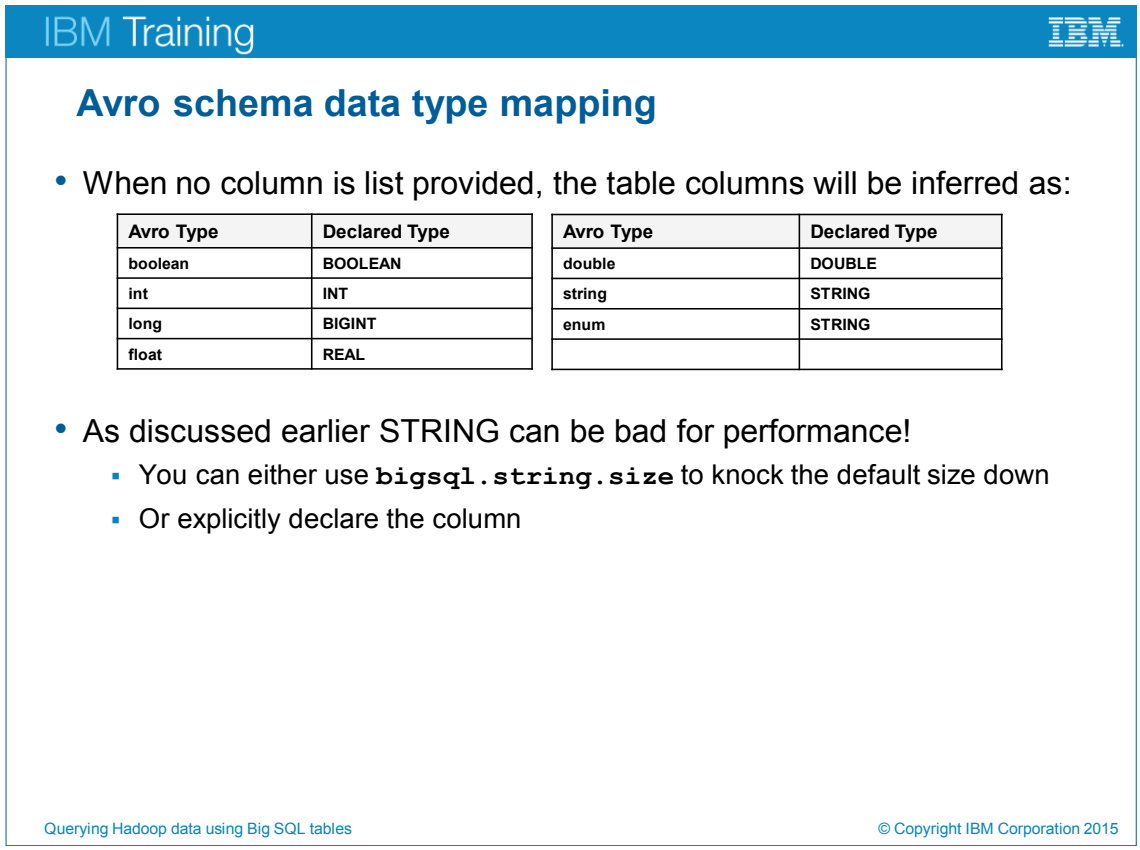
Querying Hadoop data using Big SQL tables

Creating an Avro Table - external schema

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.

Example

```
CREATE HADOOP TABLE avro_external
  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'
    );
```



Avro schema data type mapping

When no column list is specified, the table columns will be a inferred as shown in the table.

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

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.

IBM Training

Avro schema - example

```
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'
    );
```


Avro schema - example

IBM Training

Checkpoint

1. What is the recommended method for getting data into your Big SQL table for best performance?
2. What are the three categories of data type?
3. Should you use the default STRING data type?
4. What should you include in each of the DATE values?
5. List the various file storage formats.
6. Which file format has the highest performance?

Checkpoint

IBM Training

Checkpoint solutions

1. What is the recommended method for getting data into your Big SQL table for best performance?
   - Using the LOAD operation
2. What are the three categories of data type?
   - Declared, SQL, Hive
3. Should you use the default STRING data type?
   - No, by default, STRING is mapped to the VARCHAR (32K) which can lead to performance degradation. Recommend using the VARCHAR that you need or change the default size.
4. What should you include in each of the DATE values?
   - A time value because DATE is mapped to TIMESTAMP.
5. List the various file storage formats.
   - Delimited, Sequence (Text/Binary), Parquet, ORC, RC, Avro.
6. Which file format has the highest performance?
   - Parquet

Checkpoint solution

IBM Training

Demonstration 1

Working with Big SQL data

At the end of this demonstratrion, you will be able to:

- Create Big SQL tables that use Hadoop text file and Parquet file formats.
- Populate Big SQL tables from local files and from the results of queries.
- Query Big SQL tables using projections, restrictions, joins, aggregations, and other popular expressions.
- Create and query a view based on multiple Big SQL tables.

Querying Hadoop data using Big SQL tables

Demonstration 1: Working with Big SQL data

Demonstration 1: Working with Big SQL data

Purpose:
Now you're ready to query your tables. Based on earlier demonstrations, you've already seen that you can perform basic SQL operations.

In this demonstration, you will create and run Big SQL queries that join data from multiple tables as well as perform aggregations and other SQL operations. Note that the queries included in this section are based on queries shipped with Big SQL client software as samples.

Estimated time: 60 minutes

User/Password: **biadmin/biadmin**

root/dalvm3

Services Password: **ibm2blue**

Task 1. Loading data into Big SQL tables.

Use either JSqsh or the Big SQL service via Ambari.

Load data into each of following tables using sample data provided. If necessary, change the SFTP and file path specifications in each of the following examples to match your environment. LOAD returns a **warning** message providing details on the number of rows loaded, etc.

The 2_Load Statements.sql file is located in the */home/biadmin/labfiles/bigsql/BIG_SQL_Data_Analysis* folder.

1. Issue the first LOAD statement and verify that the operation completed successfully.

```
load hadoop using file url

'sftp://bigsql:ibm2blue@ibmclass.localdomain:22/usr/ibmp
acks/bigsql/4.0/bigsql/samples/data/GOSALESDW.GO_REGION_
DIM.txt' with SOURCE PROPERTIES ('field.delimiter'='\t')
INTO TABLE GO_REGION_DIM overwrite;
```

Each example loads data into a table using a file URL specification that relies on SFTP to locate the source file. In particular, the SFTP specification includes a valid user ID and password (yourID/yourPassword), the target host server and port (ibmclass.localdomain:22), and the full path of the data file on that system.

The WITH SOURCE PROPERTIES clause specifies that fields in the source data are delimited by tabs (“\t”). The INTO TABLE clause identifies the target table for the LOAD operation.
The OVERWRITE keyword indicates that any existing data in the table will be replaced by data contained in the source file. (If you wanted to simply add rows to the table’s content, you could specify APPEND instead.)

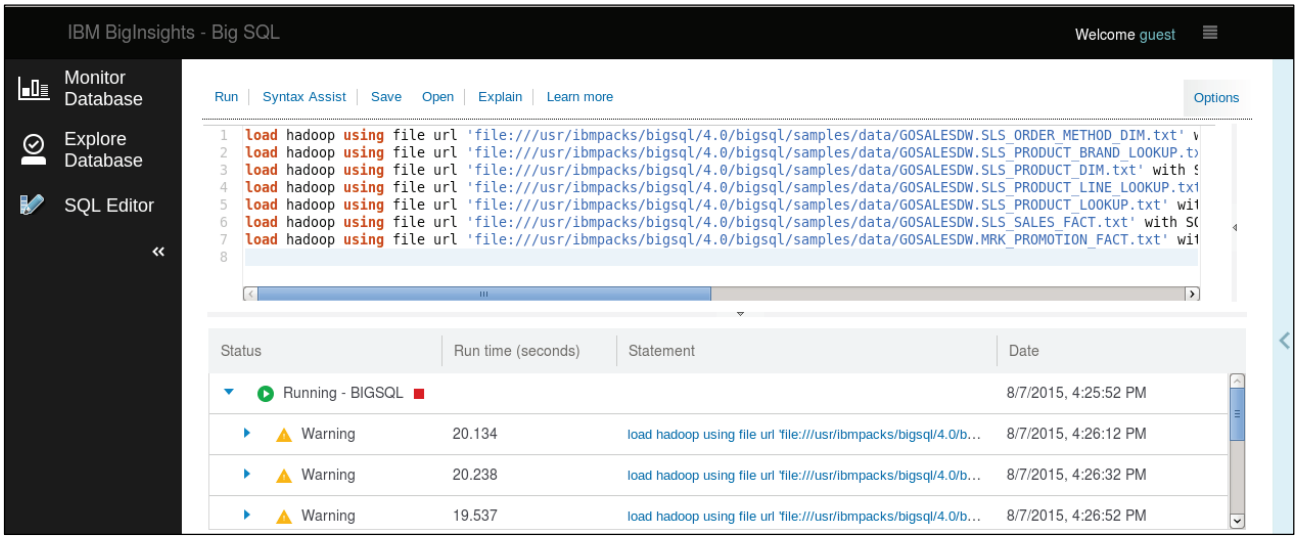
Using SFTP (or FTP) is one way in which you can invoke the LOAD command.

If your target data already resides in your distributed file system, you can provide the DFS directory information in your file URL specification. Indeed, for optimal runtime performance, you may prefer to take that approach. The remaining LOAD statements refers to the file locally, instead of using SFTP

2. Issue each LOAD statement and verify that the operation completed successfully.

```
load hadoop using file url
'file:///usr/ibmpacks/bigsql/4.0/bigsql/samples/data/GOS
ALESDW.SLS_ORDER_METHOD_DIM.txt' with SOURCE PROPERTIES
('field.delimiter'='\t') INTO TABLE SLS_ORDER_METHOD_DIM
overwrite;

load hadoop using file url
'file:///usr/ibmpacks/bigsql/4.0/bigsql/samples/data/GOS
ALESDW.SLS_PRODUCT_BRAND_LOOKUP.txt' with SOURCE
PROPERTIES ('field.delimiter'='\t') INTO TABLE
SLS_PRODUCT_BRAND_LOOKUP overwrite;

load hadoop using file url
'file:///usr/ibmpacks/bigsql/4.0/bigsql/samples/data/GOS
ALESDW.SLS_PRODUCT_DIM.txt' with SOURCE PROPERTIES
('field.delimiter'='\t') INTO TABLE SLS_PRODUCT_DIM
overwrite;

load hadoop using file url
'file:///usr/ibmpacks/bigsql/4.0/bigsql/samples/data/GOS
ALESDW.SLS_PRODUCT_LINE_LOOKUP.txt' with SOURCE
PROPERTIES ('field.delimiter'='\t') INTO TABLE
SLS_PRODUCT_LINE_LOOKUP overwrite;
```

```
load hadoop using file url
'file:///usr/ibmpacks/bigsql/4.0/bigsql/samples/data/GOS
ALESDW.SLS_PRODUCT_LOOKUP.txt' with SOURCE PROPERTIES
('field.delimiter'='\t') INTO TABLE SLS_PRODUCT_LOOKUP
overwrite;

load hadoop using file url
'file:///usr/ibmpacks/bigsql/4.0/bigsql/samples/data/GOS
ALESDW.SLS_SALES_FACT.txt' with SOURCE PROPERTIES
('field.delimiter'='\t') INTO TABLE SLS_SALES_FACT
overwrite;

load hadoop using file url
'file:///usr/ibmpacks/bigsql/4.0/bigsql/samples/data/GOS
ALESDW.MRK_PROMOTION_FACT.txt' with SOURCE PROPERTIES
('field.delimiter'='\t') INTO TABLE MRK_PROMOTION_FACT
overwrite;
```



3. Query the tables to verify that the expected number of rows was loaded into each table. Execute each query that follows individually and compare the results with the number of rows specified in the comment line preceding each query. You will need to expand each of the **Succeeded** lines to see the results.

   The 3_SELECT statements - validating LOAD.sql file is located in the */home/biadmin/labfiles/bigsql/BIG_SQL_Data_Analysis* folder.

```
-- total rows in GO_REGION_DIM = 21
select count(*) from GO_REGION_DIM;

-- total rows in sls_order_method_dim = 7
select count(*) from sls_order_method_dim;

-- total rows in SLS_PRODUCT_BRAND_LOOKUP = 28
select count(*) from SLS_PRODUCT_BRAND_LOOKUP;

-- total rows in SLS_PRODUCT_DIM = 274
select count(*) from SLS_PRODUCT_DIM;

-- total rows in SLS_PRODUCT_LINE_LOOKUP = 5
select count(*) from SLS_PRODUCT_LINE_LOOKUP;

-- total rows in SLS_PRODUCT_LOOKUP = 6302
select count(*) from SLS_PRODUCT_LOOKUP;

-- total rows in SLS_SALES_FACT = 446023
select count(*) from SLS_SALES_FACT;

-- total rows gosalesdw.MRK_PROMOTION_FACT = 11034
select count(*) from MRK_PROMOTION_FACT;
```

Task 2. Querying the data with Big SQL.

1. Join data from multiple tables to return the product name, quantity and order method of goods that have been sold. For simplicity, limit the number of returns rows to 20. To achieve this, execute the following query:

   The 4_SELECT statements - querying data.sql file is located in the */home/biadmin/labfiles/bigsql/BIG_SQL_Data_Analysis* folder.

```
--Fetch the product name, quantity, and order method of
products sold.
--
--Query 1
SELECT pnumb.product_name, sales.quantity,
meth.order_method_en
FROM
sls_sales_fact sales,
sls_product_dim prod,
sls_product_lookup pnumb,
sls_order_method_dim meth
WHERE
pnumb.product_language='EN'
AND sales.product_key=prod.product_key
AND prod.product_number=pnumb.product_number
AND meth.order_method_key=sales.order_method_key
fetch first 20 rows only;
```

   Data from four tables will be used to drive the results of this query (see the tables referenced in the FROM clause). Relationships between these tables are resolved through 3 join predicates specified as part of the WHERE clause. The query relies on 3 equi-joins to filter data from the referenced tables. (Predicates such as prod.product_number=pnumb.product_number help to narrow the results to product numbers that match in two tables.)

   For improved readability, this query uses aliases in the SELECT and FROM clauses when referencing tables. For example, pnumb.product_name refers to “pnumb,” which is the alias for the gosalesdw.sls_product_lookup table. Once defined in the FROM clause, an alias can be used in the WHERE clause so that you do not need to repeat the complete table name.

   The use of the predicate and pnumb.product_language='EN' helps to further narrow the result to only English output. This database contains thousands of rows of data in various languages, so restricting the language provides some optimization.

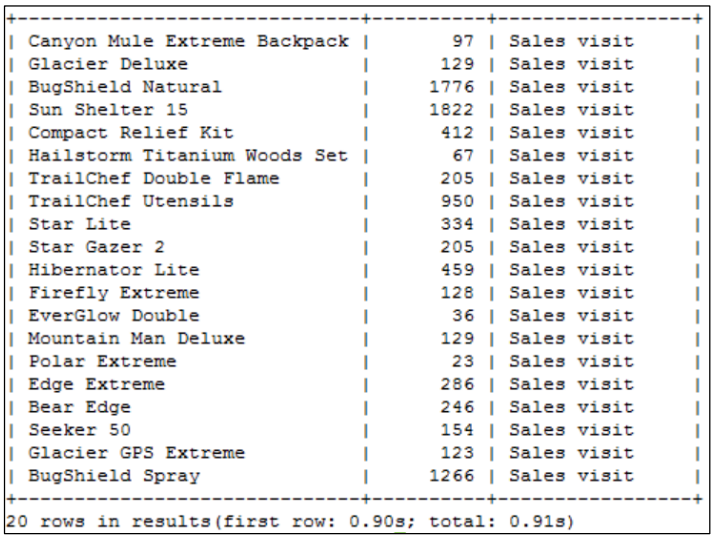
Results are shown from JSqsh, but you can use the Big SQL service via Ambari if you wish.

```
+-----------------------------+----------+-----------------+
| Compact Relief Kit          |      313 | Sales visit     |
| Course Pro Putter           |      587 | Telephone       |
| Blue Steel Max Putter       |      214 | Telephone       |
| Course Pro Gloves           |      576 | Telephone       |
| Glacier Deluxe              |      129 | Sales visit     |
| BugShield Natural           |     1776 | Sales visit     |
| Sun Shelter 15              |     1822 | Sales visit     |
| Compact Relief Kit          |      412 | Sales visit     |
| Hailstorm Titanium Woods Set |       67 | Sales visit     |
| Canyon Mule Extreme Backpack |       97 | E-mail          |
| TrailChef Canteen           |     1172 | Telephone       |
| TrailChef Cook Set          |      591 | Telephone       |
| TrailChef Deluxe Cook Set   |      338 | Telephone       |
| Star Gazer 3                |       97 | Telephone       |
| Hibernator                  |      364 | Telephone       |
| Hibernator Camp Cot         |      234 | Telephone       |
| Canyon Mule Cooler          |      603 | Telephone       |
| Firefly 4                   |      232 | Telephone       |
| EverGlow Single             |      450 | Telephone       |
| EverGlow Kerosene           |      257 | Telephone       |
+-----------------------------+----------+-----------------+
20 rows in results(first row: 1.2s; total: 1.2s)
```

2. Modify the query to restrict the order method to one type – those involving a Sales visit. To do so, add the following query predicate just before the FETCH FIRST 20 ROWS clause: AND order_method_en='Sales visit'

```
--Fetch the product name, quantity, and order method
--of products sold through sales visits.
--Query 2
SELECT pnumb.product_name, sales.quantity,
meth.order_method_en
FROM
sls_sales_fact sales,
sls_product_dim prod,
sls_product_lookup pnumb,
sls_order_method_dim meth
WHERE
pnumb.product_language='EN'
AND sales.product_key=prod.product_key
AND prod.product_number=pnumb.product_number
AND meth.order_method_key=sales.order_method_key
AND order_method_en='Sales visit'
FETCH FIRST 20 ROWS ONLY;
```

3. Inspect the results:



Canyon Mule Extreme Backpack	97	Sales visit
Glacier Deluxe	129	Sales visit
BugShield Natural	1776	Sales visit
Sun Shelter 15	1822	Sales visit
Compact Relief Kit	412	Sales visit
Hailstorm Titanium Woods Set	67	Sales visit
TrailChef Double Flame	205	Sales visit
TrailChef Utensils	950	Sales visit
Star Lite	334	Sales visit
Star Gazer 2	205	Sales visit
Hibernator Lite	459	Sales visit
Firefly Extreme	128	Sales visit
EverGlow Double	36	Sales visit
Mountain Man Deluxe	129	Sales visit
Polar Extreme	23	Sales visit
Edge Extreme	286	Sales visit
Bear Edge	246	Sales visit
Seeker 50	154	Sales visit
Glacier GPS Extreme	123	Sales visit
BugShield Spray	1266	Sales visit

20 rows in results(first row: 0.90s; total: 0.91s)

4. To find out which sales method of all the methods has the greatest quantity of orders, include a GROUP BY clause (group by pll.product_line_en, md.order_method_en). In addition, invoke the SUM aggregate function (sum(sf.quantity)) to total the orders by product and method. Finally, this query cleans up the output a bit by using aliases (e.g., as Product) to substitute a more readable column header.

```
--Query 3
SELECT pll.product_line_en AS Product,
md.order_method_en AS Order_method,
sum(sf.QUANTITY) AS total
FROM
sls_order_method_dim AS md,
sls_product_dim AS pd,
sls_product_line_lookup AS pll,
sls_product_brand_lookup AS pbl,
sls_sales_fact AS sf
WHERE
pd.product_key = sf.product_key
AND md.order_method_key = sf.order_method_key
AND pll.product_line_code = pd.product_line_code
AND pbl.product_brand_code = pd.product_brand_code
GROUP BY pll.product_line_en, md.order_method_en;
```

5. Inspect the results, which should contain 35 rows. A portion is shown below.

PRODUCT	ORDER_METHOD	TOTAL
Camping Equipment	E-mail	1413084
Camping Equipment	Fax	413958
Camping Equipment	Mail	348058
Camping Equipment	Sales visit	2899754
Camping Equipment	Special	203528
Camping Equipment	Telephone	2792588
Camping Equipment	Web	19230179
Golf Equipment	E-mail	333300
Golf Equipment	Fax	102651
Golf Equipment	Mail	80432
Golf Equipment	Sales visit	263788
Golf Equipment	Special	38585
Golf Equipment	Telephone	601506
Golf Equipment	Web	3693439
Mountaineering Equipment	E-mail	199214
Mountaineering Equipment	Fax	292408

Task 3. Creating and working with views.

Big SQL supports views (virtual tables) based on one or more physical tables. In this section, you will create a view that spans multiple tables. Then you'll query this view using a simple SELECT statement. In doing so, you'll see that you can work with views in Big SQL much as you can work with views in a relational DBMS.

The 5_VIEW statements.sql file is located in the */home/biadmin/labfiles/bigsql/BIG_SQL_Data_Analysis* folder.

1. Create a view named MYVIEW that extracts information about product sales featured in marketing promotions. By the way, since the schema name is omitted in both the CREATE and FROM object names, the current schema (your user name), is assumed.

```
create view myview as
select product_name, sales.product_key, mkt.quantity,
    sales.order_day_key, sales.sales_order_key,
order_method_en
from
    mrk_promotion_fact mkt,
    sls_sales_fact sales,
    sls_product_dim prod,
    sls_product_lookup pnumb,
    sls_order_method_dim meth
where mkt.order_day_key=sales.order_day_key
    and sales.product_key=prod.product_key
    and prod.product_number=pnumb.product_number
    and pnumb.product_language='EN'
    and meth.order_method_key=sales.order_method_key;
```

2. Now query the view:

```
select * from myview
order by product_key asc, order_day_key asc
fetch first 20 rows only;
```

3. Inspect the results:

PRODUCT_NAME	PRODUCT_KEY	QUANTITY	ORDER_DAY_KEY	SALES_ORDER_KEY	ORDER_METHOD_EN
TrailChef Water Bag	30001	1350	20040112	195305	Sales visit
TrailChef Water Bag	30001	663	20040112	195305	Sales visit
TrailChef Water Bag	30001	495	20040112	195305	Sales visit
TrailChef Water Bag	30001	1260	20040112	195305	Sales visit
TrailChef Water Bag	30001	965	20040112	195305	Sales visit
TrailChef Water Bag	30001	1035	20040112	195305	Sales visit
TrailChef Water Bag	30001	990	20040112	195305	Sales visit

Task 4. Populating a table with 'INSERT INTO ... SELECT'.

Big SQL enables you to populate a table with data based on the results of a query. In this demonstration, you will use an INSERT INTO . . . SELECT statement to retrieve data from multiple tables and insert that data into another table. Executing an INSERT INTO . . . SELECT exploits the machine resources of your cluster because Big SQL can parallelize both read (SELECT) and write (INSERT) operations.

The 6_INSERT INTO SELECT statements.sql file is located in the */home/biadmin/labfiles/bigsql/BIG_SQL_Data_Analysis* folder.

1. Execute the following statement to create a sample table named `sales_report`:

```
-- create a sample sales_report table
CREATE HADOOP TABLE sales_report
(
product_key       INT NOT NULL,
product_name VARCHAR(150),
quantity          INT,
order_method_en  VARCHAR(90)
)
ROW FORMAT DELIMITED
FIELDS TERMINATED BY '\t'
LINES TERMINATED BY '\n'
STORED AS TEXTFILE;
```

2. Now populate the newly created table with results from a query that joins data from multiple tables.

```
-- populate the sales_report data with results from a
query
INSERT INTO sales_report
SELECT sales.product_key, pnumb.product_name,
sales.quantity,
meth.order_method_en
FROM
sls_sales_fact sales,
sls_product_dim prod,
sls_product_lookup pnumb,
sls_order_method_dim meth
WHERE
pnumb.product_language='EN'
AND sales.product_key=prod.product_key
AND prod.product_number=pnumb.product_number
AND meth.order_method_key=sales.order_method_key
and sales.quantity > 1000;
```

3. Verify that the previous query was successful by executing the following query:

```
-- total number of rows should be 14441
select count(*) from sales_report;
```

Task 5. Storing data in an alternate file format (Parquet).

Until now, you've instructed Big SQL to use the TEXTFILE format for storing data in the tables you've created. This format is easy to read (both by people and most applications), as data is stored in a delimited form with one record per line and new lines separating individual records. It's also the default format for Big SQL tables.

However, if you'd prefer to use a different file format for data in your tables, Big SQL supports several formats popular in the Hadoop environment, including Avro, sequence files, RC (record columnar) and Parquet. While it's beyond the scope of this demonstration to explore these file formats, you'll learn how you can easily override the default Big SQL file format to use another format, in this case, Parquet. Parquet is a columnar storage format for Hadoop that's popular because of its support for efficient compression and encoding schemes. For more information on Parquet, visit http://parquet.io/.

The 7_Parquet statements.sql file is located in the */home/biadmin/labfiles/bigsql/BIG_SQL_Data_Analysis* folder.

1. Create a table named big_sales_parquet.

```
CREATE HADOOP TABLE IF NOT EXISTS big_sales_parquet
(
product_key      INT NOT NULL,
product_name VARCHAR(150),
quantity     INT,
order_method_en   VARCHAR(90)
)
STORED AS parquetfile;
```

   With the exception of the final line (which specifies the PARQUETFILE format), all aspects of this statement should be familiar to you by now.

2. Populate this table with data based on the results of a query.

   Note that this query joins data from 4 tables you previously defined in Big SQL using a TEXTFILE format. Big SQL will automatically reformat the result set of this query into a Parquet format for storage.

```
insert into big_sales_parquet
SELECT sales.product_key, pnumb.product_name,
sales.quantity,
meth.order_method_en
FROM
sls_sales_fact sales,
sls_product_dim prod,
sls_product_lookup pnumb,
sls_order_method_dim meth
WHERE
pnumb.product_language='EN'
AND sales.product_key=prod.product_key
AND prod.product_number=pnumb.product_number
AND meth.order_method_key=sales.order_method_key
and sales.quantity > 5500;
```

3. Query the table.

   Note that your SELECT statement does not need to be modified in any way because of the underlying file format.

```
select * from big_sales_parquet;
```

4. Inspect the results.

   A portion of the results appear as follows:

PRODUCT_KEY	PRODUCT_NAME	QUANTITY	ORDER_METHOD_EN
30107	BugShield Extreme	5937	Sales visit
30107	BugShield Extreme	6282	E-mail
30107	BugShield Extreme	6121	Mail
30107	BugShield Extreme	7300	Sales visit
30107	BugShield Extreme	8772	Web
30090	Single Edge	6619	Special
30107	BugShield Extreme	5855	Sales visit
30107	BugShield Extreme	5523	Web
30107	BugShield Extreme	5658	Web
30107	BugShield Extreme	6948	Sales visit

   Remember that parquet files are not human-readable.

   Navigate to /apps/hive/warehouse/bigsql.db/big_sales_parquet directory in the hdfs if you want to see the table's actual content.

Task 6. Working with external tables.

The previous tasks in this demonstration caused Big SQL to store tables in a default location (in the Hive warehouse). Big SQL also supports the concept of an externally managed table – i.e., a table created over a user directory that resides outside of the Hive warehouse. This user directory contains all the table's data in files.
As part of this demonstration, you will create a DFS directory, upload data into it, and then create a Big SQL table that over this directory. To satisfy queries, Big SQL will look in the user directory specified when you created the table and consider all files in that directory to be the table's contents. Once the table is created, you'll query that table.

1. If necessary, open a terminal window.

2. Check the directory permissions for your DFS.

```
hdfs dfs -ls /
```

```
Found 8 items
drwxrwxrwx   - yarn   hadoop          0 2015-04-07 11:33 /app-logs
drwxr-xr-x   - hdfs   hdfs            0 2015-03-31 05:53 /apps
drwxr-xr-x   - hdfs   hdfs            0 2015-03-31 07:28 /biginsights
drwxr-xr-x   - hdfs   hdfs            0 2015-03-31 05:51 /iop
drwxr-xr-x   - mapred hdfs            0 2015-03-31 05:51 /mapred
drwxr-xr-x   - hdfs   hdfs            0 2015-03-31 05:51 /mr-history
drwxrwxrwx   - hdfs   hdfs            0 2015-03-31 05:53 /tmp
drwxr-xr-x   - hdfs   hdfs            0 2015-03-31 07:28 /user
```

   If the /user directory cannot be written by the public (as shown in the example above), you will need to change these permissions so that you can create the necessary subdirectories for this task using your standard lab user account.

3. From the command line, issue this command to switch to the root user ID temporarily (if you are not already using the root user):

```
su root
```

4. If prompted, enter the password for this account. The password for root is **dalvm3**.

5. Then switch to the **hdfs** ID.

```
su hdfs
```

6. While logged in as user hdfs, issue this command:

```
hdfs dfs -chmod 777 /user
```

7. Confirm the effect of your change:

```
hdfs dfs -ls /
```

```
Found 8 items
drwxrwxrwx   - yarn   hadoop          0 2015-04-07 11:33 /app-logs
drwxr-xr-x   - hdfs   hdfs            0 2015-03-31 05:53 /apps
drwxr-xr-x   - hdfs   hdfs            0 2015-03-31 07:28 /biginsights
drwxr-xr-x   - hdfs   hdfs            0 2015-03-31 05:51 /iop
drwxr-xr-x   - mapred hdfs            0 2015-03-31 05:51 /mapred
drwxr-xr-x   - hdfs   hdfs            0 2015-03-31 05:51 /mr-history
drwxrwxrwx   - hdfs   hdfs            0 2015-03-31 05:53 /tmp
drwxrwxrwx   - hdfs   hdfs            0 2015-03-31 07:28 /user
```

8. Exit the hdfs user account:

```
exit
```

9. Exit the root user account and return to your standard user account.

```
exit
```

10. Create directories in your distributed file system for the source data files and ensure public read/write access to these directories. (If desired, alter the DFS information as appropriate for your environment.)

```
hdfs dfs -mkdir /user/bigsql_lab

hdfs dfs -mkdir /user/bigsql_lab/sls_product_dim

hdfs dfs -chmod -R 777 /user/bigsql_lab
```

11. Switch to the **bigsql** user with the password **ibm2blue**.

```
su bigsql
```

12. Upload the source data files into their respective DFS directories and change the local and DFS directories information below to match your environment.

```
hdfs dfs -copyFromLocal
/usr/ibmpacks/bigsql/4.0/bigsql/samples/data/GOSALESDW.SLS
_PRODUCT_DIM.txt
/user/bigsql_lab/sls_product_dim/SLS_PRODUCT_DIM.txt
```

13. List the contents of the DFS directories into which you copied the files to validate your work.

```
hdfs dfs -ls /user/bigsql_lab/sls_product_dim
```

```
Found 1 items
-rw-r--r--   3 bigsql hdfs      24089 2015-09-03 15:25 /user/bigsql_lab/sls_prod
uct_dim/SLS_PRODUCT_DIM.txt
```

14. Exit the **bigsql** user account and return to your standard user account:

```
exit
```

15. From your query execution environment (such as JSqsh or the SQL Editor), create an external Big SQL table for the sales product dimension (sls_product_dim_external).

    Note that the LOCATION clause in each statement references the DFS directory into which you copied the sample data.

    The 8_External tables.sql file is located in the /home/biadmin/labfiles/bigsql/BIG_SQL_Data_Analysis folder

```
-- product dimension table stored in a DFS directory
external to Hive
CREATE EXTERNAL HADOOP TABLE IF NOT EXISTS
sls_product_dim_external
 ( product_key INT NOT NULL
 , product_line_code INT NOT NULL
 , product_type_key INT NOT NULL
 , product_type_code INT NOT NULL
 , product_number INT NOT NULL
 , base_product_key INT NOT NULL
 , base_product_number INT NOT NULL
 , product_color_code INT
 , product_size_code INT
 , product_brand_key INT NOT NULL
 , product_brand_code INT NOT NULL
 , product_image VARCHAR(60)
 , introduction_date TIMESTAMP
 , discontinued_date TIMESTAMP
 )
 ROW FORMAT DELIMITED
 FIELDS TERMINATED BY '\t'
 LINES TERMINATED BY '\n'
 location '/user/bigsql_lab/sls_product_dim';
```

16. Query the table.

```
select product_key, introduction_date from
sls_product_dim_external
where discontinued_date is not null
fetch first 20 rows only;
```

17. Inspect the results.

PRODUCT_KEY	INTRODUCTION_DATE
30134	2003-01-01 00:00:00.000
30139	2003-01-01 00:00:00.000
30143	2003-01-01 00:00:00.000
30146	2003-01-01 00:00:00.000
30147	2003-01-01 00:00:00.000
30154	2003-01-01 00:00:00.000
30156	2003-01-01 00:00:00.000
30159	2003-01-01 00:00:00.000
30160	2003-01-01 00:00:00.000
30162	2003-01-01 00:00:00.000
30169	2003-01-01 00:00:00.000
30174	2003-01-01 00:00:00.000
30178	2003-01-01 00:00:00.000
30186	2003-01-01 00:00:00.000
30199	2003-01-01 00:00:00.000
30202	2003-01-01 00:00:00.000
30204	2003-01-01 00:00:00.000
30205	2003-01-01 00:00:00.000
30213	2003-01-01 00:00:00.000
30217	2003-01-01 00:00:00.000

20 rows in results(first row: 0.48s; total: 0.51s)

Results:
In this demonstration, you created and ran Big SQL queries that join data from multiple tables as well as perform aggregations and other SQL operations.

IBM Training

Unit summary

- Loading data into Big SQL tables
- List and understand the Big SQL data types
- Understand and use Big SQL supported file formats
- Querying Big SQL tables

Unit summary

IBM Training

Administering and managing Big SQL tables

IBM BigInsights v4.0

IBM Training

Unit objectives

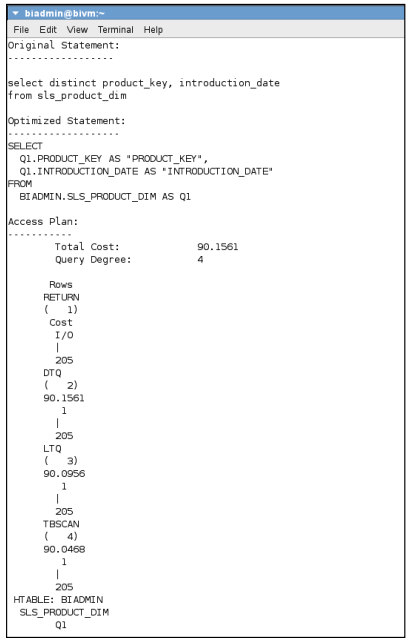
- Understanding Big SQL data access plans
- Using Big SQL statistics to improve query performance
- Controlling data access using column masking and row-based access control

Unit objectives

IBM Training

Big SQL data access plans

- Cost-based optimizer with query rewrite technology
  - Example: SELECT DISTINCT COL_PK, COL_X . . . FROM TABLE
  - Automatically rewrite query to avoid sort – primary key constraint indicates no nulls, no duplicates
  - Transparent to programmer
- ANALYZE TABLE … to collect statistics
- EXPLAIN to report detailed access plan
  - Subset shown at right



Administering and managing Big SQL tables

Big SQL data access plans

On a related topic, let's talk a little bit about data access plans. If you have a relational DBMS background, you know the importance of cost-based query optimization. Big SQL is based on IBM's decades of investment in query optimization technology, and that's one of the features that we believe has helped us achieve such remarkable results on internal performance tests.

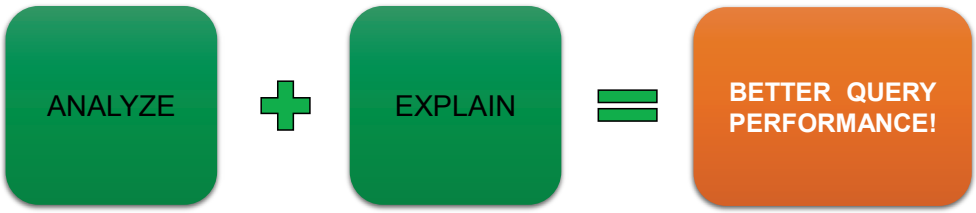
Let's take a look at a simple example. Here, we have a SELECT DISTINCT query issued against a table that includes a primary key. As shown at right, the optimizer automatically rewrites our submitted query into its logical equivalent by removing the DISTINCT clause and avoiding an unnecessary sort.

Big SQL gives programmers and administrators popular RDBMS mechanisms to influence and inspect query plans. These include an ANALYZE TABLE command to collect statistics about your data and an EXPLAIN facility to display detailed information about the data access plan for a given query.

IBM Training

Big SQL statistics

- Big SQL query optimizer uses certain statistics to determine an efficient data access strategy
- Use the ANALYZE command to update your statistics
  - Allows the optimizer to run with more accurate and up-to-date statistics
  - Use this information to better organize your data
- Tune your queries using the EXPLAIN statement
  - See the query that the optimizer have selected



Administering and managing Big SQL tables

Big SQL statistics

Big SQL query optimizer uses certain statistics to figure out an efficient data access strategy to run your query. However, in some cases, the optimizer might not have sufficient statistics to properly determine an efficient access strategy. This is where you will want to run the ANALYZE command to update the statistics. The ANALYZE command gathers information and statistics on the tables and columns. With this information, you can better organize your data to help the optimizer do its job. Once the statistics have been updated, you can tune your queries using the EXPLAIN command. This command allows you to see the query that the optimizer has selected to run. This will help you tune your queries for better performance.

The following slides will cover the ANALYZE and the EXPLAIN command to give you the complete picture.

IBM Training

ANALYZE - overview

- Gather statistics and provide them to the statistics table to optimize the current statement
- Big SQL analyze is based on Hive's analyze
  - Similar syntax
  - Uses Hive's analyze in the background for Hadoop objects
- Works on Big SQL database-only objects (non Hadoop) as well
  - Invokes runstats to do the statistics collection
  - Statistics are stored in the database catalog only
- Most statistic gathered are stored in both the Hive Metastore and the database catalog, except:
  - Histogram
  - Most Frequent Values (MFV)
  - Column Group Cardinality
  - Min and Max for string types
- Executing Hive ANALYZE will not update the database catalog
  - Can be used to debug issues

Administering and managing Big SQL tables

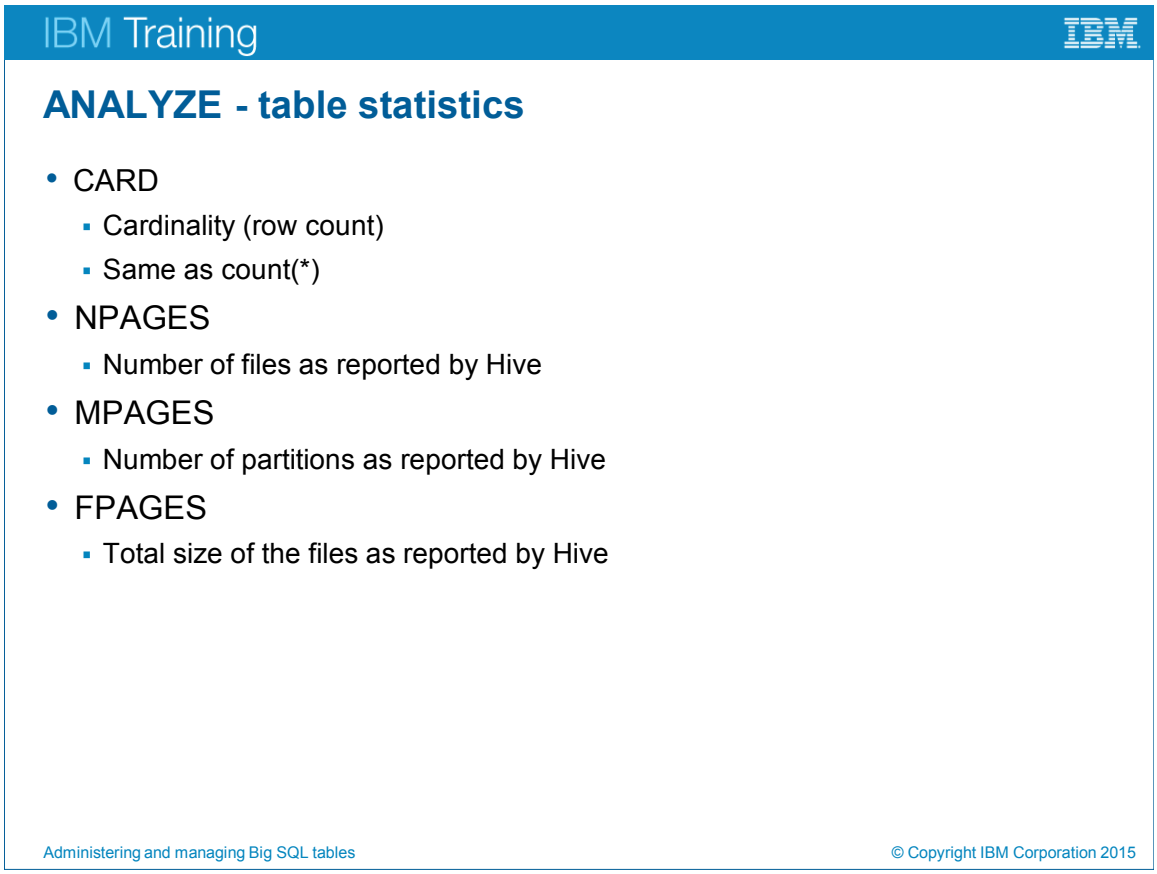
ANALYZE - overview

The ANALYZE command gathers and provides statistics to the statistics table to optimize the current SQL statement. The Big SQL analyze is based on Hive's analyze. In fact, it uses Hive's analyze in the background for Hadoop objects.

The ANALYZE command also works on database-only (or non Hadoop) objects. It invokes the runstats to do the statistics collection. The database-only statistics are stored separately in the database catalog only.

Most statistics gathered are stored in both catalogs, except for histogram, most frequent values (MFV), column group cardinality and min and max for string types. Those values are stored only in the database catalog.

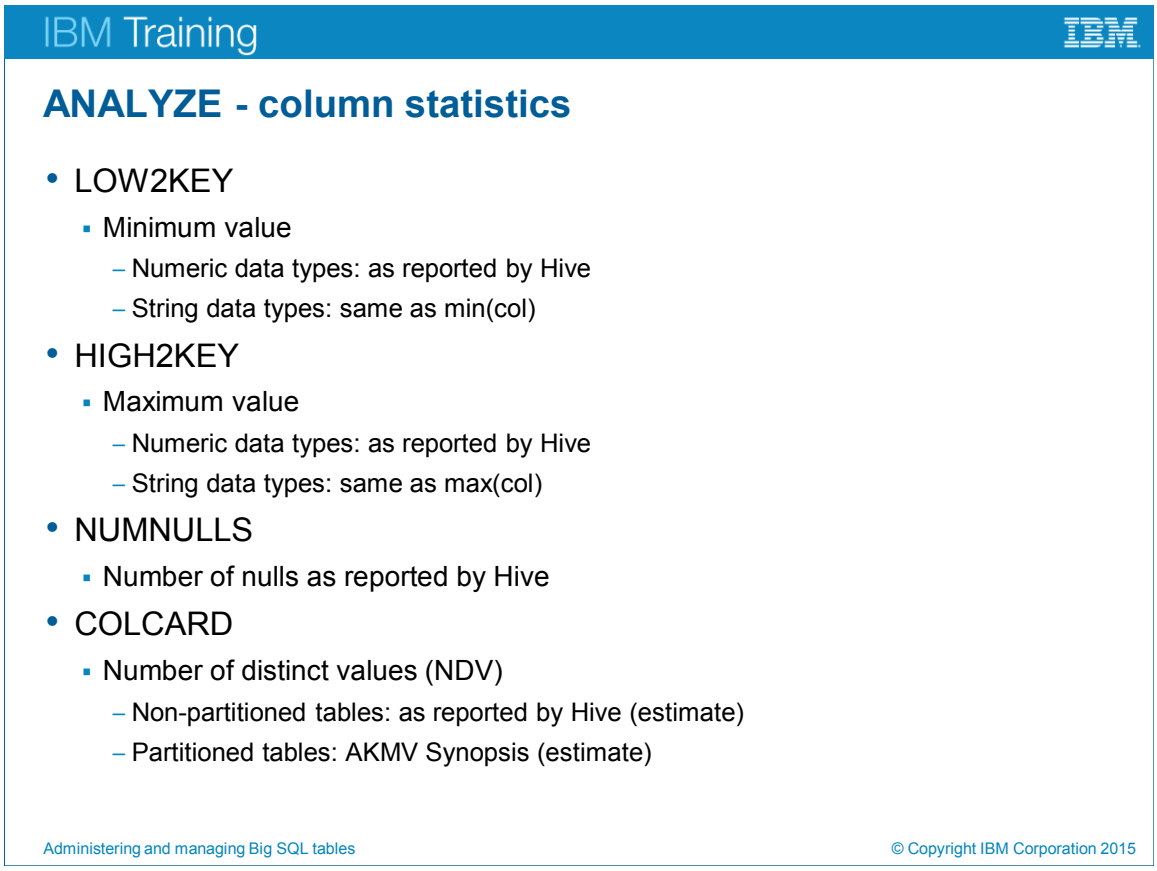
Executing the Hive ANALYZE command will only update the Hive MetaStore. It will not update the database catalog. You can use that to debug issues.



ANALYZE - table statistics

Here we have the table statistics that are collected when you run the ANALYZE command.

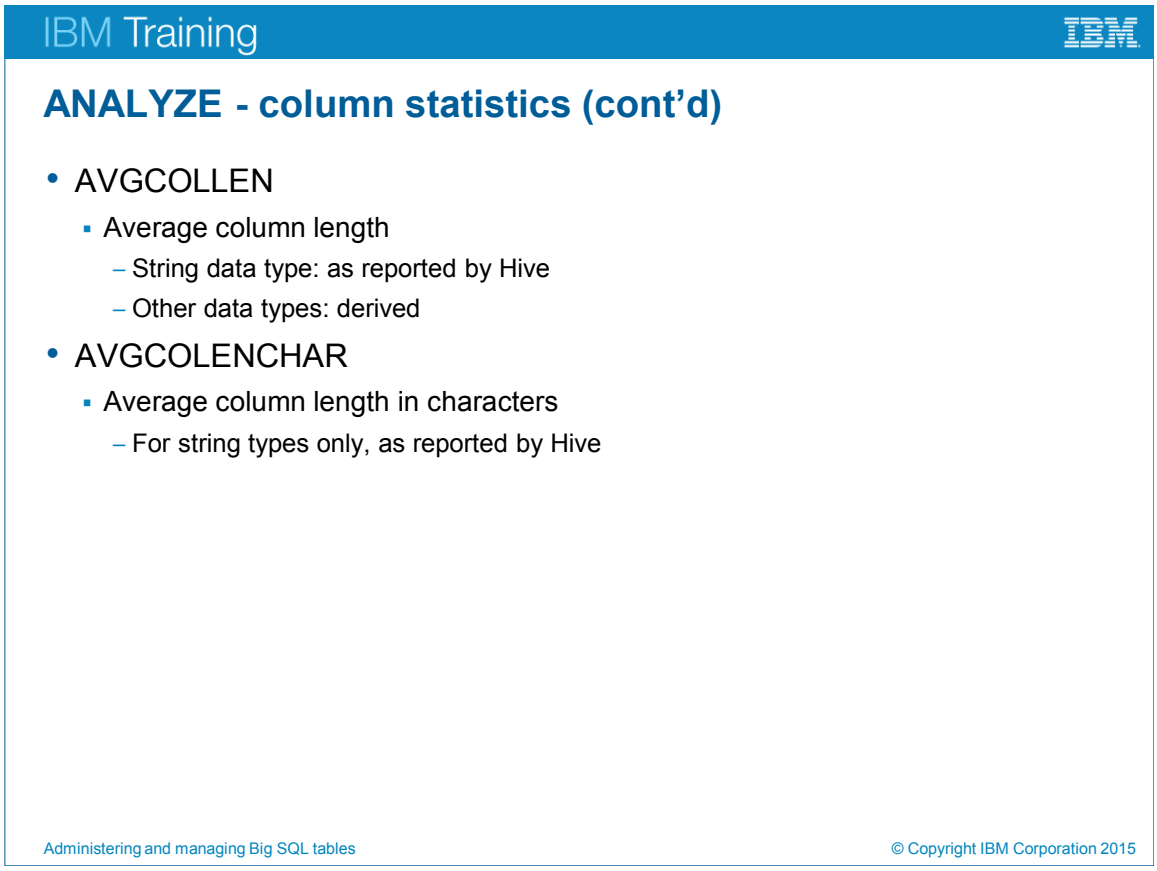
The first we have here is the CARD or cardinality which essentially is the number of rows. The next three are all statistics reported by Hive. NPAGE is the number of files. MPAGES is the number of partitions and FPAGES is the total size of the files



ANALYZE - column statistics

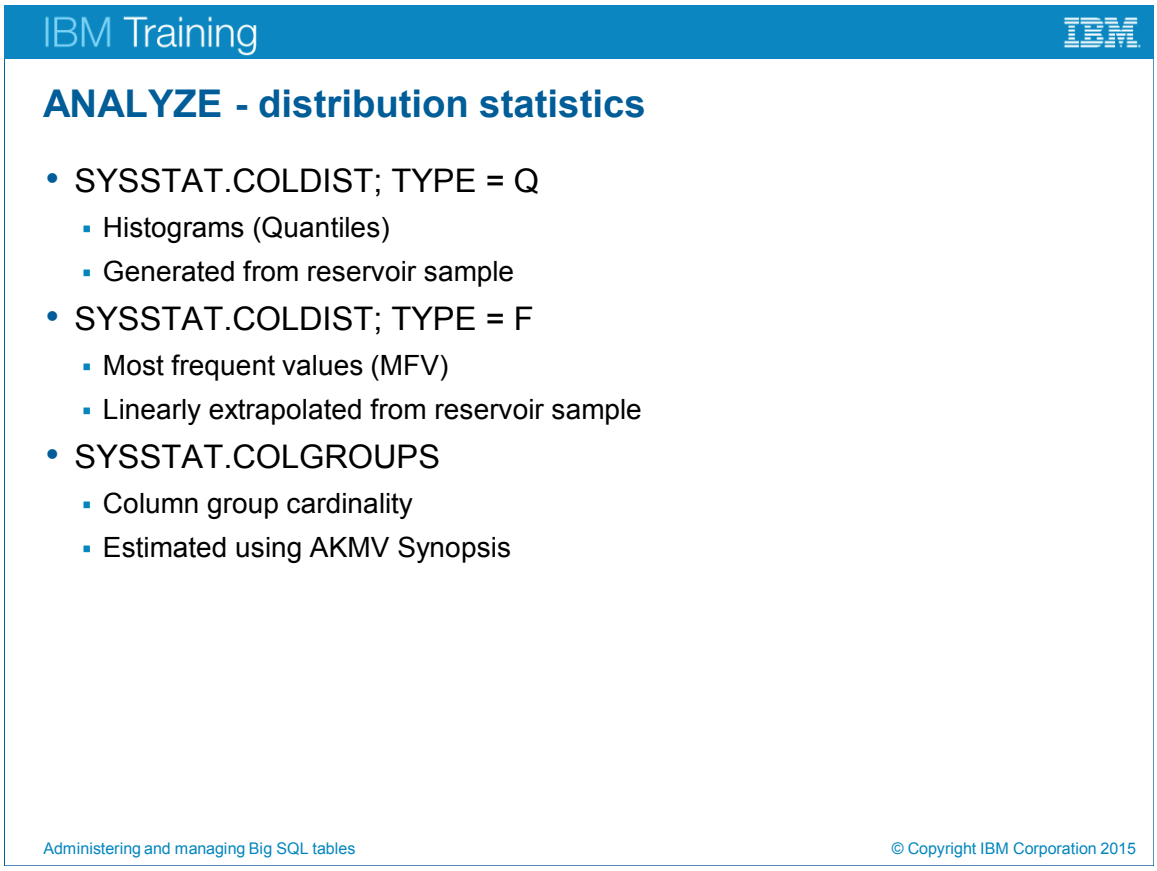
Here we have the column statistics that are collected when you run the ANALYZE command.

The LOW2KEY and the HIGH2KEY are essentially the min and max as reported by Hive for the numeric data types. Hive did not provide the statistics for string data type, so Big SQL has one which basically uses the min and max function. The NUMNULLS is just the number of NULLS as reported by Hive. COLCARD, or the number of distinct values, Hive provides statistics for non-partitioned table. For partitioned tables, we used the AKMV synopsis to get an estimate of the number of distinct values across the partitions.



ANALYZE - column statistics (cont'd)

These are two more column statistics collected by the ANALYZE command. These two are the average column length and column length in characters. The string data type is reported by Hive. The other data types are derived.



ANALYZE - distribution statistics

Here are the distribution statistics that are collected by the ANALYZE command.

The first set of statistics comes from the SYSTAT.COLDIST table. With the type = Q, you get the histograms or quantiles. This data is generated from the reservoir sample. With the type = F, you get the most frequent values which is linearly extrapolated from the reservoir sample that is taken from the data. The column group cardinality is collected in the SYSSTAT.COLGROUPS table and is estimated using the same AKMV Synopsis that is used on the partition tables.

IBM Training

ANALYZE - syntax

```
>>-ANALYZE TABLE--table-name------------------>

>--COMPUTE STATISTICS--+-| analyze-col |-+---><
                       +-NOSCAN----------+
                       +-PARTIALSCAN-----+
                       '-COPYHIVE--------'
analyze-col
   .-FULL--------.
   +-INCREMENTAL-+
|--+-------------+--FOR COLUMNS--| colgroup |--|


colgroup
     .-,---------------------------.
     V                             |
|------+-coln--------------------+-+----------|
       '-(--cola--,--colb--...--)-'
```

Administering and managing Big SQL tables

ANALYZE - syntax

Now we take a look at the syntax. The following pages will cover these options.

>>-ANALYZE TABLE--table-name------------------>

>--COMPUTE STATISTICS--+-| analyze-col |-+---><

+-NOSCAN----------+

+-PARTIALSCAN-----+

'-COPYHIVE--------'

analyze-col

.-FULL--------.

+-INCREMENTAL-+

|--+-------------+--FOR COLUMNS--| colgroup |--|

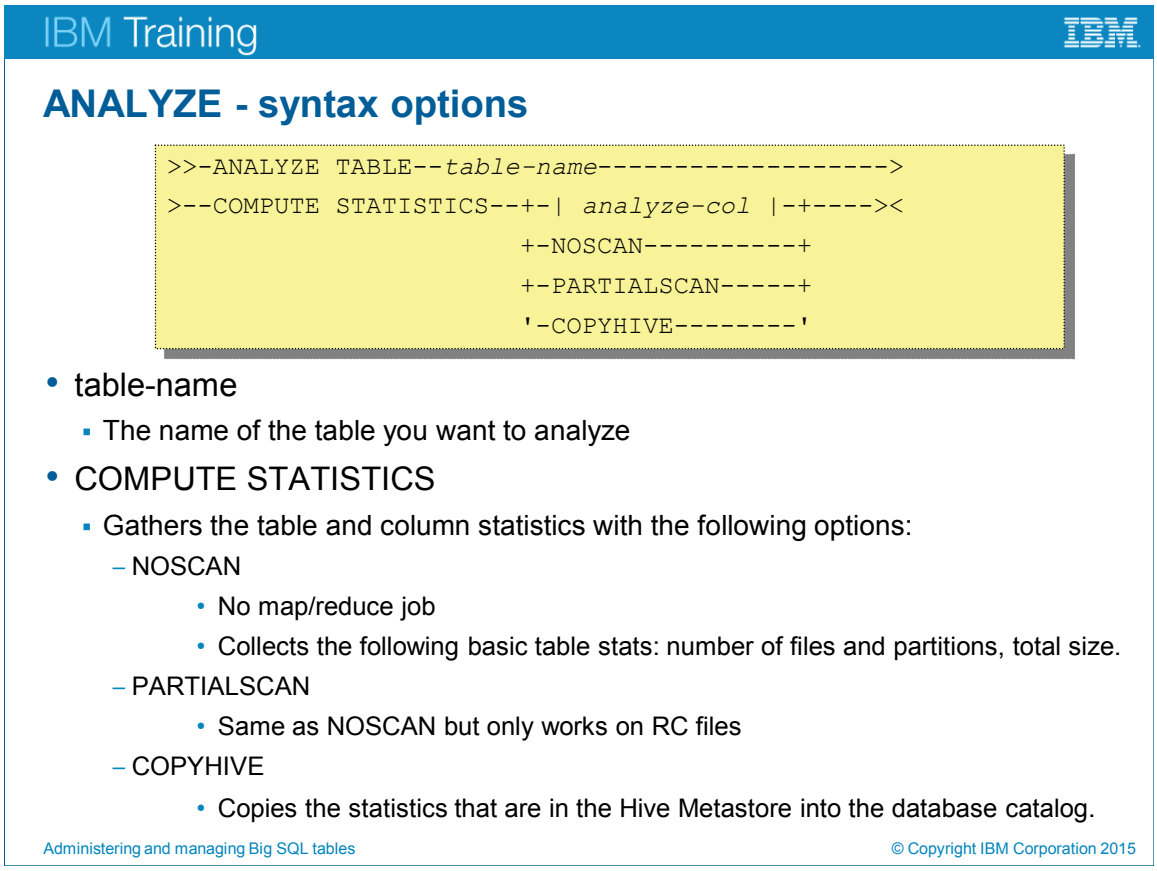
colgroup

.-,---------------------------.

V |

|------+-coln--------------------+-+----------|

'-(--cola--,--colb--...--)-'



ANALYZE - syntax options

These are the syntax options for the ANALYZE command.

The obvious one is the table-name, where you specify the name of the table you wish to analyze.

Then you can specify some options for COMPUTE STATISTICS that will gather the table and column statistics.

NOSCAN tells it to not run any map/reduce jobs and just collect the following basic table stats:

- number of files and partitions
- total size

PARTIALSCAN is the same as NOSCAN, but it only works on RC files.

COPYHIVE tells it to copy the statistics from the Hive MetaStore into the database catalog.

IBM Training

ANALYZE - syntax options (cont'd)

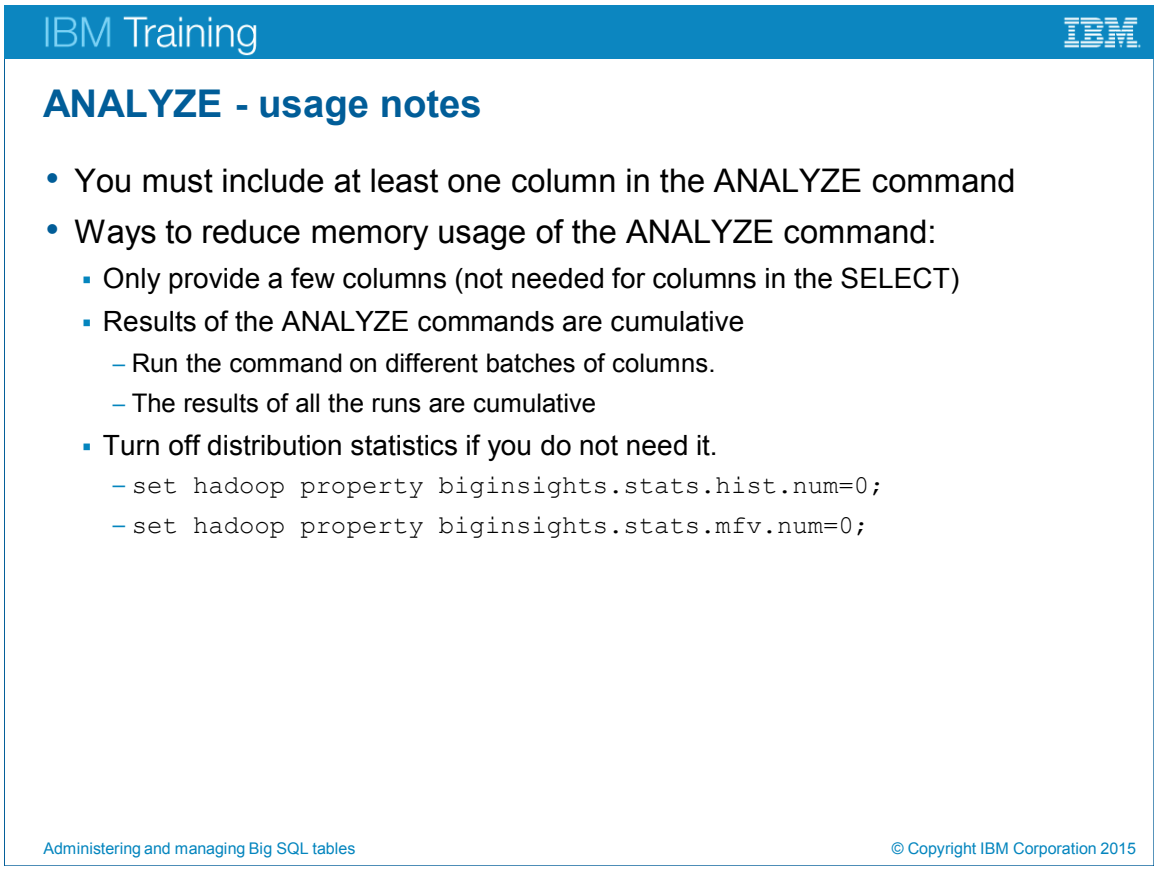
```
analyze-col
   .-FULL--------.
   +-INCREMENTAL-+
|--+-------------+--FOR COLUMNS--| colgroup |--|

colgroup
     .-,----------------------------.
     V                              |
|------+-coln----------------------+-+----------|
       '-(--cola--,--colb--...--)-'
```

- FULL
  - Collect statistics on all partitions for a partitioned table
  - On non-partition tables, it is always a FULL scan
- INCREMENTAL
  - Collect statistics only on partitions that needs to be updated
  - Ignored for a non-partitioned table
- FOR COLUMNS
  - Table statistics and column-level statistics are gathered for the columns that you specify
    - colgroup is separated by commas if there are more than one columns

ANALYZE - syntax options (cont'd)

You can also do a FULL or an INCREMENTAL scan. A FULL scan collect the statistics on all partitions for a partitioned table. An INCREMENTAL scan only works on the partitions that needs to be updated. These options are ignored for a non-partitioned table because it will always run a FULL scan.

Use the FOR COLUMNS option to give valuable information to the optimizer. You specify the columns in your tables for which you want the statistics to be collected. You separate the columns by commas if there are more than one column.



ANALYZE - usage notes

Here are some usage notes. In the ANALYZE command, you must include at least one column. However, to conserve memory, you should only include columns that matter, particularly the ones that are referenced in the predicates. This includes the join predicates. You do not need to include any columns in the SELECT clause.

The ANALYZE command itself can be quite memory intensive, so here are a few things you can do to help alleviate memory usage. The results of the ANALYZE command are cumulative, meaning that you can run them in batches. For example, you can run the command with one set of columns, and then run the command again with a different set of columns, until all the columns have been analyzed. You should also turn off the distribution statistics if you do not need it. This saves a substantial amount of memory.

There are a few more properties that you can set. Refer to the IBM Knowledge Center for more information.

IBM Training

ANALYZE - examples

- Analyzing a non-partitioned table

```
ANALYZE TABLE myschema.Table2 COMPUTE STATISTICS FOR
COLUMNS (c1,c2),c3,c4;
```

- Analyze a table and specific columns and then use SYSSTAT.COLUMNS to view the statistics.

```
ANALYZE TABLE gosalesdw.MRK_PROD_SURVEY_TARG_FACT COMPUTE
STATISTICS FOR COLUMNS
month_key,product_key,product_survey_key,
product_topic_target;
```

```
SELECT cast(COLNAME as varchar(20)) AS "COL_NAME", COLCARD,
cast(HIGH2KEY as varchar(100)) AS HIGH2KEY",
cast(LOW2KEY as varchar(100)) AS "LOW2KEY", NUMNULLS ,AVGCOLLEN
from SYSSTAT.COLUMNS WHERE TABNAME = MRK_PROD_SURVEY_TARG_FACT'
and (COLNAME='PRODUCT_KEY' or COLNAME='PRODUCT_TOPIC_TARGET')
order by COLNAME;
```

Administering and managing Big SQL tables

ANALYZE - examples

Let us look at some examples. The first one is to analyze a non-partitioned table.

Example

```
ANALYZE TABLE myschema.Table2 COMPUTE STATISTICS FOR COLUMNS
(c1,c2),c3,c4;
```

The next example shows how to view the statistics after you have run the ANALYZE command.

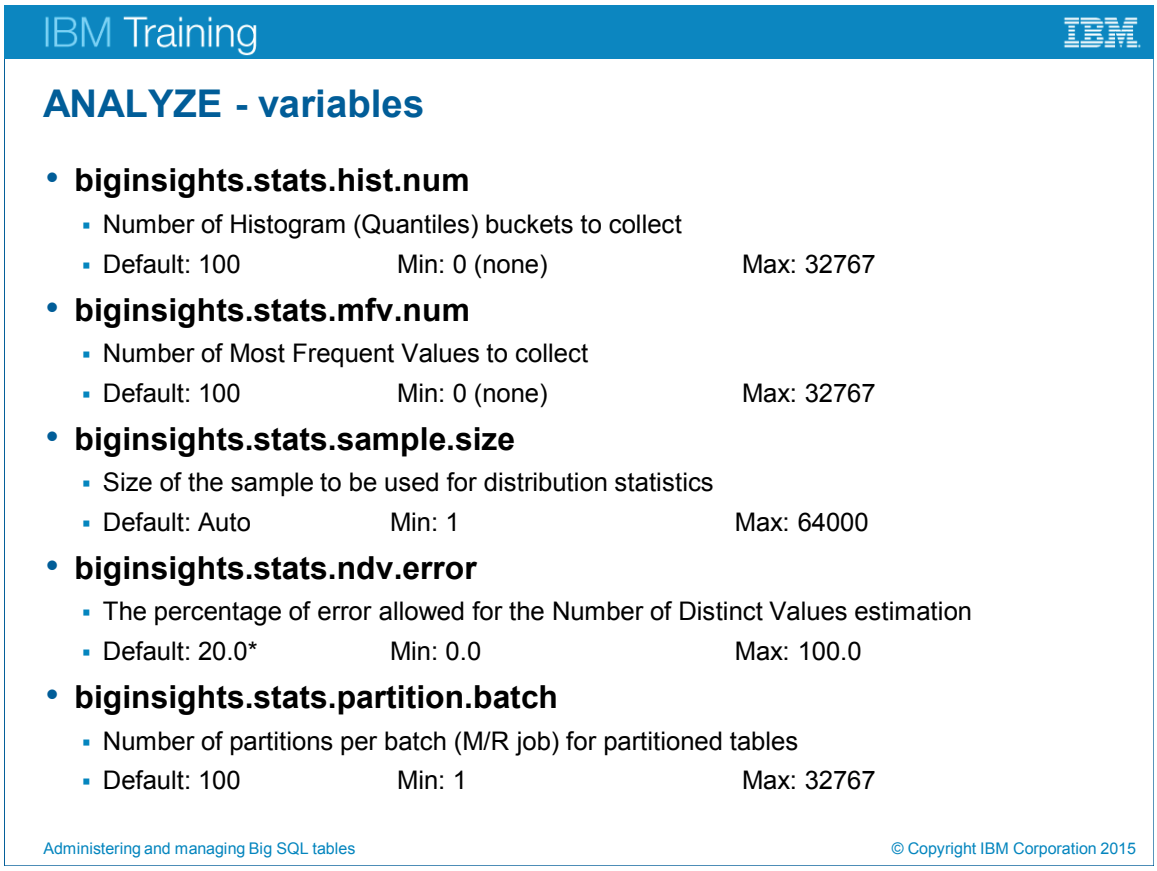
Example

```
ANALYZE TABLE gosalesdw.MRK_PROD_SURVEY_TARG_FACT COMPUTE
STATISTICS FOR COLUMNS
month_key,product_key,product_survey_key,
product_topic_target;
```

Execute the shown select statement to view the results of the ANALYZE command.

Example

```
SELECT cast(COLNAME as varchar(20)) AS "COL_NAME",
COLCARD,      cast(HIGH2KEY as varchar(100)) AS HIGH2KEY",
cast(LOW2KEY as varchar(100)) AS "LOW2KEY", NUMNULLS
,AVGCOLLEN
from SYSSTAT.COLUMNS WHERE TABNAME =
MRK_PROD_SURVEY_TARG_FACT'
and (COLNAME='PRODUCT_KEY' or
COLNAME='PRODUCT_TOPIC_TARGET') order by COLNAME;
```



ANALYZE - variables

Here are some variables that you set and their current defaults.

biginsights.stats.hist.num

- sets the number of histogram buckets to collect. Defaults to 100, Min at 0, Max at 32767.

biginsights.stats.mfv.num

- sets the number of most frequent values to collect. Default: 100, Min at 0, Max 32676.

biginsights.stats.sample.size

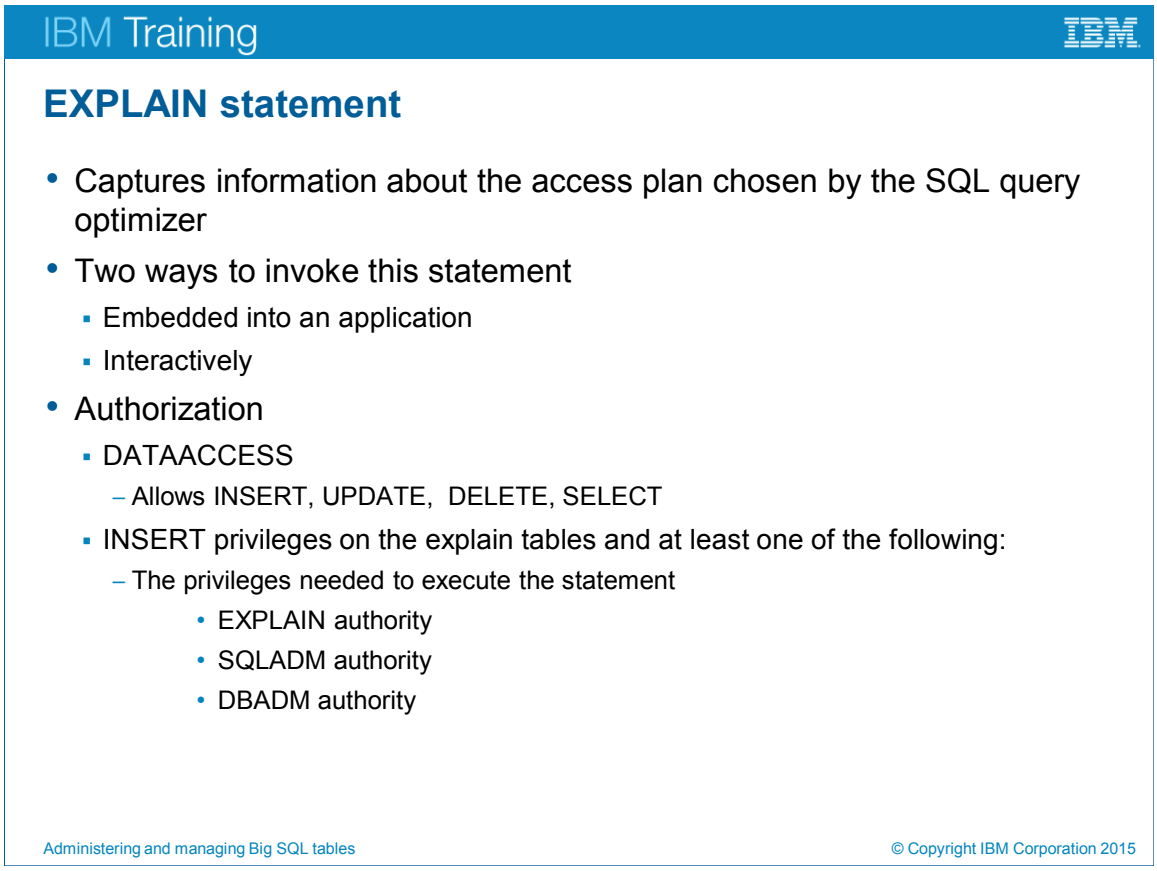
- sets the size of the sample to use for the distribution. Default: Auto, Min: 1, Max 64000

biginsights.stats.ndv.error

- sets the percentage of error allowed for the number of distinct values estimation. Default: 20*, Min 0, Max 100. The default of 20 comes from the Hive variable. If you do not set this property, it will use whatever the value of the Hive variable.

biginsights.stats.partition.batch

- sets the number of partitions per batch (M/R job) for partitioned tables. Hive only allows you to collect stats on one partition at a time. In Big SQL, this ability allows to analyze multiple partitions at once to use fewer map/reduce jobs. Default: 100, Min: 1, Max 32767.



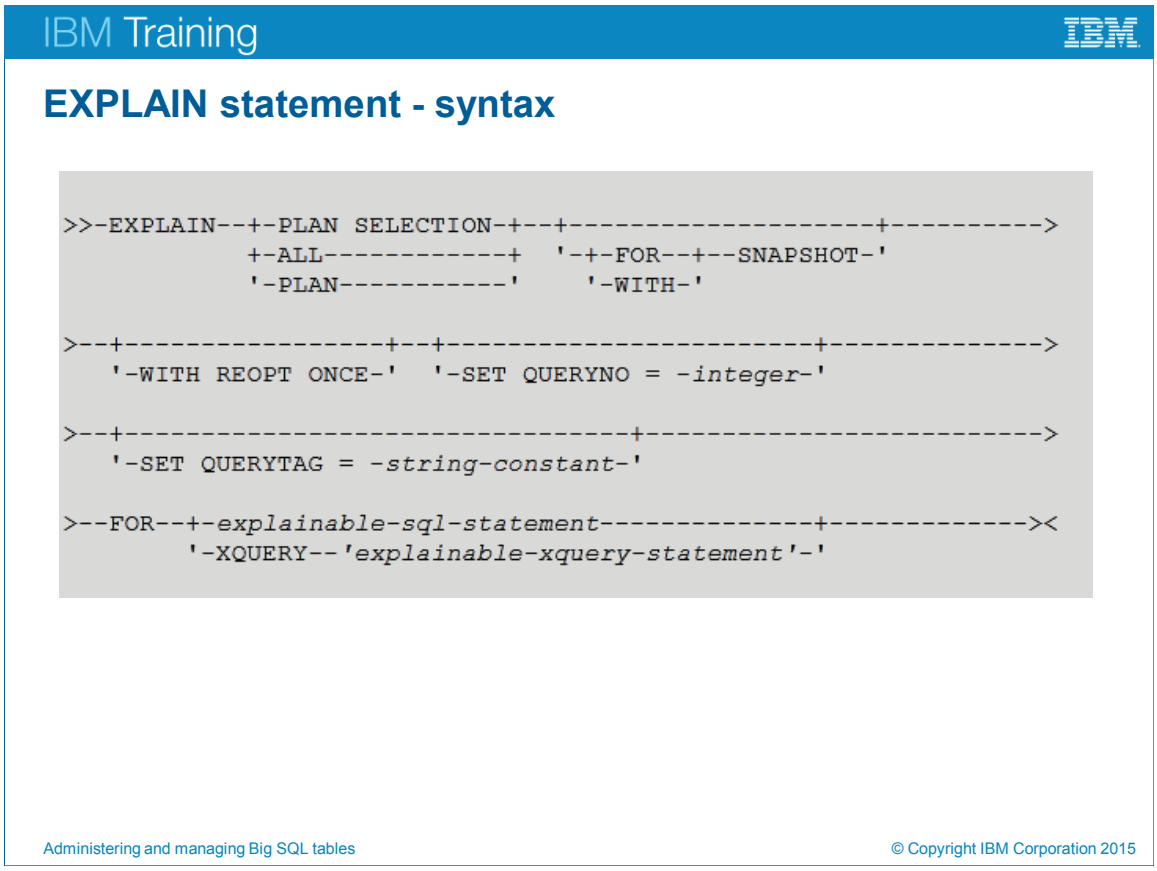
EXPLAIN statement

The EXPLAIN statement captures information about the access plan that is chosen by the SQL query optimizer. You can view this plan to see how it is executed to fine tune your queries. There are two ways to invoke this statement. You can either embed this statement into an application or run it interactively.

To run this statement, you must have the proper authorization. You must have at least one of the following authorizations:

- DATAACCESS which allows INSERT, UPDATE, DELETE, OR SELECT
- INSERT privileges with at least one of the following privileges needed to execute the statement itself:
  - EXPLAIN authority
  - SQLADM authority
  - DBADM authority

For example. If you are running the EXPLAIN on a DELETE statement, then you must have the authority on the DELETE statement.



EXPLAIN statement - syntax

Here is the EXPLAIN syntax. I'll go over some of the options here. You can find additional information in the IBM Knowledge Center.

PLAN SELECTION

- indicates the information for the plan selection.

ALL

- specifying ALL is the same as specifying PLAN SELECTION

PLAN

- this option provides syntax tolerance for existing database application from other systems.

FOR SNAPSHOT

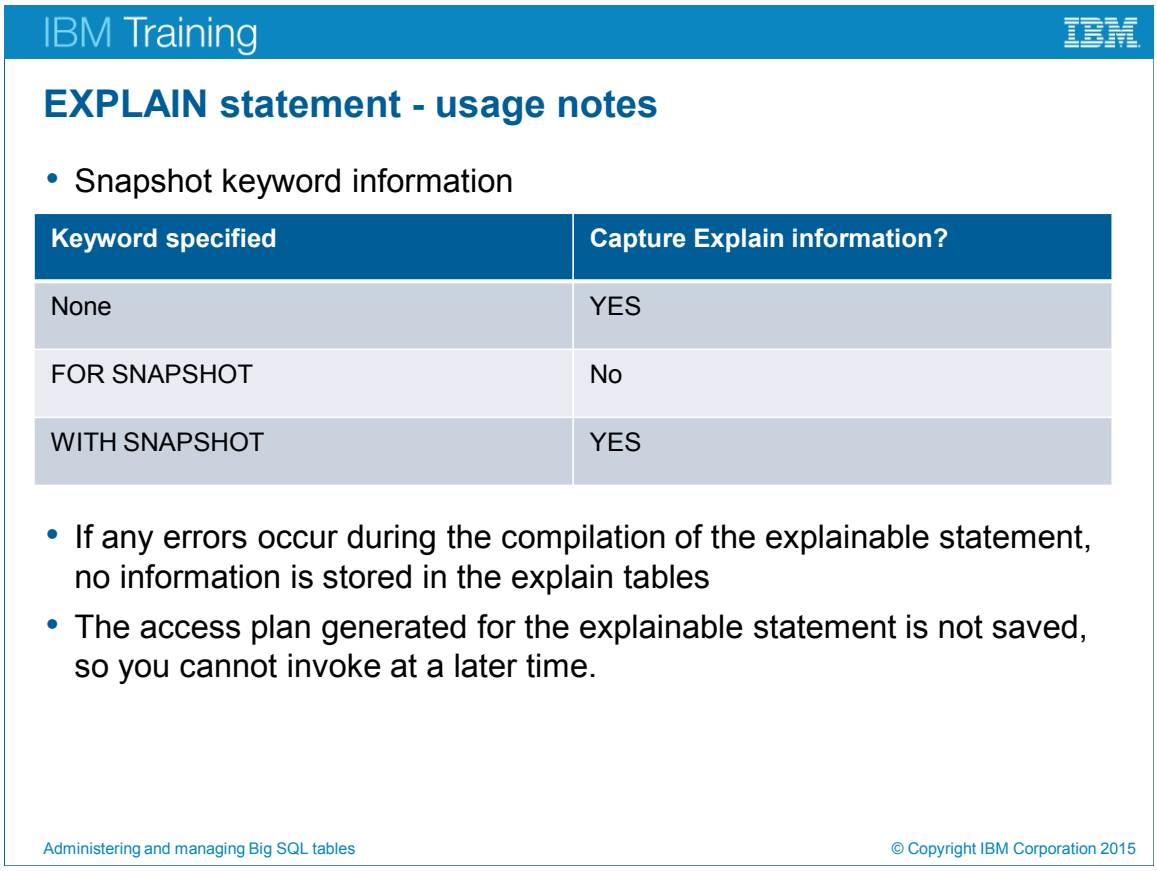
- This clause indicates that only an explain snapshot is to be taken and placed into the SNAPSHOT column of the EXPLAIN_STATEMENT table. No other information is taken.

WITH SNAPSHOT

- This clause indicates that in addition to the regular explain information, an explain snapshot is to be taken.

The FOR *explainable-sql-statement* clause specifies the SQL statement to be explained. The statement can be any valid CALL, DELETE, INSERT, MERGE, SELECT, ETC. If the EXPLAIN is embedded in a program, the *explainable-sql-statement* can contain references to host variables as long as they are defined in the program. It is important to note that the *explainable-sql-statement* must be a valid SQL statement that can run on its own.

You can also run EXPLAIN on a XQUERY explainable statement

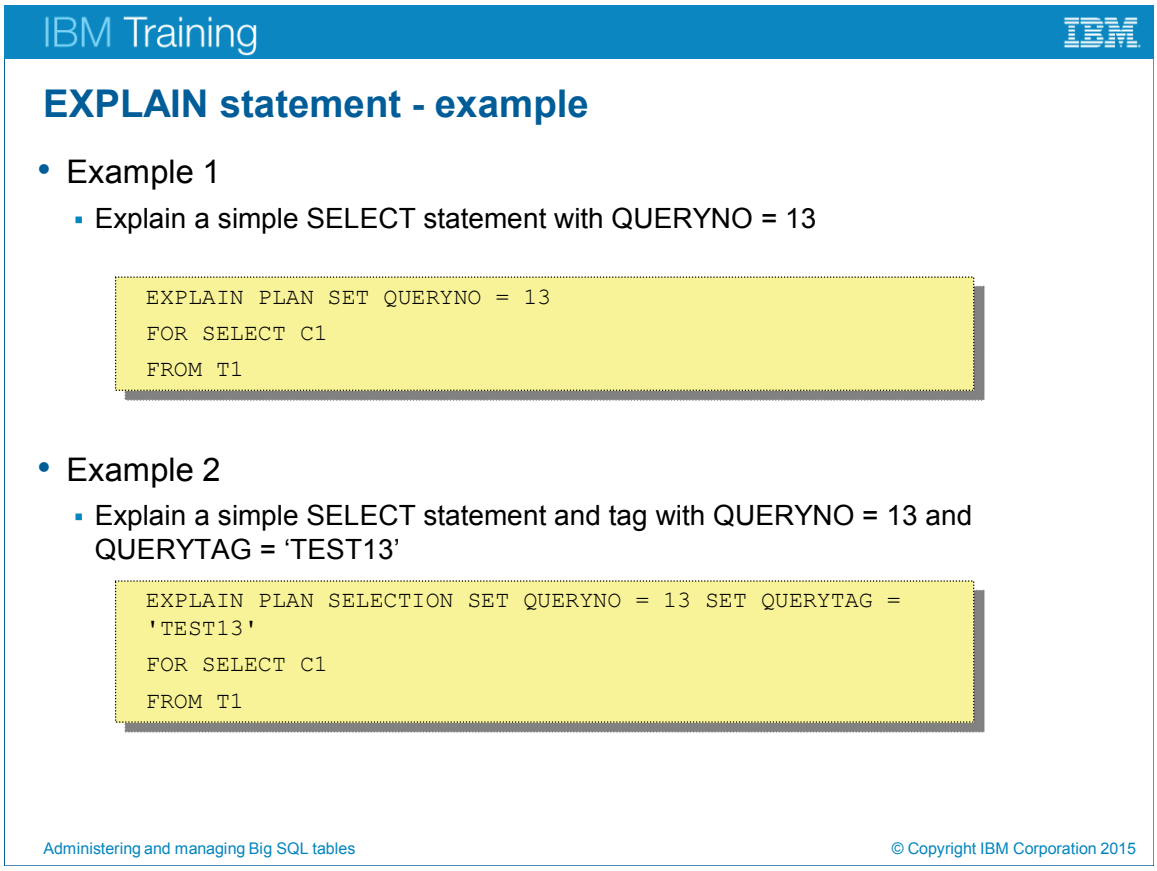


EXPLAIN statement - usage notes

Here are just some usage notes about the EXPLAIN statement. The snapshot keyword information shows when the explain information is captured. When you use the FOR SNAPSHOT clause, no information is collected. When you use the WITH SNAPSHOT, explain information is collected. When no keyword is specified, the default is also to collect the information

It is important to note that if any errors occur during the compilation of the explainable statement, nothing is stored in the tables.

Also, the access plan that is generated for the explainable statement is not saved, so you cannot invoke it at a later time.



EXPLAIN statement - examples

Here are two examples of using the EXPLAIN statement.

The first one shows selecting c1 from t1 with a predicate defined.

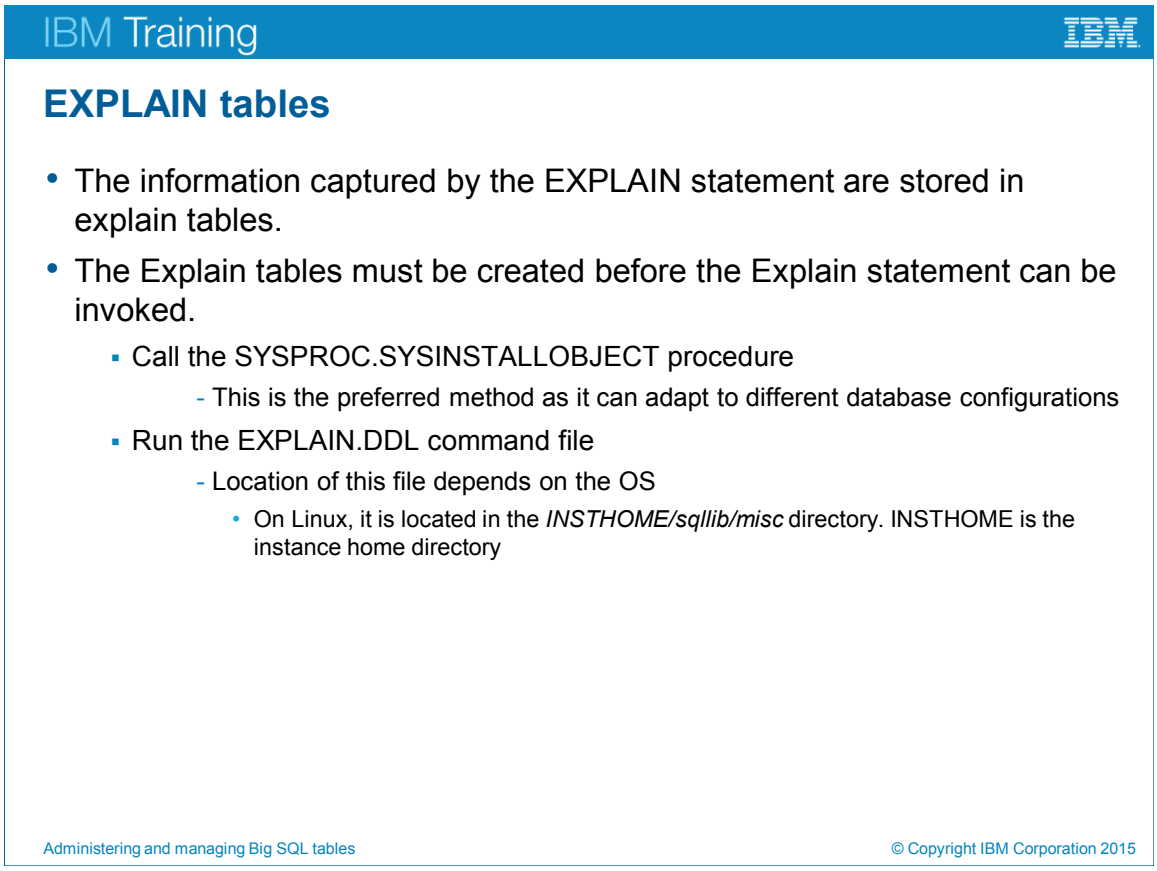
Example

```
EXPLAIN PLAN SET QUERYNO = 13
FOR SELECT C1
FROM T1
```

The second example shows it with two predicates defined.

Example

```
EXPLAIN PLAN SELECTION SET QUERYNO = 13 SET QUERYTAG =
'TEST13'
FOR SELECT C1
FROM T1
```



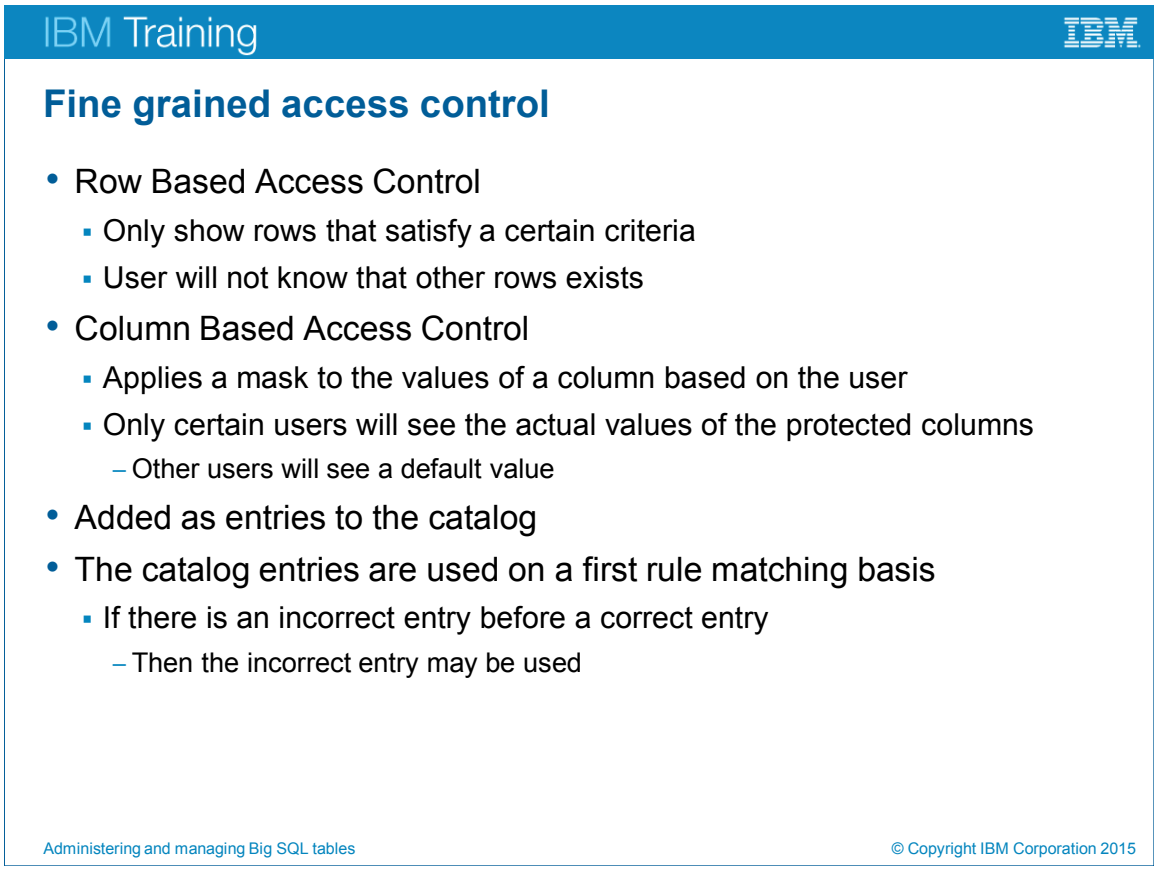
EXPLAIN tables

The EXPLAIN information is collected and stored in the Explain tables. These tables must exist before you can invoke the EXPLAIN command. There are two ways you can create these tables.

The first way is to call the SYSPROC.SYSINSTALLOBJECT procedure.

The second way you can create the Explain tables is by running the EXPLAIN.DDL file. The location of this file depends on the OS. On Linux, it is located under INSTHOME/sqllib/misc where INSTHOME is the instance home directory.

The preferred way is to call the procedure because it can adapt to different database configurations. For example, if BLOCKNONLOGGED parameter is set to *yes*, then some statements in EXPLAIN.DDL fail because NOT LOGGED clause is used for LOB columns. The procedure is smart enough to not use the NOT LOGGED clause if the BLOCKNONLOGGED is set to *yes*.



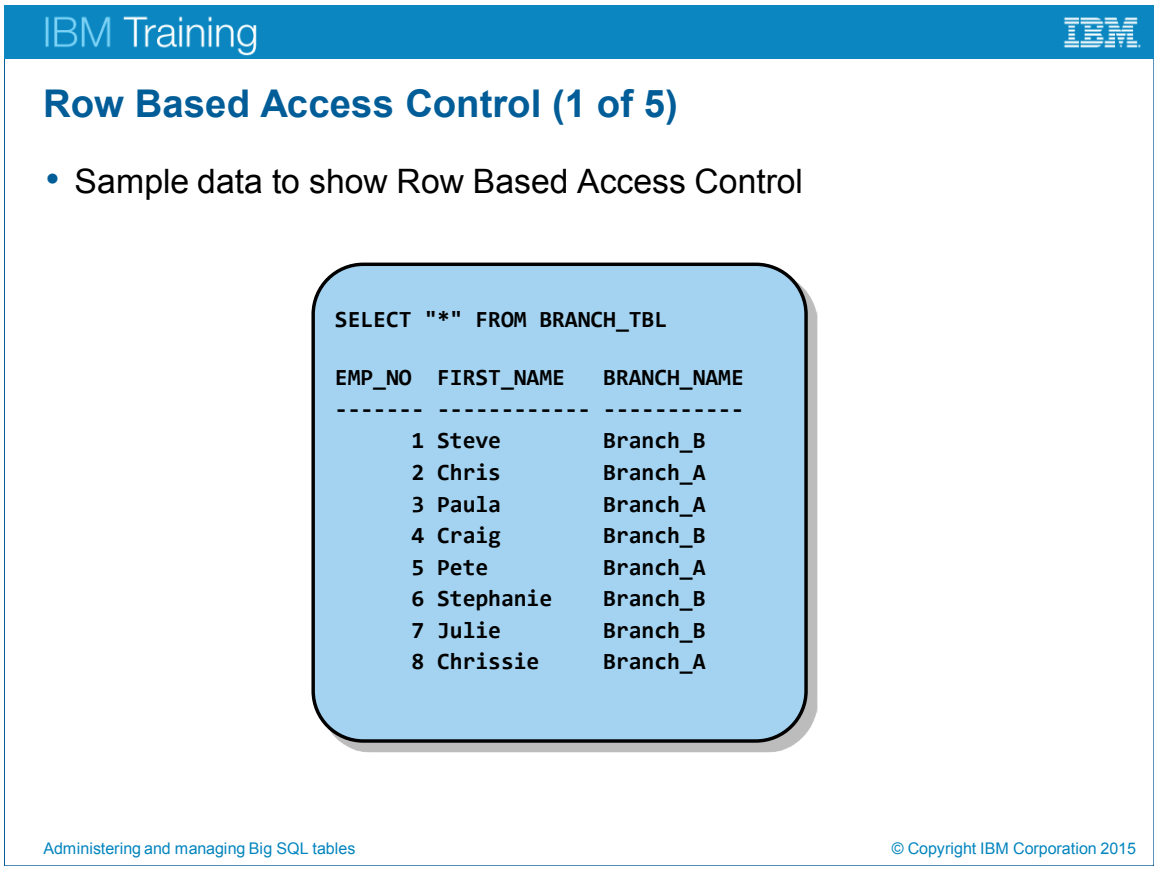
Fine grained access control

Fine grained access control consists of two parts: row based and column based access control.

Row based access control only shows rows that satisfy a certain criteria. The user will only see what they have been given permission to see – they will not even know about the other rows.

Column based access control applies a mask to the values of a column based on a user. If a user is not supposed to see a particular column, a default value will be returned instead.

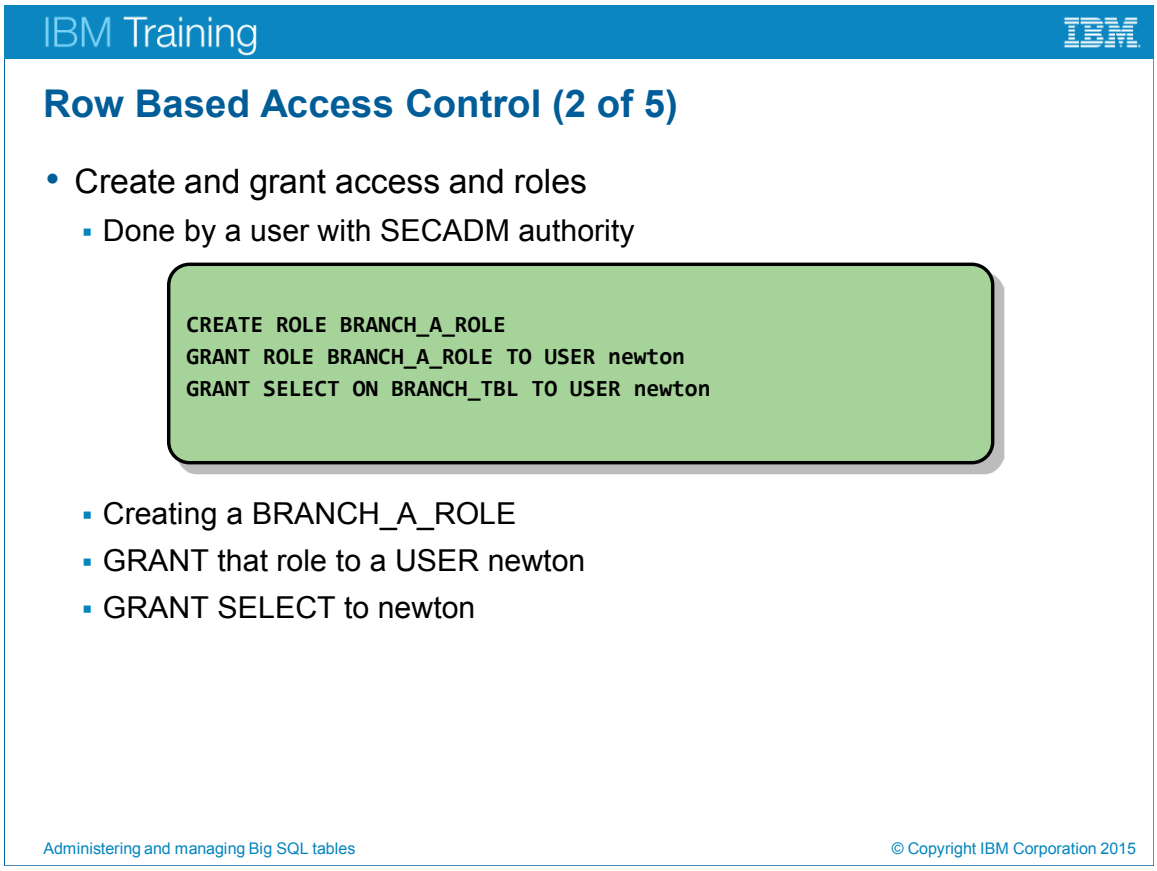
Each of the rules of the row based or column based access controls are added as entries to the catalog. These entries are used on a first rule matching basis. In other words, if you added an incorrect rule and then add a correct one, the incorrect rule may be used first. When that happens, take a look at the catalog and remove the incorrect rules.



Row Based Access Control (1 of 5)

The next five slides will show an example of row based access control.

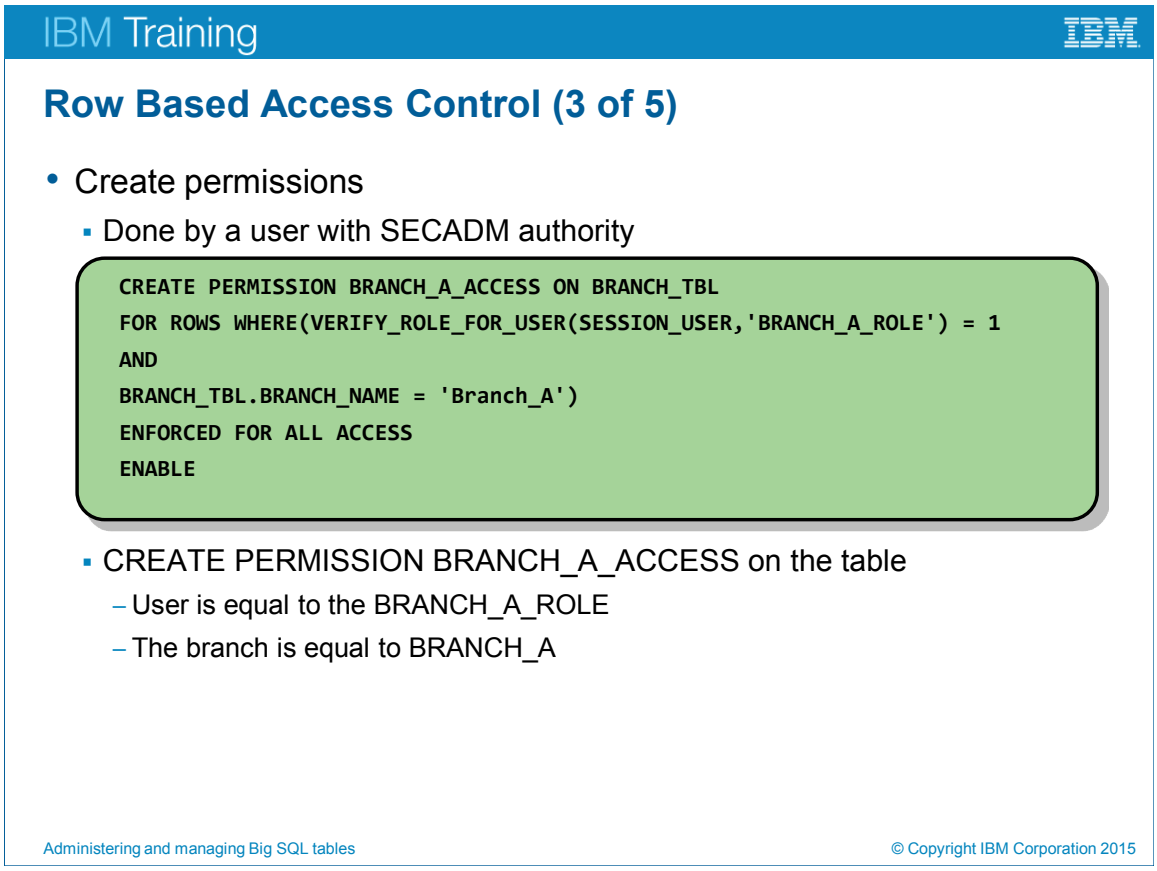
Start off with a sample dataset. Do a SELECT * FROM BRANCH_TBL, which represents the employees and the branch that they work at. There are three columns, EMP_NO, FIRST_NAME, and BRANCH_NAME with only eight rows of sample data. It just so happens that there are rows from Branch_A and four rows from Branch_B.



Row Based Access Control (2 of 5)

First thing to do for setting up the row based access control is to create roles and grant access to the role. This step has to be done by a user with the SECADM authority.

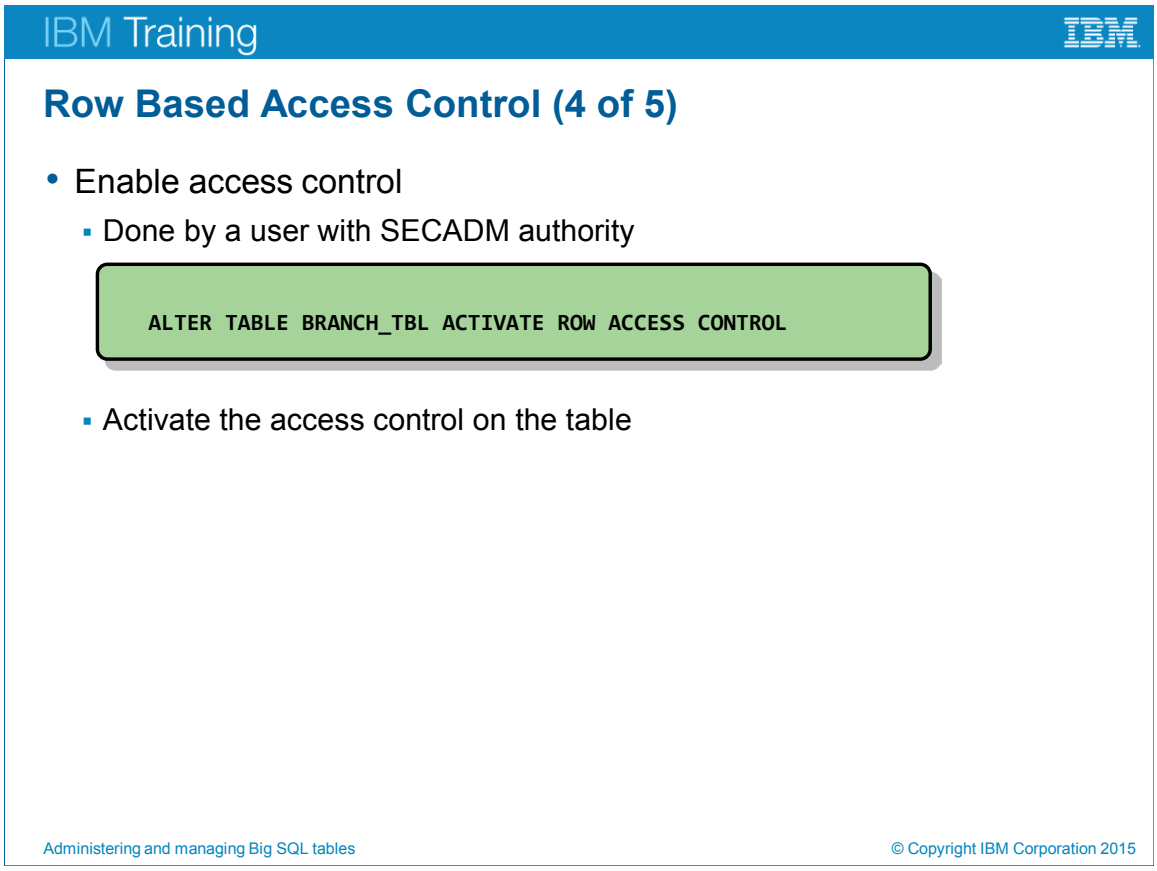
First, the user creates the role. Then the role is granted a user, *newton*, in this case. Then GRANT SELECT on BRANCH_TBL to newton as well.



Row Based Access Control (3 of 5)

Once the role has been created and granted, the permission needs to be created. This is also done by a user with the SECADM authority. This permission will basically tell Big SQL which rows it should return for a specified role.

In the CREATE PERMISSION statement, you specify that the user has to be part of the BRANCH_A_ROLE and also that the branch name is *Branch_A.*



Row Based Access Control (4 of 5)

When the permission has been created, the user with the SECADM authority must enable it on that table using the ALTER TABLE statement.

```
ALTER TABLE BRANCH_TBL ACTIVATE ROW ACCESS CONTROL
```

IBM Training

Row Based Access Control (5 of 5)

- Select as Branch_A user

```
CONNECT TO TESTDB USER newton

SELECT "*" FROM BRANCH_TBL

EMP_NO      FIRST_NAME   BRANCH_NAME
----------- ------------ -----------
          2 Chris        Branch_A
          3 Paula        Branch_A
          5 Pete         Branch_A
          8 Chrissie     Branch_A

  4 record(s) selected.
```

  - Only the 4 records from Branch_A are returned

Administering and managing Big SQL tables

Row Based Access Control (5 of 5)

Finally, connect as a user from the *Branch_A* role, which is newton in the example. Then execute a SELECT * FROM BRANCH_TBL statement. Four rows from Branch_A are returned. Note that in this case, the user in this role is not even aware of any other rows in that table. The permission prevents the user from knowing what else is in the table and only see what is accessible to that role.

IBM Training

Column Based Access Control (1 of 5)

- Data from the salary table
- Three columns
  - EMP_NO
  - FIRST_NAME
  - SALARY

```
SELECT "*" FROM SAL_TBL

EMP_NO  FIRST_NAME   SALARY
------- ------------ -----------
      1 Steve            250000
      2 Chris            200000
      3 Paula           1000000
```


Column Based Access Control (1 of 5)

Here is a column based access control example. Similar with row based access control, we have a sample set of data. This time it is table of salary with three columns: EMP_NO, FIRST_NAME, SALARY.

IBM Training

Column Based Access Control (2 of 5)

- Create and grant access and roles
  - Created by a user with SECADM authority
- Create a manager and an employee role and grant it to two different users.

```
CREATE ROLE MANAGER
CREATE ROLE EMPLOYEE

GRANT SELECT ON HADOOP.SAL_TBL TO USER socrates
GRANT SELECT ON HADOOP.SAL_TBL TO USER newton

GRANT ROLE MANAGER TO USER socrates
GRANT ROLE EMPLOYEE TO USER newton
```


Column Based Access Control (2 of 5)

As with before, you need to create roles. This time, create two roles, one manager and one employee. The intent here is to hide the salary column if it is an employee searching. Grant the respective roles to the two users socrates and newton. Also grant the SELECT privilege to those two users.

IBM Training

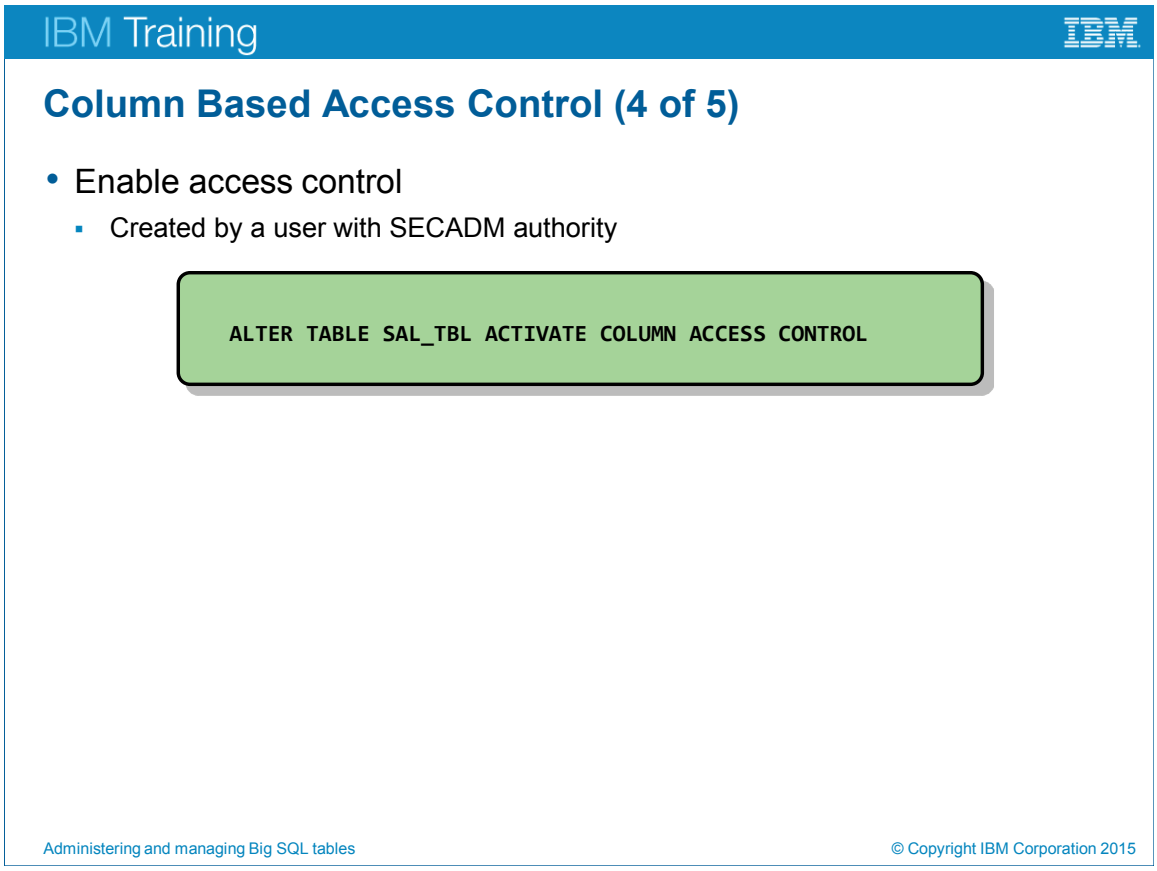
Column Based Access Control (3 of 5)

- Create permissions
  - Created by a user with SECADM authority
  - Create a mask on the salary column
    - If the user belongs to the *manager* role
      - Show the salary column
    - Else
      - Show 0.00

```
CREATE MASK SALARY_MASK ON HADOOP.SAL_TBL FOR
COLUMN SALARY RETURN
CASE WHEN VERIFY_ROLE_FOR_USER(SESSION_USER,'MANAGER') = 1
THEN SALARY
ELSE 0.00
END
ENABLE
```


Column Based Access Control (3 of 5)

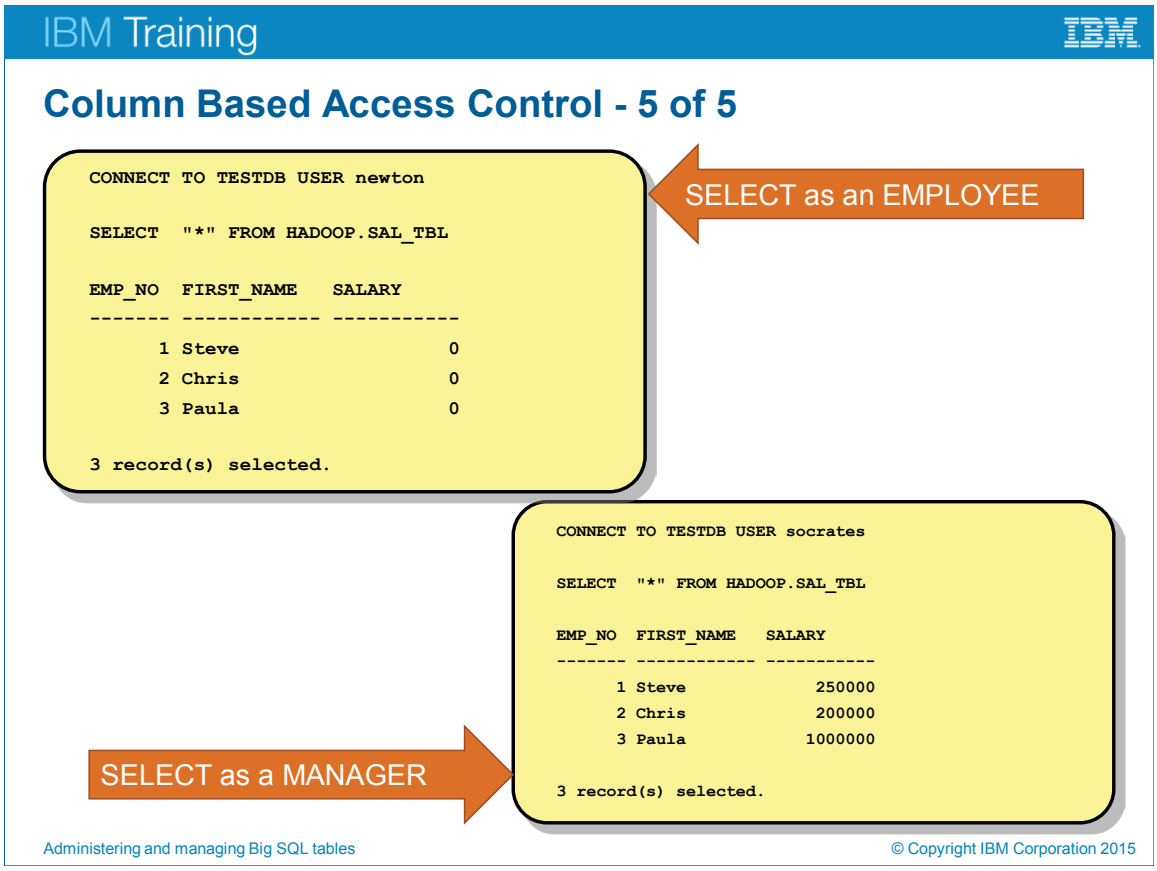
With the roles created, you will now create the permissions. The user creating these permission must have SECADM authority. Column based access control is handled by creating a mask on the column you wish to hide. In this case, we only wish to show the salary columns to the users in the *manager* role. Otherwise, return 0.00.



Column Based Access Control (4 of 5)

Just as with row based, you also need to enable the access control on the column.

```
ALTER TABLE SAL_TBL ACTIVATE COLUMN ACCESS CONTROL
```



Column Based Access Control (5 of 5)

When the user in the employee role selects from the table, he gets the value of 0 for the salary column. When the user in the manager role selects from the table, he gets the salary for each employee.

IBM Training

Checkpoint

1. Which one should you run first, ANALYZE or EXPLAIN?
2. What are some ways to reduce memory usage of the ANALYZE command?
3. True or False? The FOR SNAPSHOT clause of the EXPLAIN statement also takes a snapshot and collect other explain information.
4. What are the ways to create explain tables?
5. What is the difference between row based access control and column based access control in terms of the data the user can see?

Checkpoint

IBM Training

Checkpoint solutions

1. Which one should you run first, ANALYZE or EXPLAIN?
   - ANALYZE, because it will update the statistics for the query optimizer to use for choosing the most appropriate access plan.
2. What are some ways to reduce memory usage of the ANALYZE command?
   - Run on columns that are part of predicates (omit columns from SELECT)
   - Turn off distribution statistics if you don't need it
3. True or False? The FOR SNAPSHOT clause of the EXPLAIN statement also takes a snapshot and collect other explain information.
   - False. The FOR SNAPSHOT does not collect any other explain information.
4. What are the ways to create explain tables?
   - Run the SYSPROC.SYSINSTALLOBJECT procedure
   - Run the EXPLAIN.DDL file
5. What is the difference between row based access control and column based access control in terms of the data the user can see?
   - Row based access control – user does not even know about the data that they cannot see.
   - Column based access control – column values are replaced with a default dummy value

Checkpoint solutions

IBM Training

Demonstration 1

Securing your data with Big SQL Fine-Grained Access Control

At the end of this demonstration, you will be able to:

- Use column masking and row based access control to restrict access to your data

Demonstration 1: Securing your data with Big SQL Fine-Grained Access Control

Demonstration 1: Securing your data with Big SQL Fine-Grained Access Control

Purpose:
Big SQL offers administrators additional SQL security control mechanisms for row and column access through the definition of ROLES and GRANT/REVOKE statements. In this demonstration, you will mask information about gross profits for sales from all users who are not a MANAGER. That is, all users with SELECT privileges will be able to query the SLS_SALES_FACT table, but information in the GROSS_PROFIT column will display as 0.0 unless the user was granted the role of a MANAGER. Any user who is a MANAGER will be able to see the underlying data values for GROSS_PROFIT.

To complete this demonstration, you must have access to multiple user accounts. Examples in this demonstration are based on the following user IDs (in addition to biadmin):

bigsql, which has SECADM authority for your database environment.

user1 and *user2*, which have USER privileges for BigInsights. The password for these two are simply: password

In addition, prior to starting this demonstration, you must have created the SLS_SALES_FACT and the MRK_PROMOTION_FACT table and populated it with data, as described in an earlier demonstration. Section 1 will show you how to load the table if you do not have it created and loaded.

Estimated time: 30 minutes

User/Password: **biadmin/biadmin**
root/dalvm3
user1/password
user2/password
bigsql/ibm2blue

Services Password: **ibm2blue**

Task 1. Ensure that BigInsights is running and the Big SQL tables have been created and populated.

You may skip this task if your environment is still up and running. Refer to past demonstrations if you need to do any of these.

1. Ensure BigInsights is running via Ambari.
2. Ensure you can access the BigInsights Home page
3. Ensure you can access the Big SQL page.
4. The tables SLS_SALES_FACT and MRK_PROMOTION_FACT should have been created and populated with data from a previous exercise. If not, use the script provided under /home/biadmin/labfiles/bigsql/ and complete this now.

Task 2. Using column masking to secure data.

The SQL statements are provided in a script located under **/home/biadmin/labfiles/bigsql/Big_SQL_FGAC/**

You will create two new users, **user1** and **user2** both with the **password** as the password.

1. Switch to the **root** user with **dalvm3**

```
su -
```

2. Create **user1** and **user2**

```
adduser user1
adduser user2
```

3. Set the password as **password**

```
passwd user1
passwd user2
```

Create two roles. These commands must be executed by the bigsql user.

4. Using either JSqsh or the Big SQL console, create these two roles:

```
CREATE ROLE manager;
CREATE ROLE staff;
```

5. Grant SELECT (read) access to the table to users.

```
GRANT SELECT ON sls_sales_fact TO USER user1;
GRANT SELECT ON sls_sales_fact TO USER user2;
```

6. Issue GRANT statements that assign appropriate roles to desired users.

```
GRANT ROLE MANAGER TO USER user1;
GRANT ROLE STAFF TO USER user2;
```

7. Create a column access control rule. Specifically, create a mask called PROFIT_MASK for the GROSS_PROFIT column of the MYBIGSQL.SLS_SALES_FACT table that will display a value of 0.0 to any user who queries this column that is not a MANAGER.

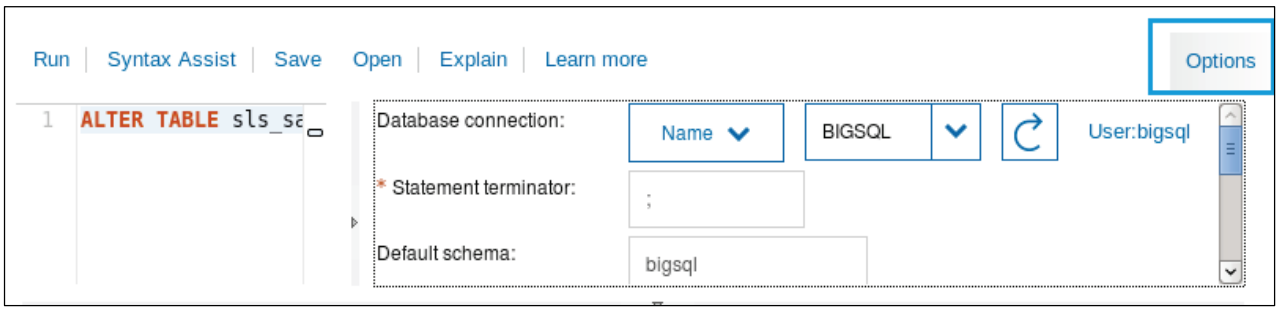
```
CREATE MASK PROFIT_MASK ON
sls_sales_fact
FOR COLUMN gross_profit
RETURN
CASE WHEN VERIFY_ROLE_FOR_USER(SESSION_USER,'MANAGER') =
1
THEN gross_profit
ELSE 0.0
END
ENABLE;
```

8. Issue the following ALTER TABLE statement to activate the column based access control restriction.

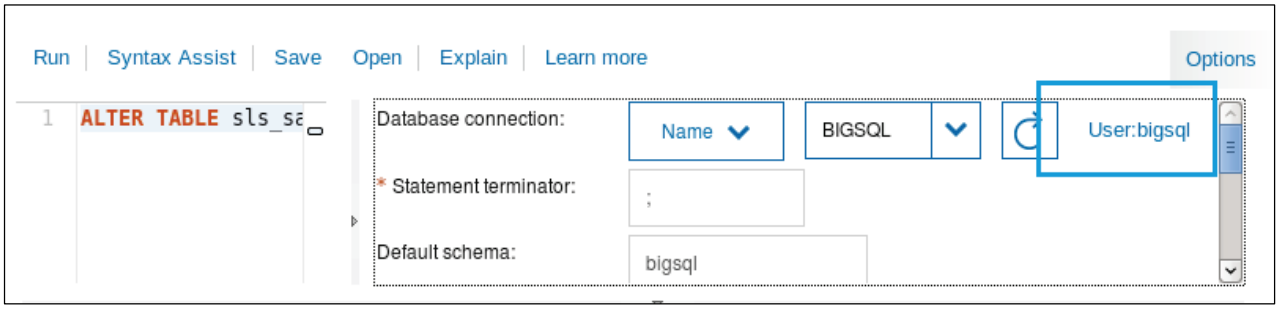
```
ALTER TABLE sls_sales_fact ACTIVATE COLUMN ACCESS
CONTROL;
```

Now you are ready to test the results of your work by querying the SLS_SALES_FACT table using different user accounts. You will do this using the Big SQL page.

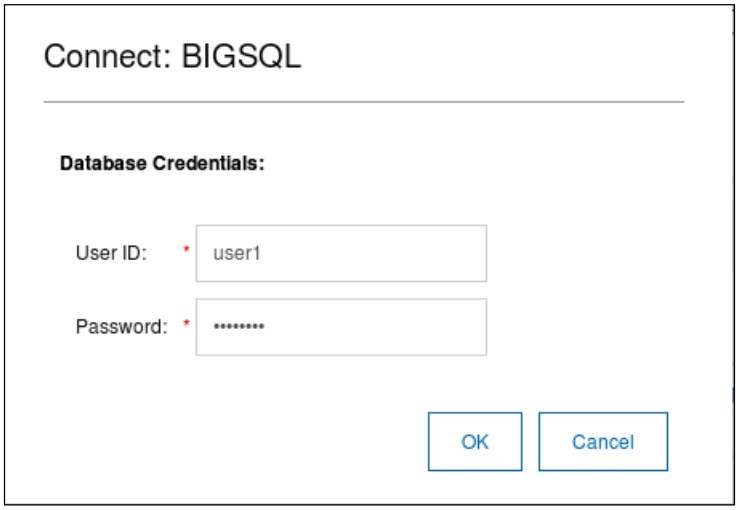
9. Click on the **Options** tab.



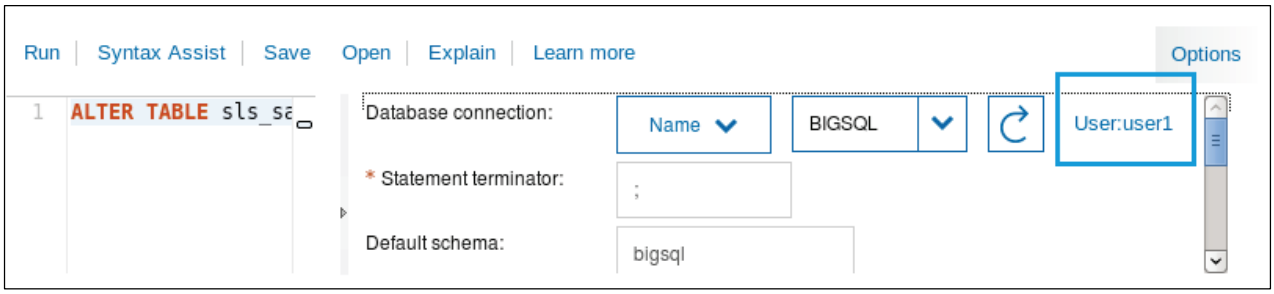
10. Click on the **user:bigsql** link to change to another user.



11. Change to the **user1** account. The password is **password**.



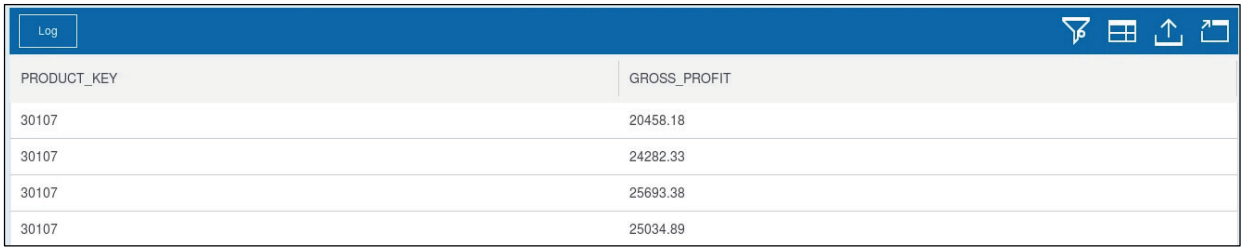
12. Once you have logged in as **user1**, the link should reflect that.



13. Collapse the **Options** pane.
14. Run the following query as user1.

```
select product_key, gross_profit
from sls_sales_fact
where quantity > 5000 fetch first 5 rows only;
```

15. Inspect the results, and note that various data values appear in the GROSS_PROFIT column. This is what you should expect, because USER1 is a MANAGER, and your column mask rule allows MANAGERs to see the data values for this column.



16. Using the same steps as before, switch to **user2/password.**
17. Run the same query as before.

```
select product_key, gross_profit
from sls_sales_fact
where quantity > 5000 fetch first 5 rows only;
```

18. Inspect the results. Note that the GROSS_PROFIT values are masked (appearing as 0.0). Again, this is what you should expect, because USER2 has a STAFF role. Any users who are not MANAGERs are not allowed to see values for this column.

19. Switch to the bigsql account and deactivate the column access restriction.

```
ALTER TABLE sls_sales_fact DEACTIVATE COLUMN ACCESS
CONTROL;
```

Task 3. Using row-based access control.

The effort required to implement row-based access control rules is similar. You will explore a row-based scenario now using the MRK_PROMOTION_FACT table you created and populated in an earlier lab (or in this lab). After implementing this example, you'll see that SELECT statements issued by user1 will only return rows related to a specific retailer key. Restrict consultants (CONSULT role users) to accessing only rows. For retailer key 7166 commands must be executed from bigsql user ID. A valid ID must exist for user1. In this demonstration, user1 is a consultant.

1. Create a new role named CONSULT.

```
CREATE ROLE CONSULT;
```

2. Grant SELECT (read) access to the table to user1 and user2.

```
GRANT SELECT ON mrk_promotion_fact TO USER user1;
GRANT SELECT ON mrk_promotion_fact TO USER user2;
```

3. Issue GRANT statements that assign appropriate roles to desired users. In this case, assign user1 CONSULT role. Do not assign any role to user2.

```
GRANT ROLE CONSULT TO USER user1;
```

4. Create a row access control rule. Specifically, restrict read operations on mybigsql.mrk_promotion_fact to users with the CONSULT role. Furthermore, allow such users to only see rows in which RETAILER_KEY column values are 7166.

```
CREATE PERMISSION RETAILER_7166
ON mrk_promotion_fact
FOR ROWS
WHERE(VERIFY_ROLE_FOR_USER(SESSION_USER,'CONSULT') = 1
AND
retailer_key = 7166)
ENFORCED FOR ALL ACCESS
ENABLE;
```

5. Issue the following ALTER TABLE statement to activate the row based access control restriction.

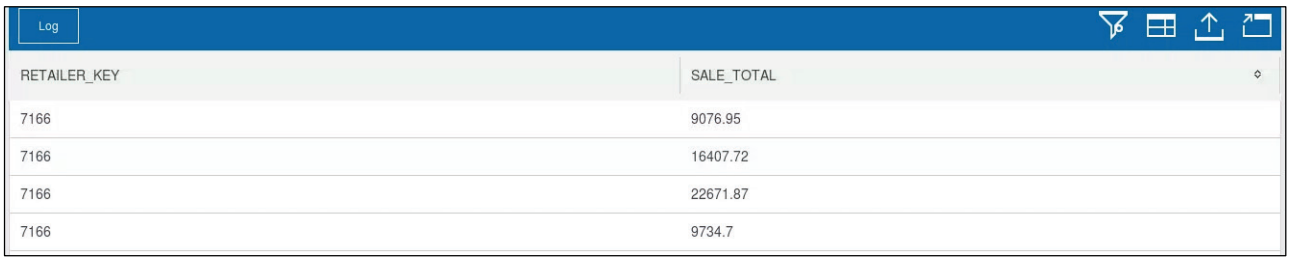
```
ALTER TABLE mrk_promotion_fact ACTIVATE ROW ACCESS
CONTROL;
```

Now you are ready to test the results of your work by querying the table.

6. Switch to the **user1** user with the password = **password**.
7. Run the query as user1:

```
select retailer_key, sale_total
from mrk_promotion_fact
where rtl_country_key = 90010
fetch first 100 rows only;
```

8. Inspect the results, and note that only rows for retailer 7166 appear. This is what you should expect, because USER1 is associated with the CONSULT row.



9. Switch to **user2** with the password = **password**.
10. Run the query again.

```
select retailer_key, sale_total
from mrk_promotion_fact
where rtl_country_key = 90010
fetch first 100 rows only;
```

11. Inspect the results. Note that the query runs successfully but returns no rows. Why? The row access control mechanism you implemented specified that only users of the CONSULT row are permitted to see data from the data and that data would be restricted to rows related to a specific RETAILER_KEY value.
12. Switch back to the **bigsql** user and deactivate the access control:

```
ALTER TABLE mrk_promotion_fact DEACTIVATE ROW ACCESS
CONTROL;
```

Results:
In this demonstration, you masked information about gross profits for sales from all users who are not a MANAGER. You accessed multiple user accounts, including *bigsql*, which has SECADM authority for your database environment.

IBM Training

Unit summary

- Understanding Big SQL data access plans
- Using Big SQL statistics to improve query performance
- Controlling data access using column masking and row-based access control

Unit summary

IBM Training

Data federation using Big SQL

IBM BigInsights v4.0

IBM Training

Unit objectives

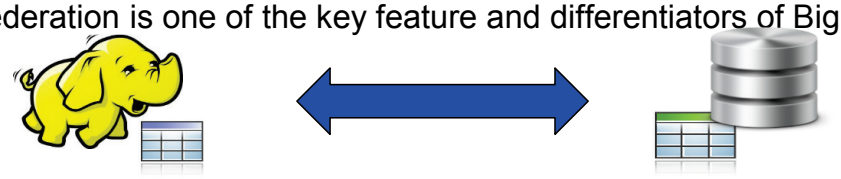
- Understand the concept of Big SQL federation
- List the supported data sources
- Set up and configure a federation server to use different data sources

Unit objectives

IBM Training

Big SQL federation overview

- Users can process SQL queries and statements from multiple data sources.
  - As if they were ordinary tables or views
  - Integrate existing data sources with Big SQL
    - DB2 LUW, Oracle, Teradata and Netezza (IBM PureData for Analytics)
- Federation is one of the key feature and differentiators of Big SQL



```
CREATE NICKNAME get_remote_table for
remote_node.admin.remote_table
```

```
SELECT 'This '
||htbl.col_big||rtbl.col_text||' is
awesome!' FROM get_remote_table rtbl,
hadoop_table htbl
```

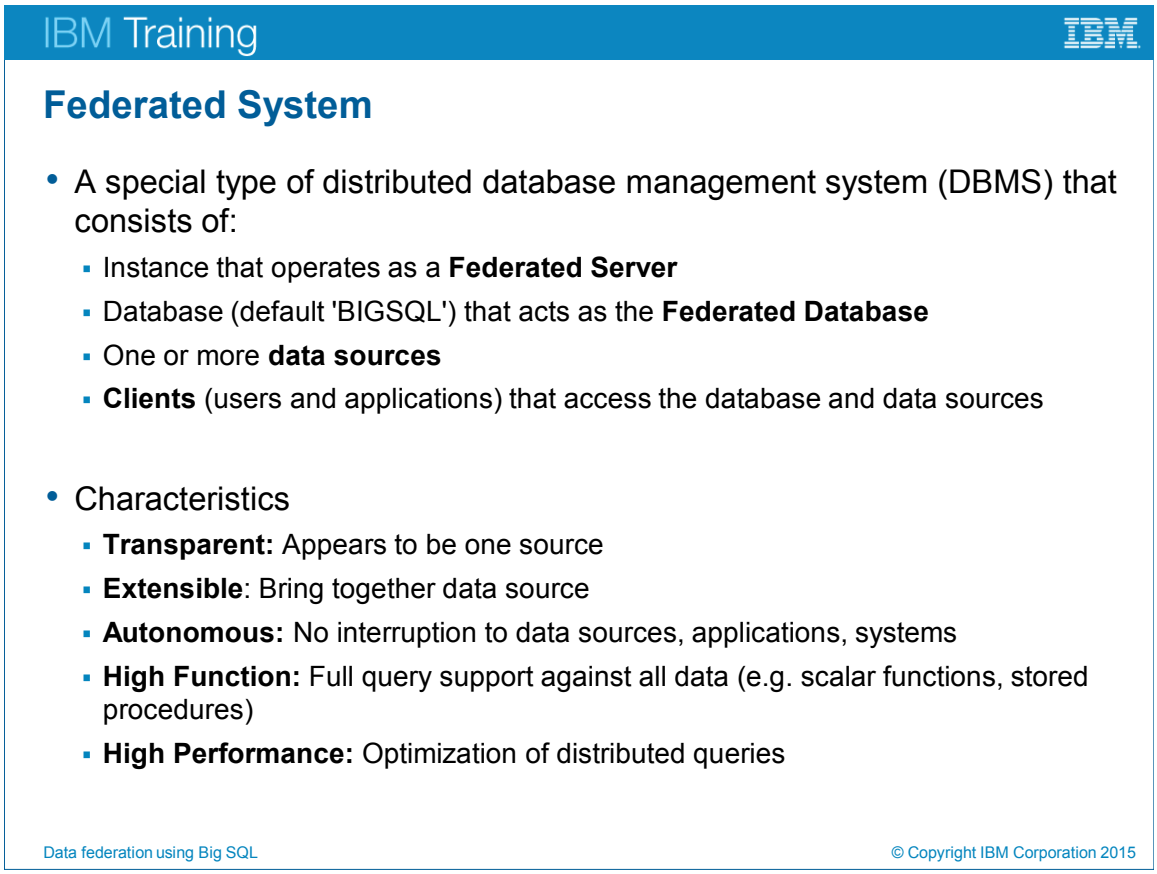
```
This bigsql is awesome!
This BIGsql is awesome!
This Bigsql is awesome!
```

Data federation using Big SQL

Big SQL federation overview

One of the key differentiators of Big SQL is the ability to process queries and statements from multiple data sources as if they were ordinary tables or views. This allows user to integrate existing data sources with Big SQL without the need to migrate the data into Big SQL. Some of the data sources that are currently supported are listed here. This will be covered in more details in a couple of slides.

Here is just a quick example of how federation works in Big SQL. You create a nickname for a remote table. Then you can select from that nickname as well as other Big SQL tables, as if it was a table within Big SQL. The results returned will be a consolidate result.



Federated System

In a federated system, you have an instance that operates as a Federated Server. You also have a Federated Database – the default database is BIGSQL. You have one or more data sources. You have your clients who are the users and applications that access the database and the data sources.

These are the characteristics of a federated system:

- Transparent: appears to be one source to the user
- Extensible: brings together data sources
- Autonomous: no interrupted to data sources, applications, systems
- High function: full query support against all data such as scalar functions and stored procedures
- High performance: optimization of distributed queries

IBM Training

Supported data sources

- Big SQL supports the following data sources

Data source	Supported versions
DB2 LUW	9.7, 9.8, 10.1, 10.5
Oracle	11g, 11gR1, 11gR2
Teradata	12, 13
IBM PureData for Analytics (Netezza)	4.6, 5.0. 6.0, 7.2
MS SQL Server	2014

Data federation using Big SQL

Supported data sources

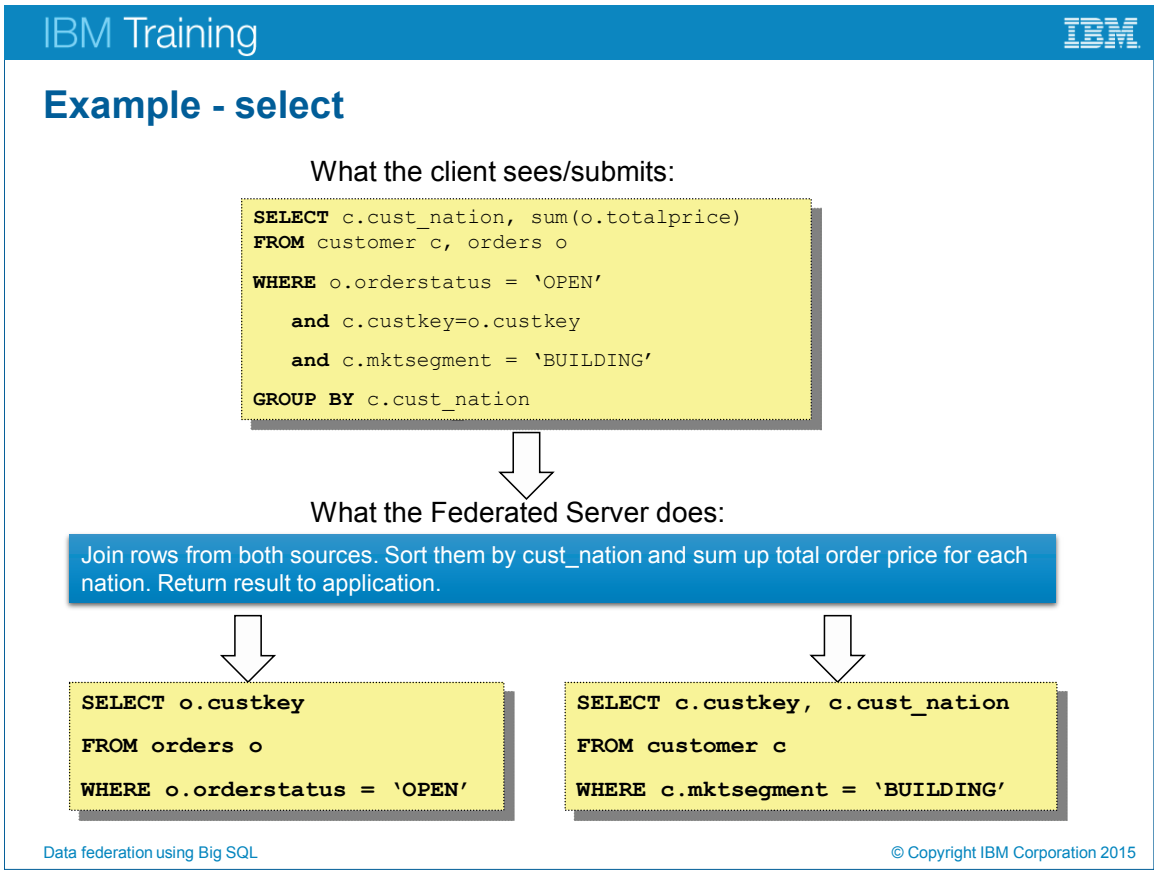
These are the currently supported data sources by Big SQL.

DB2 LUW - supported versions are 9.7, 9.8. 10.1, and 10.5

Oracle - supported versions are 11g, 11gR1, 11gR2

Teradata - 12 and 13

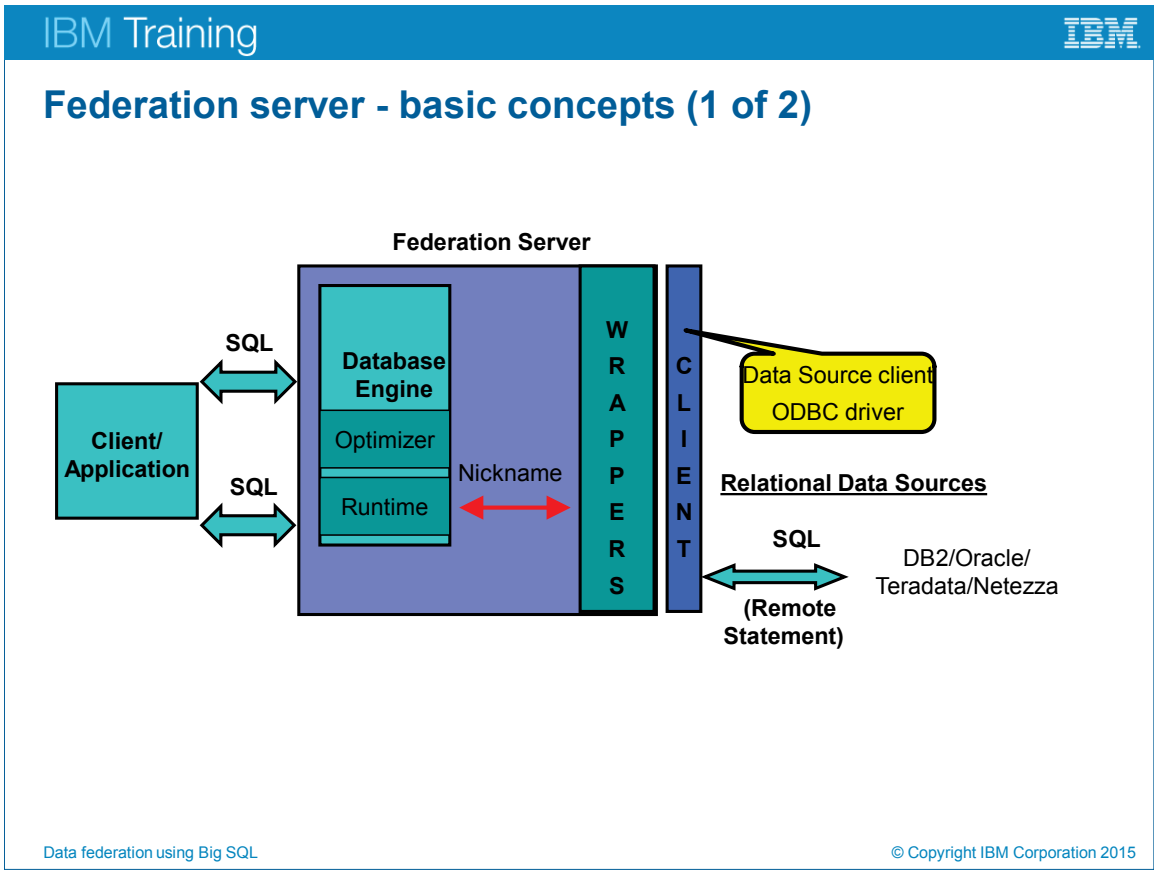
IBM PureData for Analytics (Netezza) – 4.6, 5.0, 6.0, 7.2



Example - select

Before we go further, here is an example of this in action. In the query at the top, the query looks just like any ordinary query. You would not be able to tell that the data from the two tables selected were from different data sources. Big SQL combines the results from both data sources groups them appropriately before returning it to the application. This transparency makes it easy to work with multiple data sources.

This is just one example of how the query optimizer can choose for the most efficient query access plan. You will see how the query optimizer plays a role in the next couple of slides.



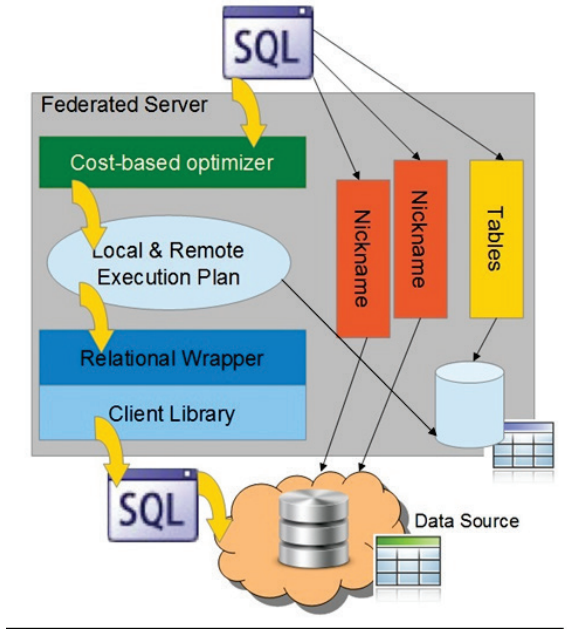
Federation server - basic concepts (1 of 2)

Here is a diagram of the federation server. Let us go from the left to right. You have your client application that queries against the federation server. The database engine has the nicknames from the wrappers of the various data sources. The wrappers essentially communicates with the data sources to retrieve the data via remote SQL statements.

IBM Training

Federation server - basic concepts (2 of 2)

- Nicknames appear as tables
- Execution plans chosen by optimizer
- Optimizer decides how to distribute work
  - Between federated server and data sources
  - Part on federated server and part on data sources
  - Cost-based pushdown of operation



Data federation using Big SQL

Federation server - basic concepts (2 of 2)

The nicknames that you create from the data sources appears as tables (there is no distinction from the user's perspective). The execution plan is chosen by the optimizer in the database engine. The optimizer decides how to distribute the work. It can decide either to have the federated server do all the work, or have the data sources do all the work. It can also decide to have some of the work done on the federated server, and some on the data sources.

IBM Training

Federation server terminologies (1 of 3)

- **Wrapper**
  - A library to access a particular type of data sources
    - A library of routines are used to access data source
  - Register in the federated database using CREATE WRAPPER statement
    - One wrapper to be registered for each data source type, regardless of the number of data sources of that type
  - For data sources that support SQL
    - Queries are submitted in SQL
  - For other data sources
    - Queries are translated into the native query language or native API calls

Data source	Library files	
DB2 LUW	libdb2drdaU.so libdb2drda.so libdb2drdaF.so	
Oracle	libdb2net8U.so libdb2net8.so libdb2net8F.so	
Teradata	libdb2teradataU.so libdb2teradata.so *libdb2teradataF.a	*requires djxlink script to generate **libdb2teradataF.so** on Unix/Linux
Netezza	libdb2rcodbcU.so Libdb2rcodbc.so libdb2rcodbcF.so	

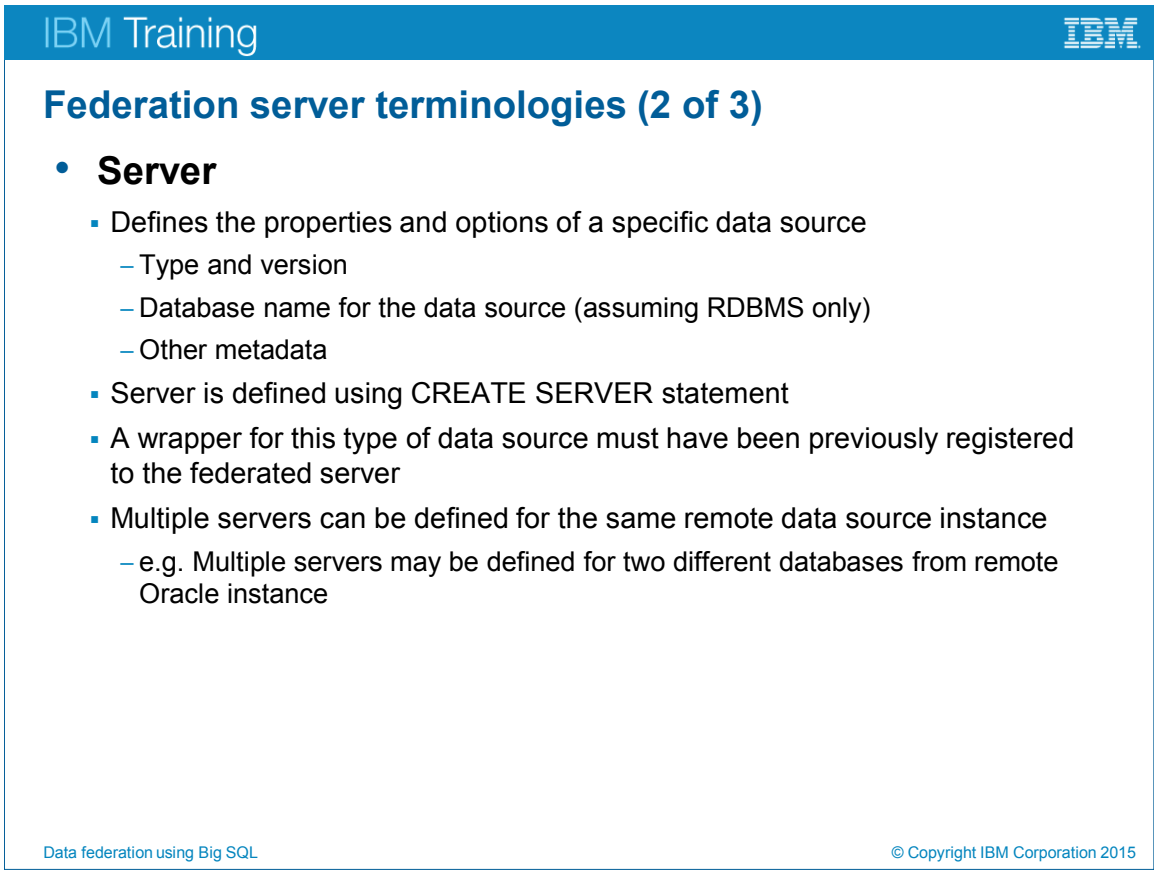
Data federation using Big SQL

Federation server terminologies (1 of 3)

Let's get a bit deeper into how this all works so well with some terminologies. A wrapper is basically a library to access a particular type of data source. Obviously, there are different wrappers for different data sources that are supported. Wrappers are essentially a library of routines that are used to access the data source. You register the wrappers in the federated database using the CREATE WRAPPER statement. One wrapper has to be registered for each data source type, regardless of the number of data sources of that type.

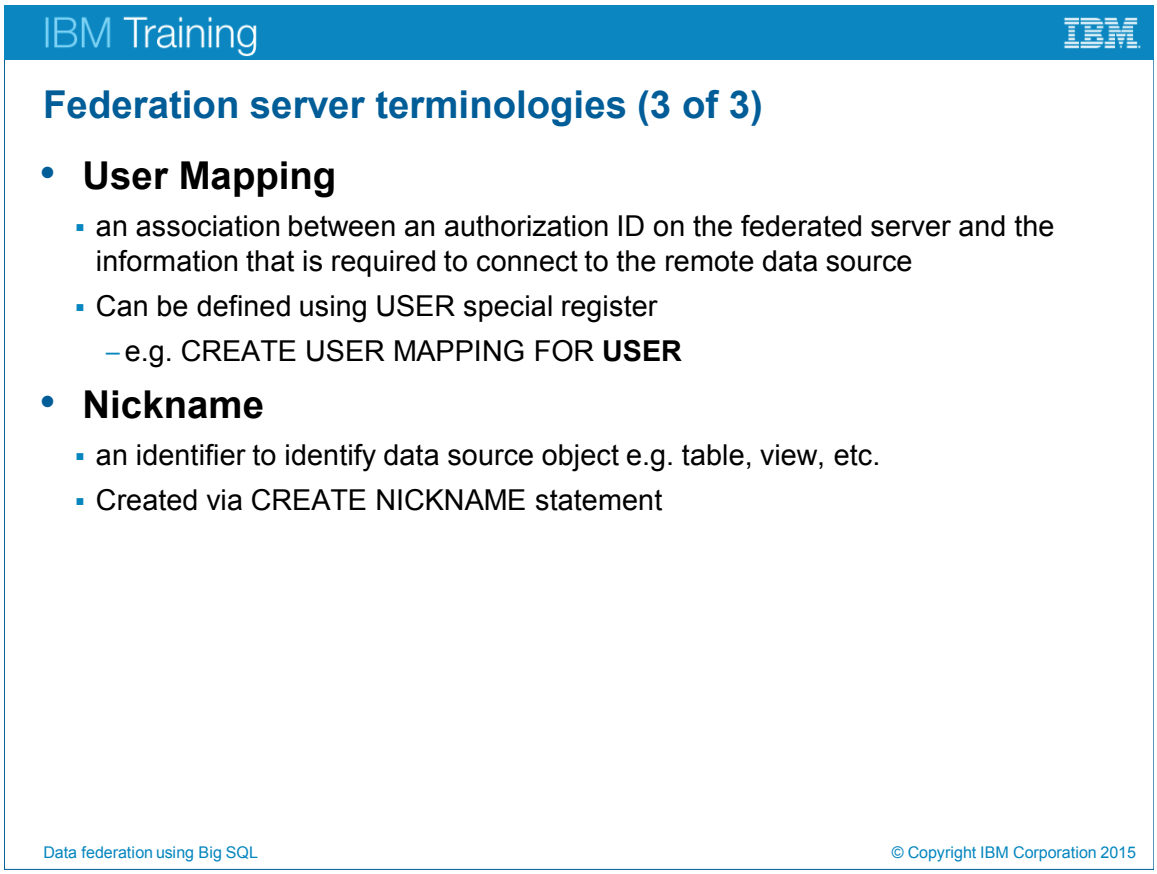
For data sources that support SQL, the queries are submitted in SQL. For other data sources, the queries are translated into the native query language or native API calls.

Here is a table that shows the library files that are required for each of the data source type.



Federation server terminologies (2 of 3)

Once a wrapper is registered, there needs to be a way for the user or application to connect to the data source. The server acts as the medium between the application and the data source. You define the properties and the options of specific data source. The server is defined using the CREATE SERVER statement. A wrapper must have been previously registered before you can use the CREATE SERVER statement. Multiple servers can be defined for the same remote data source instance. For example, multiple servers may be defined for two different databases from the remote Oracle instance.



Federation server terminologies (3 of 3)

User mapping is an association between the authorization ID and the information that is required to connect to the remote data source. This step may be optional depending on how the data source is set up. If the environment uses federated trusted contexts and proxy authentication, none or only a few user mappings may be required. You can define the user using the USER special register or a specific user such as “Mary” or PUBLIC.

We have seen nicknames already. Nicknames are essentially an identifier to a data source object such as a table or a view. Each object should have a unique nickname. You create nicknames using the CREATE NICKNAME statement.

IBM Training

Implication of BINARY collation

- Supports only Binary collation sequence (Hadoop compatibility)
  - CREATE DB BIGSQL USING CODESET UTF-8 TERRITORY US COLLATE USING BINARY
- Binary collation affects string data types
- Binary collation is blank sensitive
  - Blank sensitivity (preserved trailing blanks, no padding)
    - With BINARY collation: 'ABC' != 'ABC '
    - With native database current behaviour: 'ABC' = 'ABC '
- If data source collation is the **same** as Big SQL collation
  - Set *COLLATING_SEQUENCE='Y'* from server definition option
  - More can be pushed down if the collating sequence of the remote system is the same as the collating sequence of the federated database
- If data source collation is **different** to Big SQL collation
  - Set *COLLATING_SEQUENCE='N'* from server definition option

Data federation using Big SQL

Implication of BINARY collation

Big SQL only supports the binary collation sequence for Hadoop compatibility.

Collating sequence is a defined ordering for character data that determines whether a character sorts higher, lower, or the same as another character. Basically, this affects the string data types.

For example, binary collation is blank sensitive. This means that when you compare two strings, if there is a space or blank, it will be compared.

‘ABC’ does not equal ‘ABC ‘ (with a space at the end of the second ABC).

If the data source collation is the same as the Big SQL collation, then set the COLLATING_SEQUENCE=‘Y’ from the server definition options.

If the collation is different, then set the COLLATING_SEQUENCE=‘N’.

This allows the optimizer to determine the best cost-based access plan for the query. It can choose to push down the query to the data source or choose to handle it. Obviously, there are other factors involved in making that decision, such as volume of data, speed of the data source, bandwidth, etc.

IBM Training

Implication of BINARY collation

- Except for Netezza, which has the Binary collation fully compatible with Big SQL, most data sources do not have a compatible collation
- The last column shows the recommended configuration from issuing CREATE SERVER statement

Data source	Collation Sequence	Compatible with Binary collation On Blank Sensitivity	Compatible with Binary collation On Empty string and NULL	Recommended Cofiguration (from CREATE SERVER)
DB2 LUW	Identity	N	Y	COLLATING_SEQUENCE=N PUSHDOWN = Y
Oracle	Binary	Y	N	COLLATING_SEQUENCE=N PUSHDOWN = Y
Teradata	ASCII	N	Y	COLLATING_SEQUENCE=N PUSHDOWN = Y
Netezza	Binary	Y	Y	COLLATING_SEQUENCE=Y PUSHDOWN = Y

Data federation using Big SQL

Implication of BINARY collation

Except for Netezza, most data sources do not have a compatible collation with Big SQL. You will need to specify the correct configuration from the CREATE SERVER statement when creating the server for each data source. The table shows the recommended configuration from issuing the CREATE SERVER statement.

DB2 uses the identity collation sequence, and is not compatible with the binary collation on blank sensitivity but it is compatible on empty string and NULL. The COLLATION_SEQUENCE should be set equals to N and the PUSHDOWN should be Y.

Oracle uses the binary collation, blank sensitivity is compatible, but empty string and NULL is not. Set COLLATING_SEQUENCE='N' AND PUSHDOWN='Y'

Teradata uses ASCII binary collation. Set N and Y respectively.

Netezza uses binary. Set Y and Y for both of the configurations.

IBM Training

Federation server configurations

- Enable the federated server to access data sources
  - `UPDATE DATABASE MANAGER CONFIGURATION USING FEDERATED YES`
- Install clients software from various data sources
  - ODBC driver, Oracle, Teradata clients can be download respective websites
- Setup environment variables for the clients in db2dj.ini
  - Default path: *instancehome/sqllib/cfg/db2dj.ini*
  - Can be changed via registry variable DB2_DJ_INI
- Run djxlink script as root – for some wrappers e.g. Teradata
  - Default path: *instancehome/sqllib/bin*
  - e.g. *instancehome/sqllib/bin> djxlinkTeradata*
    - Will generate *libdb2teradataF.so* file for Teradata wrapper
- Do a quick test
  - Make sure the client tool can connect to data source before creating any federated objects

Data federation using Big SQL

Federation server configurations

Here are some federation server configurations.

The first one is to enable the federated server to access data sources. You need to do this before you can work with data sources.

You will need to install the client software from the various data sources before you can create a wrapper from it. You will need to get them from their respective website.

The setup environment variables for the all the clients is contained in the db2dj.ini file. It is located in the default path instancehome/sqllib/cfg/db2db.ini.

For some wrappers such as Teradata, you will need to run the djxlink script as root to have it create the wrapper.

It is good to do a quick test to make sure that the client tool can connect to the data source before creating any federated objects.

IBM Training

Required configurations from data sources

Data source	Variables in db2dj.ini	Instance profile	Others
DB2 LUW	NA	NA	**- Catalog a DB2 node entry** e.g. CATALOG TCPIP NODE *db2_node* REMOTE *system42* SERVER *db2tcp42* **- Catalog the remote DB2 database** e.g. CATALOG DATABASE *DB2DB390* AT NODE *db2_node*
Oracle	**ORACLE_HOME** e.g. */opt/oracleclient/* **TNS_ADMIN** *(default)* *$ORACLE_HOME/network/admin*	**DB2LIBPATH** e.g. */opt/oracleclient/lib*	
Teradata	**TERADATA_LIB_DIR** e.g. */opt/teradata/client/13.10/lib64*		**Add Teradata hostnames in /etc/hosts** 9.xx.xxx.xxx terahost1 teraCOP1 9.xx.xxx.xxx terahost2 teraCOP2
Netezza	**NZ_ODBC_INI_PATH** e.g. *$HOME/bin* **ODBCINI** e.g. *$HOME/bin/odbc.ini*	**DB2LIBPATH** e.g. */opt/oemclient/odbc60/lib*	**Add a new entry in odbc.ini** e.g. `[ODBC Data Sources]` `NZSQL = NetezzaSQL` `[NZSQL]` `Driver =` `/usr/local/nz/lib64/libnzodbc.so` `Description = NetezzaSQL ODBC` `Servername = <server name>` `Port = 5480` `Database = NZDB` `Username = admin` `Password = *****`

Data federation using Big SQL

Required configurations from data sources

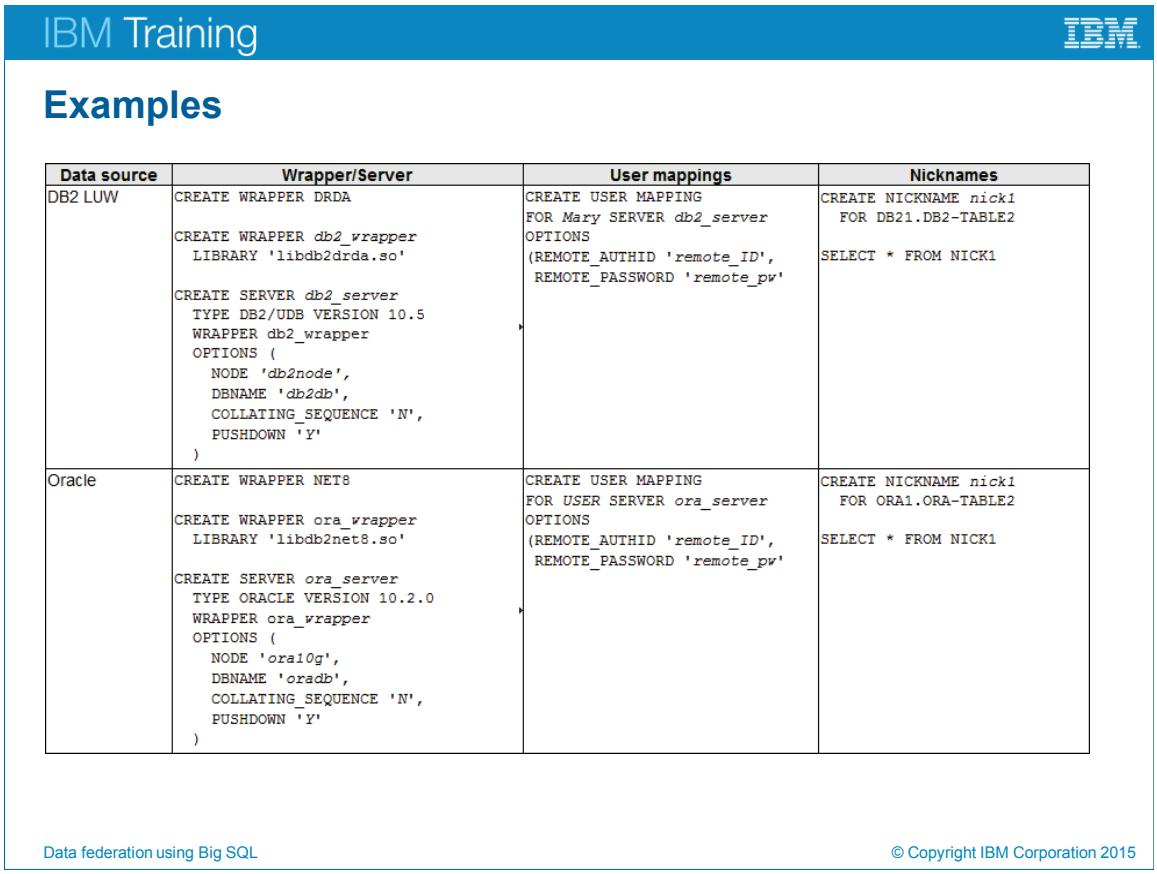
Here are the required configuration from the data sources.

DB2 LUW has two required configurations to define a catalog node and a catalog remote database to be able to connect to the remote data source.

Oracle has some variables in the db2dj.ini file that needs to be defined such as the ORACLE_HOME and the TNS_ADMIN variable. It also needs to specify the DB2LIBPATH in the instance profile.

Teradata needs to specify the TERADATA_LIB_DIR as well as adding the Teradata hostnames in the /etc/hosts/

Netezza requires the NZ_ODBC_INI_PATH and the ODBCINI path to be specified as well as the DB2LIBPATH. You also need to add a new entry to the odbc.ini file to include the connection properties to be able to connect to a Netezza server.



Examples

Here are two specific examples for DB2 LUW and Oracle.

DB2 LUW

Wrapper / Server:

```
CREATE WRAPPER DRDA
CREATE WRAPPER db2_wrapper LIBRARY 'libdb2drda.so'
CREATE SERVER db2_server
     TYPE DB2/UDB VERSION 10.5
     WRAPPER db2_wrapper
     OPTIONS (
          NODE 'db2node',
          DBNAME 'db2db',
          COLLATING_SEQUENCE 'N'',
          PUSHDOWN 'Y'
      )
```

User mappings:

```
CREATE USER MAPPING
FOR Mary SERVER db2_server
OPTIONS
 (REMOTE_AUTHID 'remote_id'',
  REMOTE_PASSWORD 'remote_password''
 )
```

Nicknames:

```
CREATE NICKNAME nick1 FOR DB21.DB2-TABLE2
SEELCT * FROM NICK1
```

ORACLE

Wrapper/Server:

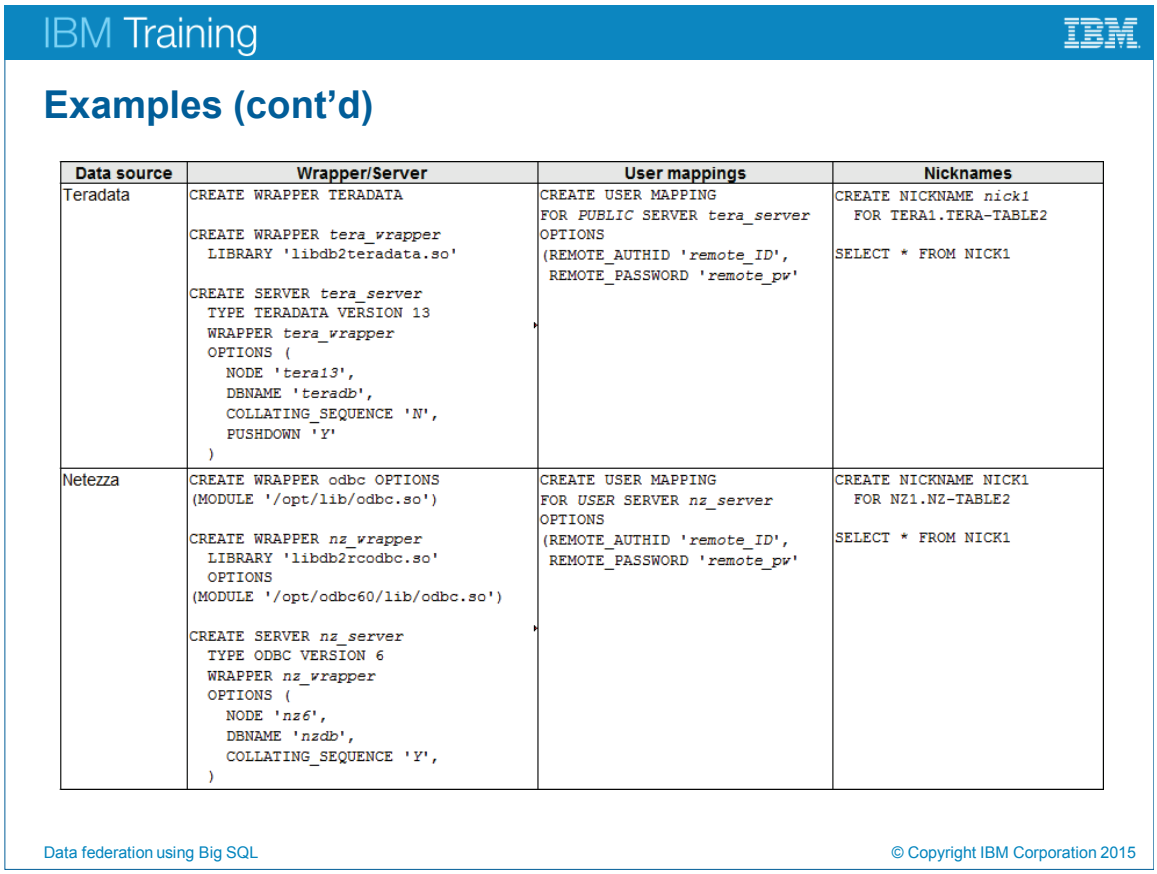
```
CREATE WRAPPER NETS
CREATE WRAPPER ora_wrapper
     LIBRARY 'libdb2net8.so'
CREATE SERVER ora_server
     TYPE ORACLE VERSION 10.2.0
     WRAPPER ora_wrapper
     OPTIONS  (
          NODE 'ora10g',
          DBNAME 'oradb',
          COLLATING_SEQUENCE 'N'',
          PUSHDOWN 'Y''
     )
```

User mappings:

```
CREATE USER MAPPING
FOR USER SERVER ora_server
OPTIONS
 (REMOTE_AUTHID 'remote_ID',
  REMOTE_PASSWORD 'remote_pw'
 )
```

Nicknames:

```
CREATE NICKNAME nick1
     FOR ORA1.ORA-TABLE2
SELECT * FROM NICK1
```



Data source	Wrapper/Server	User mappings	Nicknames
Teradata	CREATE WRAPPER TERADATA CREATE WRAPPER *tera_wrapper* LIBRARY 'libdb2teradata.so' CREATE SERVER *tera_server* TYPE TERADATA VERSION 13 WRAPPER *tera_wrapper* OPTIONS (NODE '*tera13*', DBNAME '*teradb*', COLLATING_SEQUENCE '*N*', PUSHDOWN '*Y*')	CREATE USER MAPPING FOR *PUBLIC* SERVER *tera_server* OPTIONS (REMOTE_AUTHID '*remote_ID*', REMOTE_PASSWORD '*remote_pw*'	CREATE NICKNAME *nick1* FOR TERA1.TERA-TABLE2 SELECT * FROM NICK1
Netezza	CREATE WRAPPER odbc OPTIONS (MODULE '/opt/lib/odbc.so') CREATE WRAPPER *nz_wrapper* LIBRARY 'libdb2rcodbc.so' OPTIONS (MODULE '/opt/odbc60/lib/odbc.so') CREATE SERVER *nz_server* TYPE ODBC VERSION 6 WRAPPER *nz_wrapper* OPTIONS (NODE '*nz6*', DBNAME '*nzdb*', COLLATING_SEQUENCE '*Y*',)	CREATE USER MAPPING FOR *USER* SERVER *nz_server* OPTIONS (REMOTE_AUTHID '*remote_ID*', REMOTE_PASSWORD '*remote_pw*'	CREATE NICKNAME NICK1 FOR NZ1.NZ-TABLE2 SELECT * FROM NICK1

Examples (cont'd)

TERADATA:

Wrapper/Server:

```
CREATE WRAPPER TERADATA
CREATE WRAPPER tera_wrapper
     LIBRARY 'libdb2teradata.so'
CREATE SERVER tera_server
     TYPE TERADATA VERSION 13
     WRAPPER tera_wrapper
     OPTIONS (
          NODE 'tera13',
          DBNAME 'teradb',
          COLLATING_SEQUENCE 'N'',
          PUSHDOWN 'Y'
     )
```

User mappings:

```
CREATE USER MAPPING
FOR PUBLIC SERVER tera_server
OPTIONS
 (REMOTE_AUTHID 'remote_ID',
  REMOTE_PASSWORD 'remote_pw'
 )
```

Nicknames:

```
CREATE NICKNAME nick1
     FOR TERA1.TERA-TABLE2
SELECT * FROM NICK1
```

NETEZZA

Wrapper/Server:

```
CREATE WRAPPER odbc OPTIONS
(MODULE '/opt/lib/odbc.so')
CREATE WRAPPER nz_wrapper
     LIBRARY 'libdb2rcodbc.so'
     OPTIONS
(MODULE '/opt/odbc60/lib/odbc.so')
CREATE SERVER nz_server
     TYPE ODBC VERSION 6
     WRAPPER nz_wrapper
     OPTIONS (
          NODE 'nz6',
          DBNAME 'nzdb',
          COLLATING_SEQUENCE 'Y',
          PUSHDOWN 'Y'
     )
```

User mappings:

```
CREATE USER MAPPING
FOR USER SERVER nz_server
OPTIONS
 (REMOTE_AUTHID 'remote_ID',
  REMOTE_PASSWORD 'remote_pw'
 )
```

Nicknames:

```
CREATE NICKNAME nick1
     FOR NZ1.NZ-TABLE2
SELECT * FROM NICK1
```

IBM Training

Checkpoint

1. What are the characteristics of a federated system?
2. List the currently support data sources.
3. What is the purpose of a wrapper?

Checkpoint

IBM Training

Checkpoint solutions

1. What are the characteristics of a federated system?
   - Transparent
   - Extensible
   - Autonomous
   - High Function
   - High Performance
2. List the currently support data sources.
   - DB2, Oracle, Teradata, IBM PureData for Analytics, MS SQL Server 2014
3. What is the purpose of a wrapper?
   - Provide a library of routines that communicates with the data source.

Checkpoint solutions

IBM Training

Demonstration 1

Setting up Big SQL federation with IBM DB2 10.5 for LUW

At the end of this demonstration, you will be able to:

- Create a federated system to use with DB2 and run queries that span across Big SQL and DB2 tables.

Demonstration 1: Setting up Big SQL federation with IBM DB2 10.5 for LUW

Demonstration 1: Setting up Big SQL federation with IBM DB2 10.5 for LUW

Purpose:
You will set up DB2 as a data source enabling users to send distributed requests to multiple data sources within a single statement. This will involve connecting to DB2 LUW, creating a wrapper to use with DB2, create the server as the definition of the remote DB2 database, map the user id and password at the federated server to the corresponding user id and password at the data source, create a nickname on a remote object and query the remote table from the federated Big SQL server.

Estimated time: 60 minutes

User/Password: **biadmin/biadmin**
root/dalvm3
db2inst1/ibm2blue

Services Password: **ibm2blue**

Task 1. Create and load DB2 tables (if required).

In DB2, you will be creating a table called sls_order_method_dim. This table contains the order method of each product that was sold. You will be joining the data from this table from several tables in Big SQL. Go back to the previous demonstrations to create and load the Big SQL tables.

1. Ensure BigInsights is running via Ambari.
2. Ensure you can access the BigInsights Home page.
3. Ensure you can access the Big SQL page.
   DB2 has been installed on the image. To create a database, you will need to switch user to **db2inst1** with the password **ibm2blue**. If you have a different instance user, switch to that one.
4. Launch a terminal and type in:
   ```
   su db2inst1
   ```
5. As the db2inst1 user, switch to the /home/db2inst1 directory.
   ```
   cd /home/db2inst1
   ```
6. Launch the DB2 CLP:
   ```
   db2
   ```

7. Type in this command to create a database in DB2.

```
create database db2_db
```

8. Connect to the database.

```
connect to db2_db
```

9. Create the table (Note that the statement is provided in a script under /home/biadmin/labfiles/bigsql/Big_SQL_Federation).

```
CREATE TABLE db2inst1.sls_order_method_dim (
order_method_key INT NOT NULL, order_method_code INT NOT
NULL, order_method_en VARCHAR(90) NOT NULL ,
order_method_de VARCHAR(90), order_method_fr
VARCHAR(90), order_method_ja VARCHAR(90),
order_method_cs VARCHAR(90), order_method_da
VARCHAR(90), order_method_el VARCHAR(90),
order_method_es VARCHAR(90), order_method_fi
VARCHAR(90), order_metod_hu VARCHAR(90), order_method_id
VARCHAR(90), order_method_it VARCHAR(90),
order_method_ko VARCHAR(90), order_method_ms
VARCHAR(90), order_method_nl VARCHAR(90),
order_method_no VARCHAR(90), order_method_pl
VARCHAR(90), order_method_pt VARCHAR(90),
order_method_ru VARCHAR(90), order_method_sc
VARCHAR(90), order_method_sv VARCHAR(90),
order_method_tc VARCHAR(90), order_method_th
VARCHAR(90))
```

The DB2_Federation statements.sql file is located in the */home/biadmin/labfiles/bigsql/BIG_SQL_Federation* folder.

10. Once the table has been created, load the data using this command:

```
import from
/home/db2inst1/GOSALESDW.SLS_ORDER_METHOD_DIM.txt of del
modified by coldel0x09 insert into
db2inst1.sls_order_method_dim
```

This dataset should have been previously copied into that directory, if not, modify the path for your environment.

11. Quit from the db2 shell, type:

```
quit
```

Task 2. Setting up the federation environment.

By default, Big SQL federation is not enabled. Enable it using the following steps.

1. Using the same terminal, switch to the **bigsql** user with the password **ibm2blue**.
2. Start Big SQL's CLP
   ```
   db2
   ```
3. Enable the federation. Type in:
   ```
   UPDATE DBM CFG USING FEDERATED YES
   ```
4. Quit the CLP
   ```
   quit
   ```
5. Restart the Big SQL service via Ambari or in the terminal:
   ```
   /usr/ibmpacks/current/bigsql/bigsql/bin/bigsql stop
   /usr/ibmpacks/current/bigsql/bigsql/bin/bigsql start
   ```

Task 3. Setting up the connection for DB2 LUW.

For this task, you will set up the connection from the Big SQL federation server to the existing DB2 server on the same machine.

1. Start the Big SQL CLP (remember you need to do this as the **bigsql** user)
   ```
   db2
   ```
   You must first catalog the remote server in the federated server system directory. The DB2 database was installed on the port 50002.
2. Run this command to create a db2node in the local catalog:
   ```
   CATALOG TCPIP NODE db2node REMOTE localhost SERVER 50002
   ```
   Note: Our data source in the demonstration is installed on the same node. In a real world environment, you will most likely have your data source on a different machine. In that case, you will need install the client on your machine that can connect to the remote data source.
3. Catalog the database from the remote server. Type in this command:
   ```
   CATALOG DATABASE db2_db AS db2db AT NODE db2node
   ```
4. Refresh the local catalog by execute this command
   ```
   terminate
   ```
5. Test the connection before continuing. Start the CLP again:
   ```
   db2
   ```

6. Connect to the remote database (db2) from the local database (big sql)
   ```
   CONNECT TO db2db USER db2inst1 USING ibm2blue
   ```
   If you successfully connected, then the remote data source has been cataloged correctly.
7. Quit the CLP:
   ```
   quit
   ```

Task 4. Creating the wrapper and server for the data source.

A wrapper is associated with a single library file consisting of routines that are used to access the data source. They can be written as C++ or Java applications. One wrapper is required for each data source type, regardless of the number of data sources of that particular type. If the data source supports SQL, the queries are submitted in SQL. If the data sources support a different language, then the queries are translated into the native query language or API calls.

1. Start the Big SQL CLP (as the bigsql user)
   ```
   db2
   ```
2. To create a wrapper and a remote server, you will need to be within a local Big SQL database. Connect to the bigsql database. Type in:
   ```
   connect to bigsql
   ```
3. Execute this command in the CLP to create a wrapper for the DB2 data source:
   ```
   CREATE WRAPPER db2wrapper LIBRARY 'libdb2drda.so'
   ```
4. Once the wrapper has been created, define the remote server next:
   ```
   CREATE SERVER db2server TYPE DB2/UDB VERSION 10.5
   WRAPPER db2wrapper AUTHORIZATION "db2inst1" PASSWORD
   "ibm2blue" OPTIONS (DBNAME 'DB2DB', COLLATING_SEQUENCE
   'N', PUSHDOWN 'Y')
   ```
   Note: The user id and password should be escaped to preserve the case sensitivity. Use double quotes for this purpose.

Task 5. Mapping and creating a nickname.

1. Once the remote server has been defined, you need to map the local user id and password to the corresponding user id and password on the remote server. While still connected to the bigsql server, run this command:
   ```
   CREATE USER MAPPING FOR bigsql SERVER db2server
   OPTIONS(REMOTE_AUTHID 'db2inst1', REMOTE_PASSWORD
   'ibm2blue')
   ```

2. With the id and password mapped, you can now create the nickname of the remote database object. Type in the command to create a nickname for the remote table within the DB2 data source:

```
CREATE NICKNAME bigsql.order_method FOR
db2server.db2inst1.sls_order_method_dim
```

Once the nickname has been created, you can use it as you would any other database object (local or remote). In the next section, you will run a query that joins tables from both Big SQL and DB2.

Task 6. Querying tables from both Big SQL and DB2.

For this task, it may be easier to use SQL Editor in the Big SQL console. However, you can still use JSqsh if are comfortable with it.

1. You will drop and recreate the bigsql.sls_product_dim table to ensure we have a clean copy to work with.
2. Drop the table:

```
drop table bigsql.sls_product_dim
```

3. Create the table:

```
-- product dimension table
CREATE HADOOP TABLE IF NOT EXISTS sls_product_dim
( product_key      INT NOT NULL
, product_line_code  INT NOT NULL
, product_type_key   INT NOT NULL
, product_type_code  INT NOT NULL
, product_number   INT NOT NULL
, base_product_key   INT NOT NULL
, base_product_number     INT NOT NULL
, product_color_code INT
, product_size_code  INT
, product_brand_key  INT NOT NULL
, product_brand_code INT NOT NULL
, product_image    VARCHAR(60)
, introduction_date  TIMESTAMP
, discontinued_date  TIMESTAMP
)
ROW FORMAT DELIMITED
FIELDS TERMINATED BY '\t'
LINES TERMINATED BY '\n'
STORED AS TEXTFILE
;
```

4. Load the data:

```
load hadoop using file url
'file:///usr/ibmpacks/bigsql/4.0/bigsql/samples/data/GOS
ALESDW.SLS_PRODUCT_DIM.txt' with SOURCE PROPERTIES
('field.delimiter'='\t') INTO TABLE SLS_PRODUCT_DIM
overwrite;
```

5. Take a moment to look over this query before issuing:

```
SELECT pnumb.product_name, sales.quantity,
meth.order_method_en
FROM
sls_sales_fact sales,
sls_product_dim prod,
sls_product_lookup pnumb,
order_method meth
WHERE
pnumb.product_language='EN'
AND sales.product_key=prod.product_key
AND prod.product_number=pnumb.product_number
AND meth.order_method_key=sales.order_method_key;
```

   The method selects three columns, *product_name*, *quantity*, and *order_method_en*. The third column (*order_method_en*) comes from the DB2 data source. Notice how the *order_method* table appears as a regular table. A user would not be able to tell that it is from a remote data source.

6. Launch the Big SQL console from BigInsights Home via Ambari.
7. Log in to the Big SQL console using **bigsql/ibm2blue**.
8. Inspect the results to see that the tables were joined and the columns were selected.

PRODUCT_NAME	QUANTITY	ORDER_METHOD_EN
Course Pro Putter	587	Telephone
Blue Steel Max Putter	214	Telephone
Course Pro Gloves	576	Telephone
Glacier Deluxe	129	Sales visit
BugShield Natural	1776	Sales visit

9. Go ahead and close any applications and windows that you have opened.

   This concludes the demo. You have successfully set up DB2 as a data source to be used in the same query and joined together with Big SQL tables.

Results:
You setup DB2 as a data source enabling users to send distributed requests to multiple data sources within a single statement. This involved connecting to DB2 LUW, creating a wrapper to use with DB2, created the server as the definition of the remote DB2 database, mapped the user id and password at the federated server to the corresponding user id and password at the data source, created a nickname on a remote object and queried the remote table from the federated Big SQL server.

IBM Training

Unit summary

- Understand the concept of Big SQL federation
- List the supported data sources
- Set up and configure a federation server to use different data sources

Unit summary

IBM Training

Using Big SQL operations on tables managed by HBase

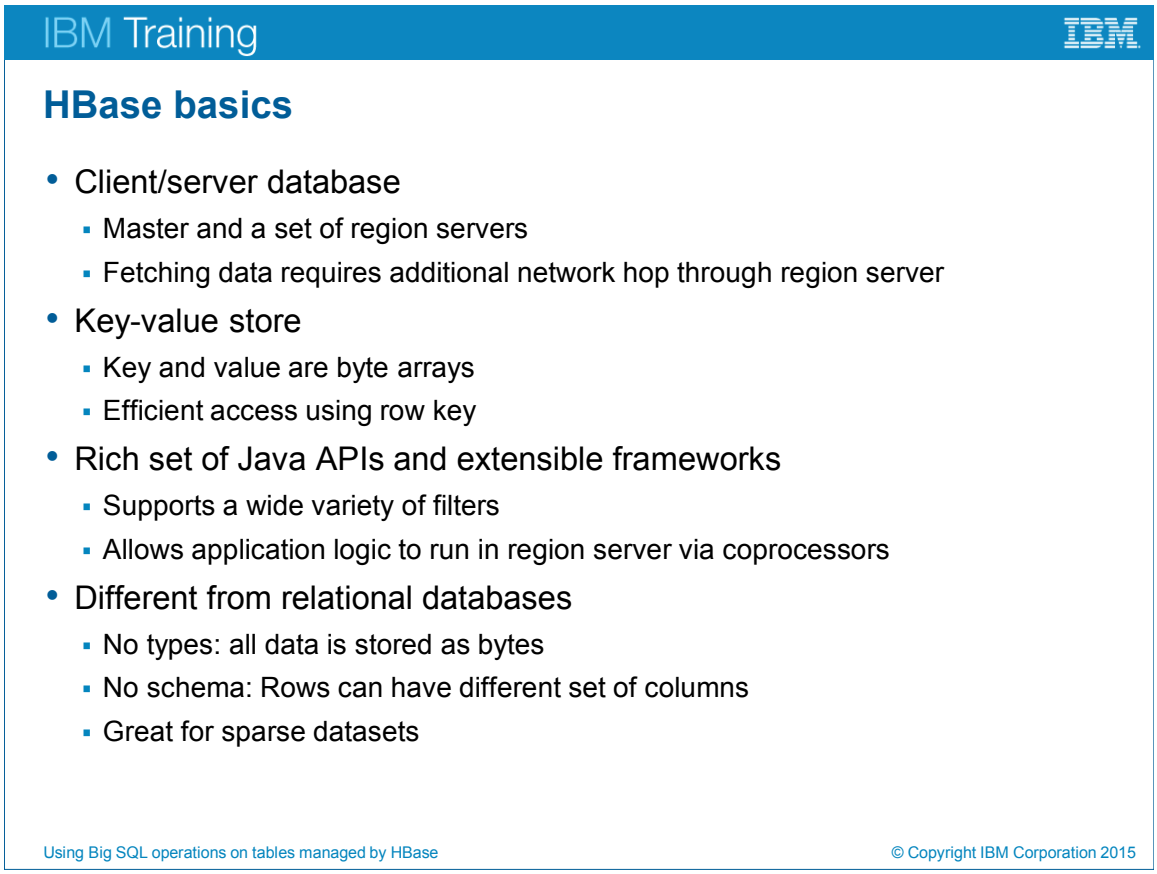
IBM BigInsights v4.0

IBM Training

Unit objectives

- Describe the basic function of HBase
- Issuing basic HBase commands
- Using Big SQL to create and query HBase tables
- Mapping HBase columns to Big SQL

Unit objectives



HBase basics

Big SQL comes with a robust HBase storage handler that understands HBase and leverages its strengths. Big SQL understands how to take your data and query to HBase and make HBase work as efficiently as possible. Before going into details, first look at some basic HBase concepts. HBase utilizes a client/server model. There are two main services in HBase, a master server and region servers. The master coordinates the cluster and performs admin operations. The real workers in the HBase world are the region servers. They each handle a subset of each table's data. Clients talk to region server to access data in HBase.

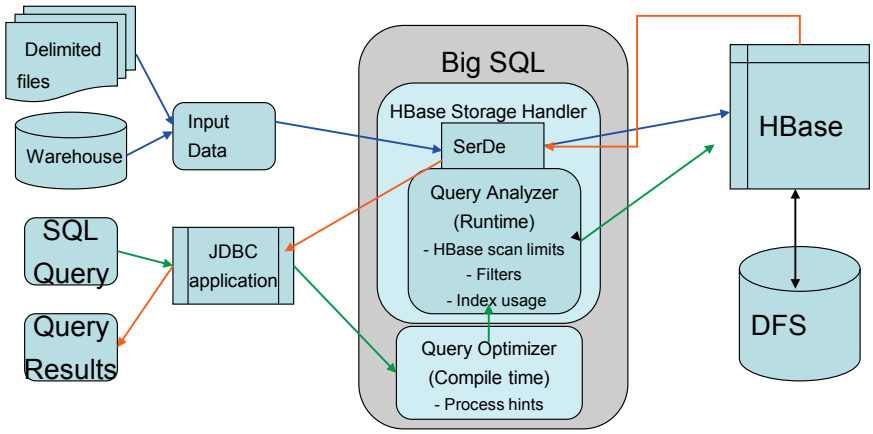
Though it adds a layer on top of the distributed file system, HBase provides a lot of features that makes it capable of low latency random reads and writes. Each row in an HBase table has a row key and HBase can efficiently get to any row given a row key. It has a rich Java API, supports a wide variety of server filters, and provides mechanisms to run application logic on region servers via coprocessor framework.

Another aspect of HBase is how it is different from relational databases. Everything in HBase is stored as bytes. There are no types. There is no schema as each row in HBase can have a different set of columns. HBase is great for sparse datasets.

IBM Training

Big SQL HBase storage handler

- Mapping of SQL to HBase data:
  - Column Mapping
- Handles serialization/deserialization of data (SerDe)
  - Efficiently handles SQL queries by pushing down predicates



Using Big SQL operations on tables managed by HBase

Big SQL HBase storage handler

Big SQL provides a SQL layer on top of HBase. The Big SQL component that talks to HBase is called HBase storage handler.

The first thing that is required for Big SQL to work with HBase is a mapping of data in the HBase table to a logical Big SQL table. When you decide on a column mapping, you are actually assigning a schema and data types to the HBase data thus tying it to a relational model. Then, you need to decide how to interpret the bytes and convert them into the types assigned through the mapping. This process of **Ser**ialization/**De**serialization is handled using a SerDe. Once you have the data model, the next step is to handle the queries efficiently.

In the picture, the blue arrow shows how data flows into HBase. Big SQL can load data from delimited files and warehouse sources. The data passes through the SerDe and gets stored into HBase tables. The green arrows show how a query flows. Big SQL automatically rewrites queries, analyzes predicates, and pushes them down to HBase. The orange arrow shows the results flowing back from HBase to the application after getting deserialized by the SerDe.

IBM Training

Creating a Big SQL table in HBase

- Standard CREATE TABLE DDL with extensions

```
CREATE HBASE TABLE
BIGSQLLAB.REVIEWS (
REVIEWID        varchar(10) primary
key not null,
PRODUCT         varchar(30),
RATING          int,
REVIEWERNAME    varchar(30),
REVIEWERLOC     varchar(30),
COMMENT         varchar(100),
TIP             varchar(100)
)
```

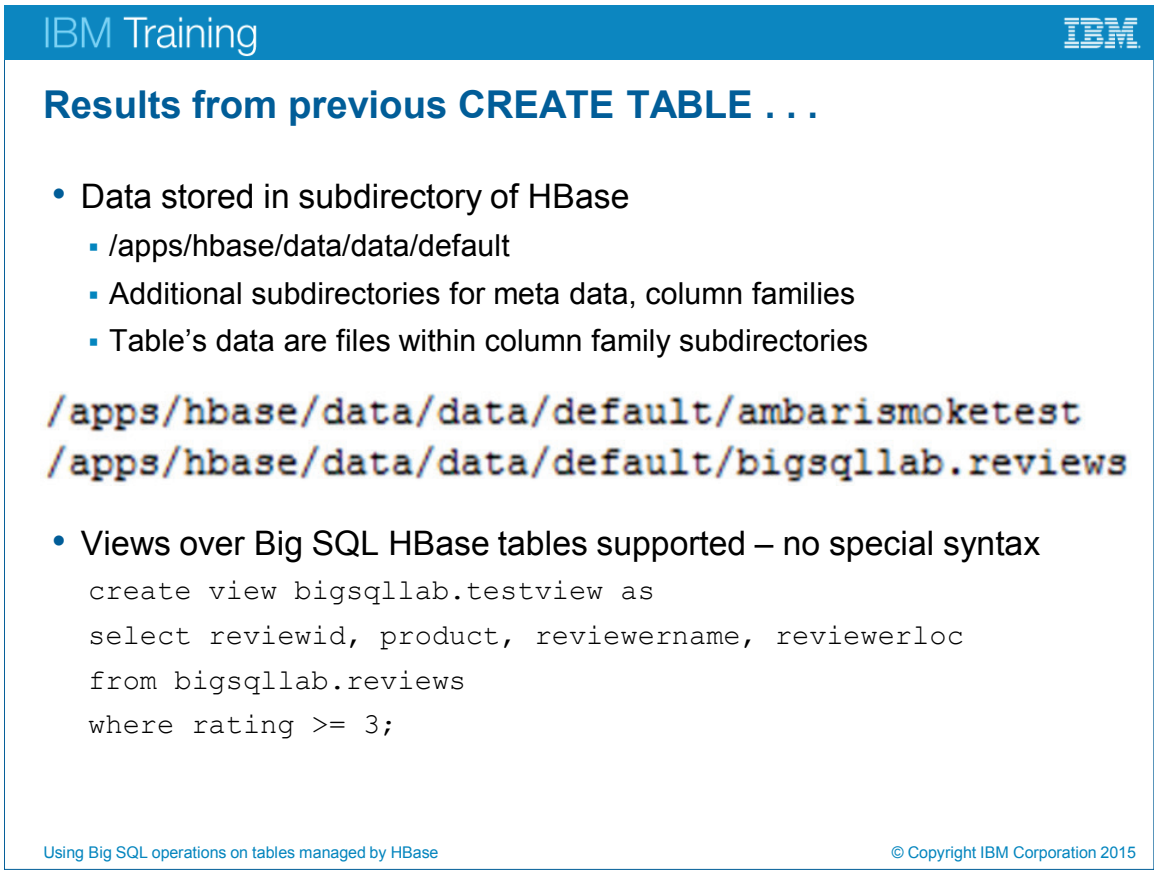
```
COLUMN MAPPING
(
key             mapped by (REVIEWID),
summary:product mapped by (PRODUCT),
summary:rating  mapped by (RATING),
reviewer:name   mapped by
(REVIEWERNAME),
reviewer:location  mapped by
(REVIEWERLOC),
details:comment    mapped by
(COMMENT),
details:tip     mapped by (TIP)
);
```


Creating a Big SQL table in HBase

Let's start with how to create a Big SQL table.

This statement creates a Big SQL table named REVIEWS in the BIGSQLLAB schema and instructs Big SQL to use HBase as the underlying storage manager. The COLUMN MAPPING clause specifies how SQL columns are to be mapped to HBase columns in column families. For example, the SQL REVIEWERNAME column is mapped to the HBase column family:column of 'reviewer:name'.

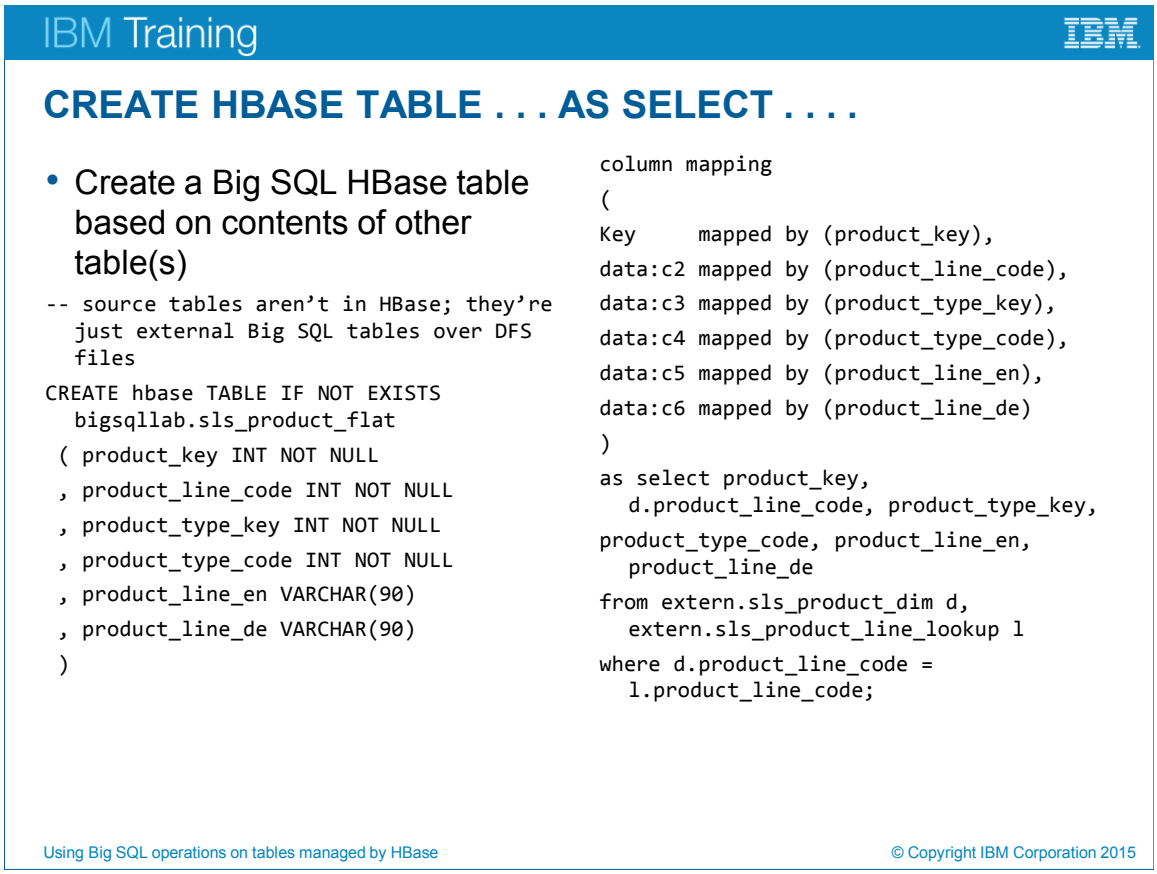
For simplicity, this example uses a 1:1 mapping of SQL columns to HBase columns. It also uses a single SQL column as the HBase row key. In some cases, you may want (or need) to map multiple SQL columns to one HBase column or to the HBase row key. Big SQL supports doing so. Also for simplicity, we've accepted various defaults for this table, including default HBase column family property settings (for VERSIONS and others) and binary encoding for storage. Finally, note that the SQL REVIEWID column was defined as the SQL primary key. This is an informational constraint that can be useful for query optimization. However, it isn't actively enforced.



Results from previous CREATE TABLE…

Executing the previous CREATE TABLE statement leaves us with a new subdirectory in the HBase data directory (by default, */hbase/data/default*). If you use basic Hadoop file system commands, you can list the contents of your HBase table's subdirectory structure and see that it contains further subdirectories for each of the column families you created for your table. Since our example specified 3 column families, we can see that 3 subdirectories for these were created within the appropriate Hbase subdirectory.

You can create Big SQL views just like you'd create a view in a relational DBMS.



CREATE HBASE TABLE … AS SELECT…

In addition to the simple CREATE HBASE TABLE statement shown earlier, you can also create and populate new Big SQL HBase tables based on the content of other Big SQL tables, including those not managed by HBase. When might you want to do this? Consider the example shown here. Since multi-way join processing isn't a strength of HBase, you may need to de-normalize a traditional relational database design to implement an efficient HBase design for your Big SQL tables. Of course, Big SQL doesn't mandate the use of HBase -- you can use Hive or simple DFS files -- but let's assume you concluded that you wanted to use HBase as the underlying storage manager.

Here we're mapping 2 tables along the PRODUCT dimension of the relational data warehouse into a single Big SQL table. Since the content for the source tables is stored in two separate files that each contain different sets of fields, we couldn't directly load them into our target table. Here, we decided to use Big SQL to help you with that task.

Specifically, we uploaded the source files into your DFS using standard Hadoop file system commands and created Big SQL externally managed tables over these files. Then we selected the data we wanted from these external tables and created a Big SQL HBase table based on that result set.

IBM Training

Populating Tables via LOAD

- Typically best runtime performance
- Load data from remote file system

```
LOAD HADOOP using file url
'sftp://myID:myPassword@myhost:22/sampleDir/bigsql/samples/data/GOSALES
  DW.SLS_PRODUCT_DIM.txt' with SOURCE PROPERTIES
  ('field.delimiter'='\t') INTO TABLE bigsqllab.sls_product_dim;
```

- Loads data from RDBMS (DB2, Netezza, Teradata, Oracle, MS-SQL, Informix) via JDBC connection

```
LOAD HADOOP USING JDBC CONNECTION URL 'jdbc:teradata://myhost/database=GOSALES'
WITH PARAMETERS (user ='myuser',password='mypassword')
FROM TABLE 'GOSALES'.'COUNTRY' COLUMNS (SALESCOUNTRYCODE, COUNTRY, CURRENCYNAME)
WHERE 'SALESCOUNTRYCODE > 9' SPLIT COLUMN 'SALESCOUNTRYCODE‘
INTO TABLE country_hbase
WITH TARGET TABLE PROPERTIES
('hbase.timestamp' = 1366221230799, 'hbase.load.method'='bulkload')
WITH LOAD PROPERTIES ('num.map.tasks' = 10, 'num.reduce.tasks'=5);
```


Populating Tables via LOAD

After you create a Big SQL table, you can use the LOAD command to populate it with data from files in a remote file system, files in your distributed file system, or data in the remote RDBMS servers listed. For RDBMS data, you specify JDBC URL properties and either enter a SQL query or a table name to identify the data to be retrieved. Behind the scenes, LOAD uses Sqoop connectors to retrieve the necessary data from the source.

In general, LOAD typically offers the best runtime performance for populating tables with data. Although not shown in this example, you can even set a LOAD property option to force a bulk load operation. Details are available in the product documentation.

However, beyond LOAD there are some other options you may want to be aware of . . . .

IBM Training

Populating Tables via INSERT

- INSERT INTO . . . SELECT FROM . . .

```
create hbase table bigsqllab.rev2 (
REVIEWID      varchar(10) primary key not null,
PRODUCT       varchar(30), RATING        int )
column mapping (
key           mapped by (REVIEWID),
cf1:product   mapped by (PRODUCT),
cf1:rating  mapped by (RATING) );

insert into bigsqllab.rev2 (reviewid, product, rating)
select reviewid, product, rating from bigsqllab.reviews;
```

- INSERT INTO . . . VALUES(. . .)

```
insert into bigsqllab.reviews
values ('198','scarf','2','Bruno',null,'Feels cheap',null);

insert into bigsqllab.reviews (reviewid, product, rating, reviewername)
values ('298','gloves','3','Beppe');
```

Using Big SQL operations on tables managed by HBase

Populating Tables via INSERT

In some cases, it may be more convenient to populate a table with data using an INSERT statement. The INSERT INTO . . . SELECT FROM statement supports parallel read and write operations and can be a convenient way to populate a table with data retrieved from a query, particularly if you want to change the underlying storage format used for data.

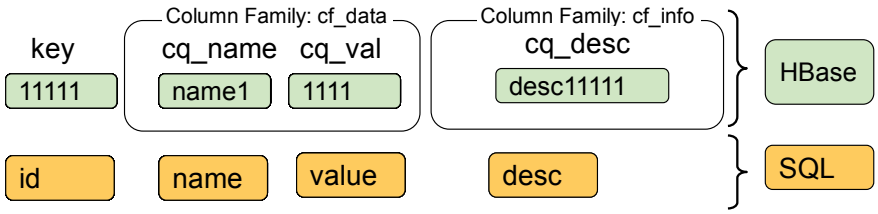
The traditional INSERT INTO . . . VALUES(…) format is also supported.

Some INSERT operations are can be processed in bulk based on various runtime factors, such as the number of rows to be inserted. See the product documentation for details on multi-row INSERT operations.

IBM Training

Column mapping

- Mapping HBase row key/columns to SQL columns
  - Supports one to one and one to many mappings
- One to one mapping
  - Single HBase entity mapped to a single SQL column

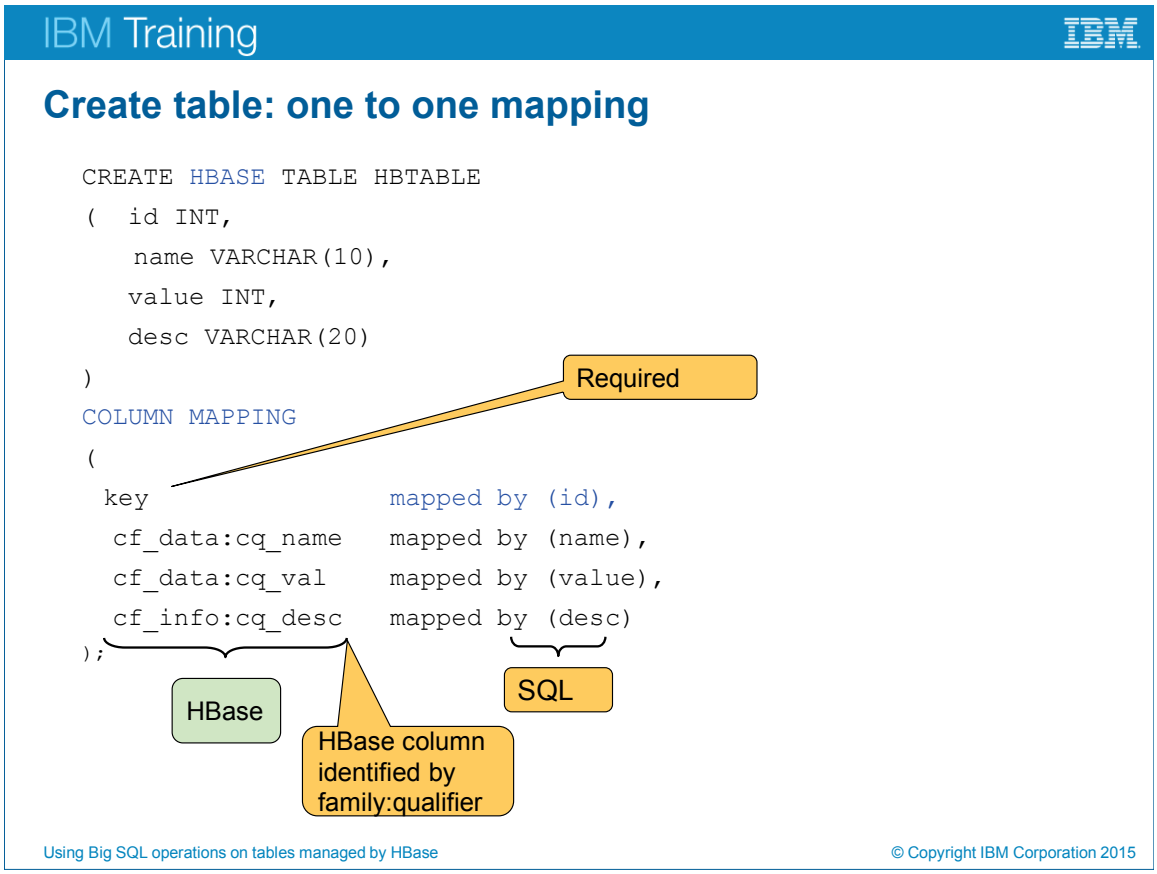


Using Big SQL operations on tables managed by HBase

Column mapping

Here is a closer look at how columns are mapped between HBase and Big SQL. Big SQL supports both one to one and one to many mappings.

In one to one mapping, the HBase row key and each HBase column are each mapped to a single Big SQL column. In the example, the HBase row key is mapped to the Big SQL column named *id*. The *cq_name* column in the column family *cf_data* is mapped to the Big SQL column named *name, and so on*.



Create table: one to one mapping

Here is the DDL statement to create a Big SQL table with the one-to-one mapping we saw in the previous slide.

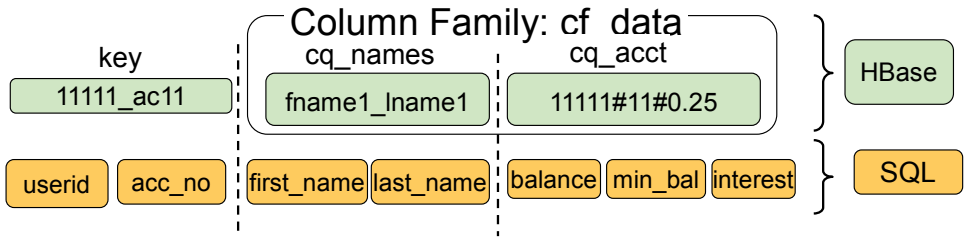
```
CREATE HBASE TABLE HBTABLE
( id INT,
name VARCHAR(10),
value INT,
desc VARCHAR(20)
)
COLUMN MAPPING
(
key mapped by (id),
cf_data:cq_name mapped by (name),
cf_data:cq_val mapped by (value),
cf_info:cq_desc mapped by (desc)
);
```

The column mapping section requires a mapping for the key. HBase columns are identified using family qualifier: column.

IBM Training

One to many column mapping

- **Single HBase entity** mapped to **multiple SQL columns**
- Composite key
  - One **HBase row key** mapped to **multiple SQL columns**
- Dense column
  - One **HBase column** mapped to **multiple SQL columns**



Using Big SQL operations on tables managed by HBase

One to many column mapping

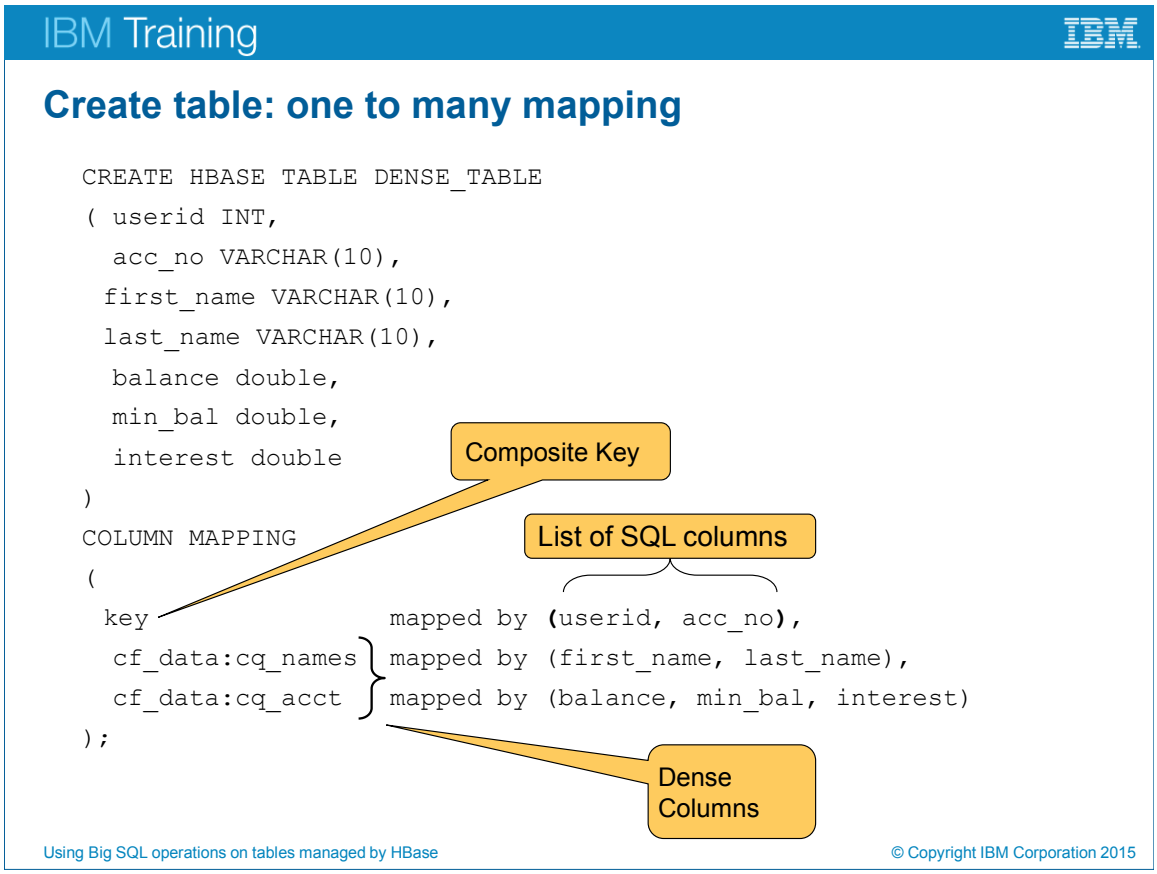
In one-to-many mapping, a single HBase entity (row key or a column) is mapped to multiple SQL columns.

There are two terms that need to be defined: composite key and dense column.

Composite key is an HBase row key that is mapped to multiple SQL columns.

Dense column is an HBase column that is mapped to multiple SQL columns.

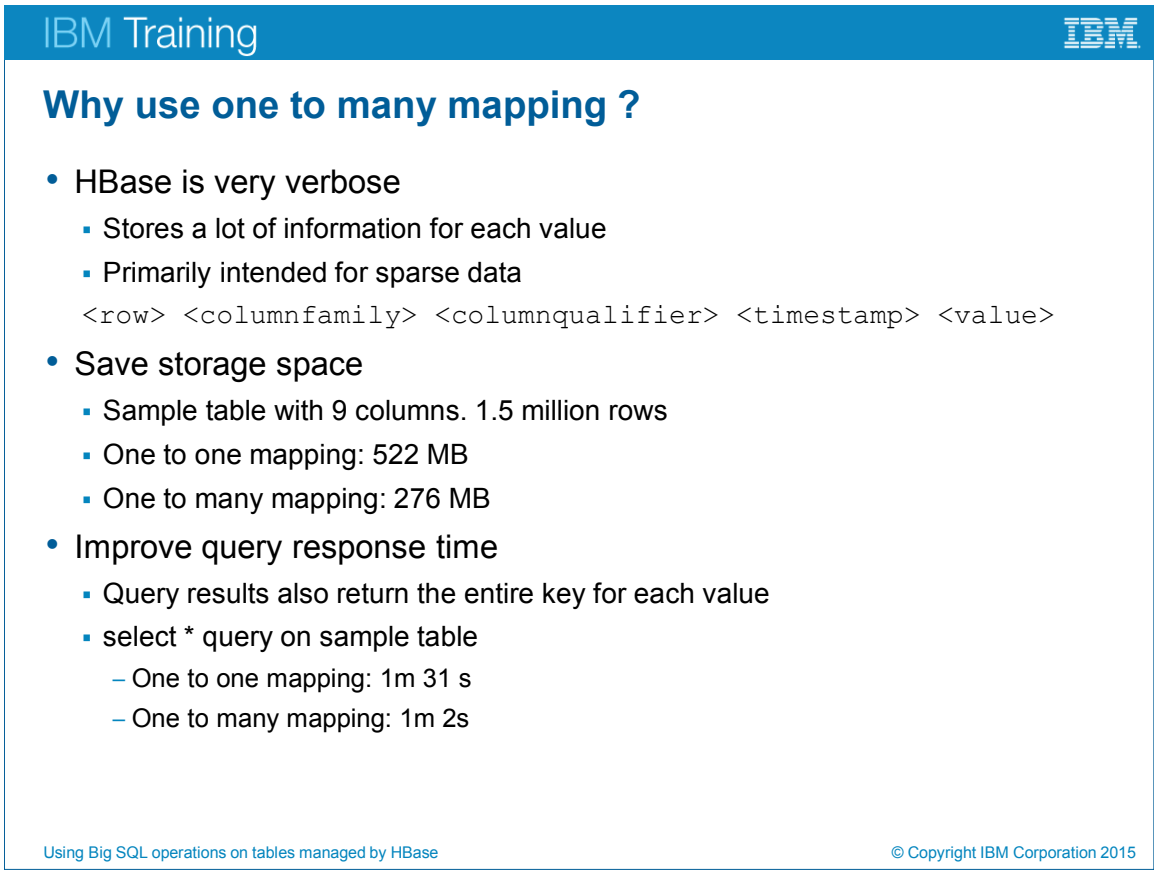
In the example, the key column is broken into two parts, userid and account number. Each part corresponds to a Big SQL column. Similarly, the HBase columns are mapped to multiple Big SQL columns. The DDL is shown on the next slide.



Create table: one to many mapping

This example shows all entities using a one-to-many mapping. Note there can be a mix. For example, there can be a composite key, a dense column and a non-dense column or any mix of these.

```
CREATE HBASE TABLE DENSE_TABLE
( userid INT,
acc_no VARCHAR(10),
first_name VARCHAR(10),
last_name VARCHAR(10),
balance double,
min_bal double,
interest double
)
COLUMN MAPPING
(
key mapped by (userid, acc_no),
cf_data:cq_names mapped by (first_name, last_name),
cf_data:cq_acct mapped by (balance, min_bal, interest)
);
```



Why use one to many mapping?

So why do you need one to many mappings?

HBase stores a lot of information for each value. Leaving out some details, you can think of each value stored along with a key, consisting of the row key, column family name, column qualifier, and timestamp.

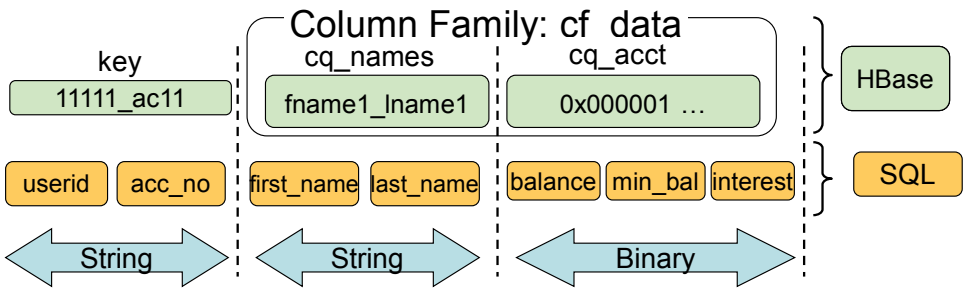
HBase is verbose and it is primarily intended for sparse data. In most cases, data in relational world is not sparse. If you were to store each Big SQL column individually, the required storage space would grow exponentially. Also, when a query executes, the query also returns the entire key for each value.

As an example, consider a table with 9 columns with 1.5 million rows. By using a one-to-one mapping, this table requires 522 MB versus a one-to-many mapping, which requires only 276 MB. The queries on the one-to-one mapped table are also slower.

IBM Training

Data encoding

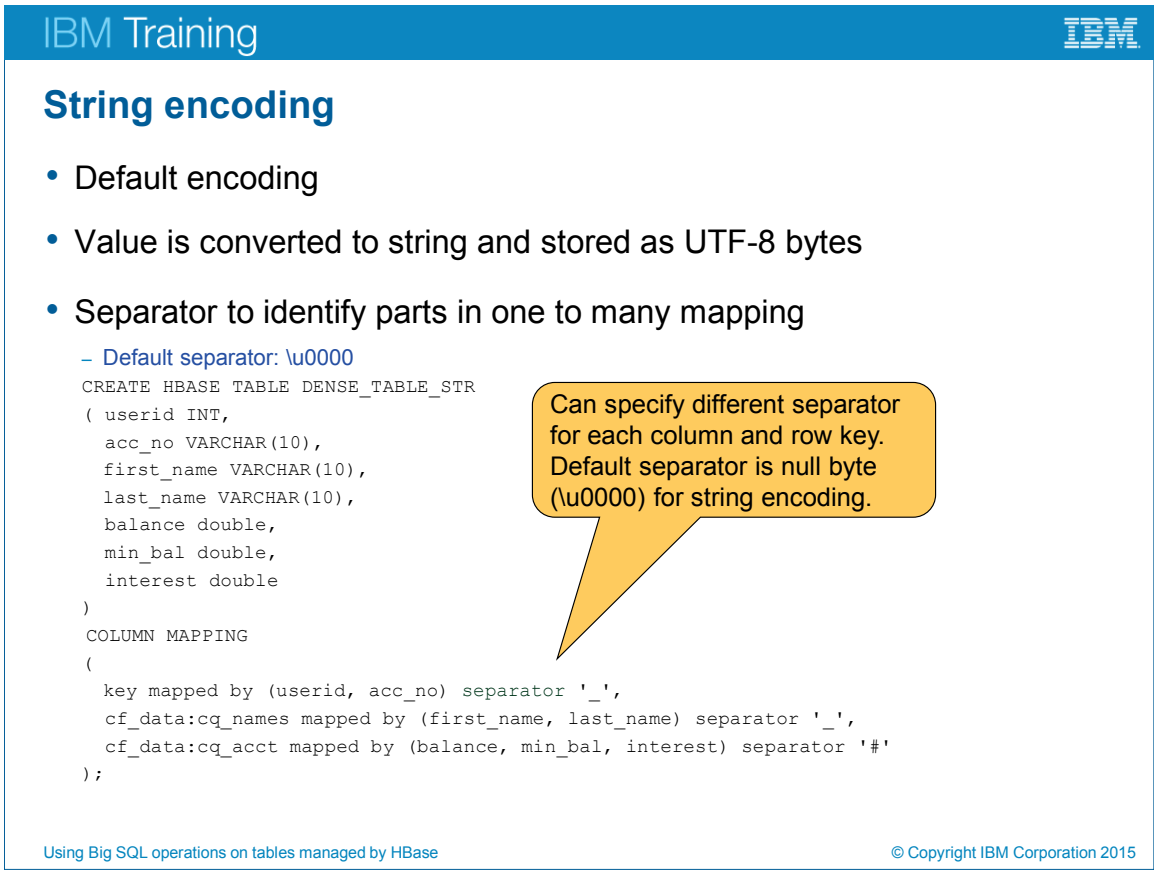
- HBase stores all data as an array of bytes
  - Application decides how to encode/decode the bytes
- Big SQL uses Hive SerDe interface for serialization/deserialization
- Supports two types of data encodings: String, Binary
- Encoding can be specified at HBase row key/column level



Data encoding

Big SQL supports two types of data encodings: string and binary.

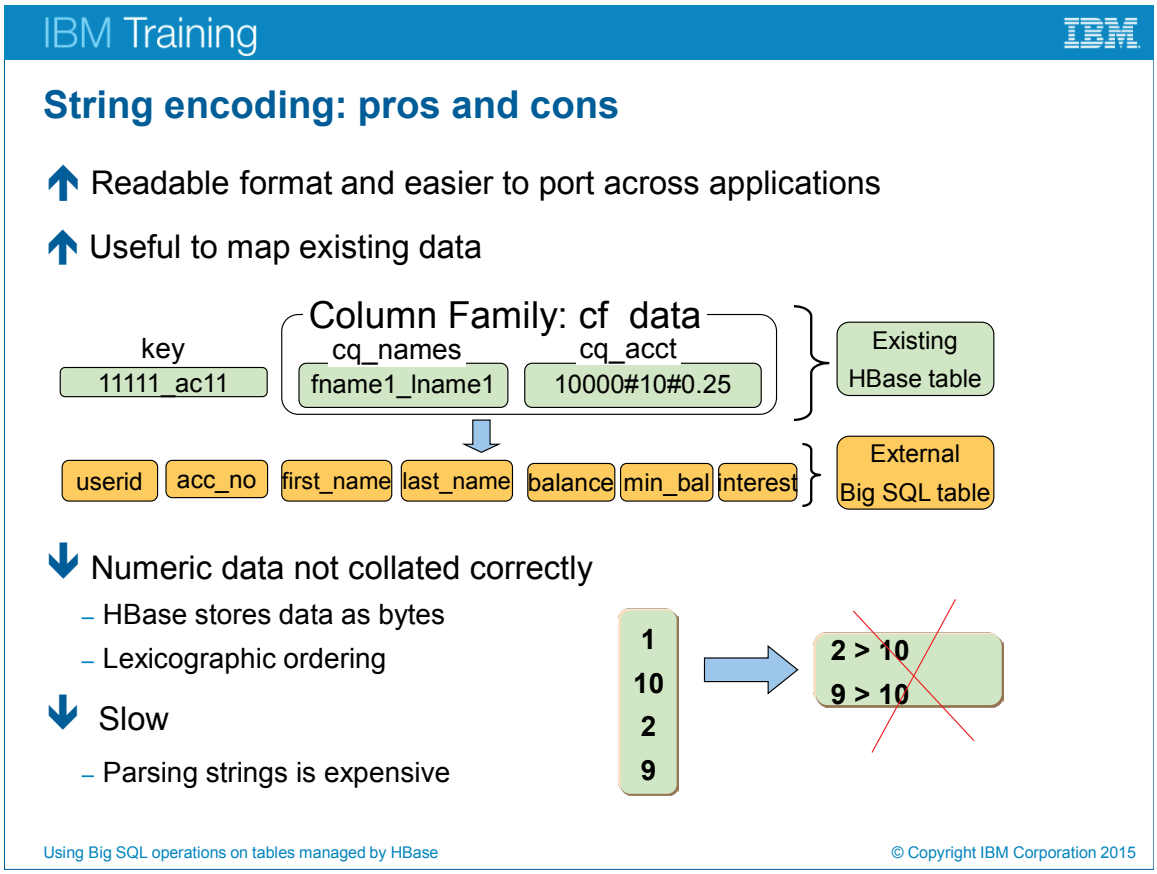
Each HBase entity can have its own encoding; that is the row key can be encoded as a string, one HBase column can be encoded as binary and another as string.



String encoding

String is the default encoding used in Big SQL HBase tables. The value is converted to string and stored as UTF-8 bytes. When multiple columns are packed into one HBase entity, separators are used to delimit data. The default separator is the null byte. As it is the lowest byte, it maintains data collation and let's range queries and partial row scans to work correctly.

The column mapping section in the create table statement allows specifying a different separator for each column and row key.



String encoding: pros and cons

There is no custom serialization/deserialization logic required for string encoding. This makes it portable in case someone wants to use another application to read data in HBase tables. A main use case for string encoding is when someone wants to map existing data. Delimited data is a common form of storing data and it can be easily mapped using Big SQL string encoding.

However, parsing strings is expensive and queries with data encoded as strings are slow. Also, numeric data is not collated correctly. HBase uses lexicographic ordering. So you might run into cases where a query returns wrong results.

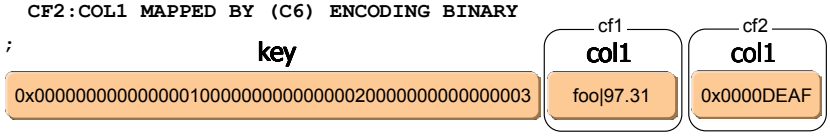
IBM Training

Binary encoding

- Data encoded using sortable binary representation
- Separators handled internally
  - Escaped to avoid issue of separator existing within data

```
CREATE HBASE TABLE MIXED_ENCODING
(
   C1   INT, C2   INT, C3   INT,
   C4   VARCHAR(10), C5   DECIMAL(5,2),
   C6   SMALLINT
)
COLUMN MAPPING
(
   KEY MAPPED BY (C1, C2, C3) ENCODING BINARY,
   CF1:COL1 MAPPED BY (C4, C5) SEPARATOR '|',
   CF2:COL1 MAPPED BY (C6) ENCODING BINARY
);
```

If encoding not specified, string is used as default



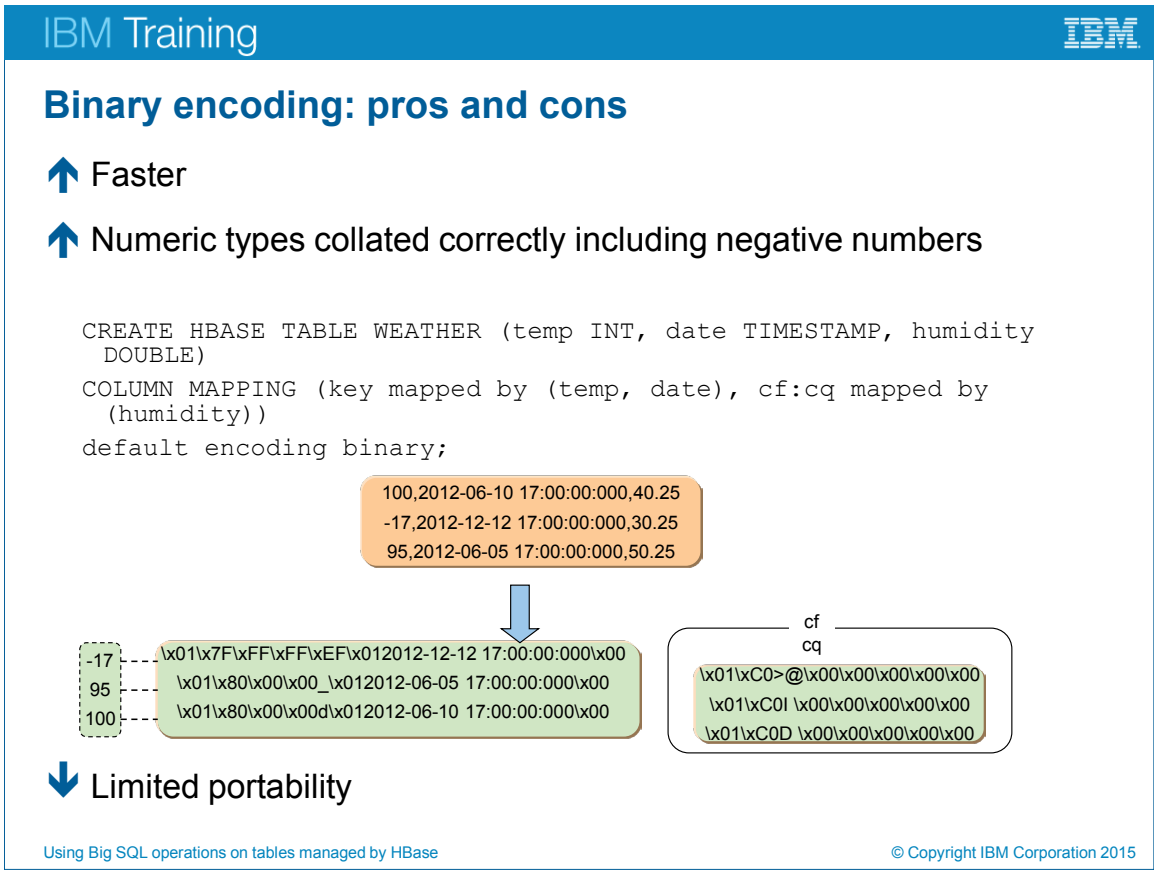
Using Big SQL operations on tables managed by HBase

Binary encoding

Binary encoding in Big SQL is sortable. So numeric data, including negative numbers, collates properly. It handles separators internally and avoids issues of separator existing within the data.

The following create table statement creates a table with mixed encodings. The row key and one of the columns are encoded as binary. For one of the columns, no encoding is specified which means it is encoded as string.

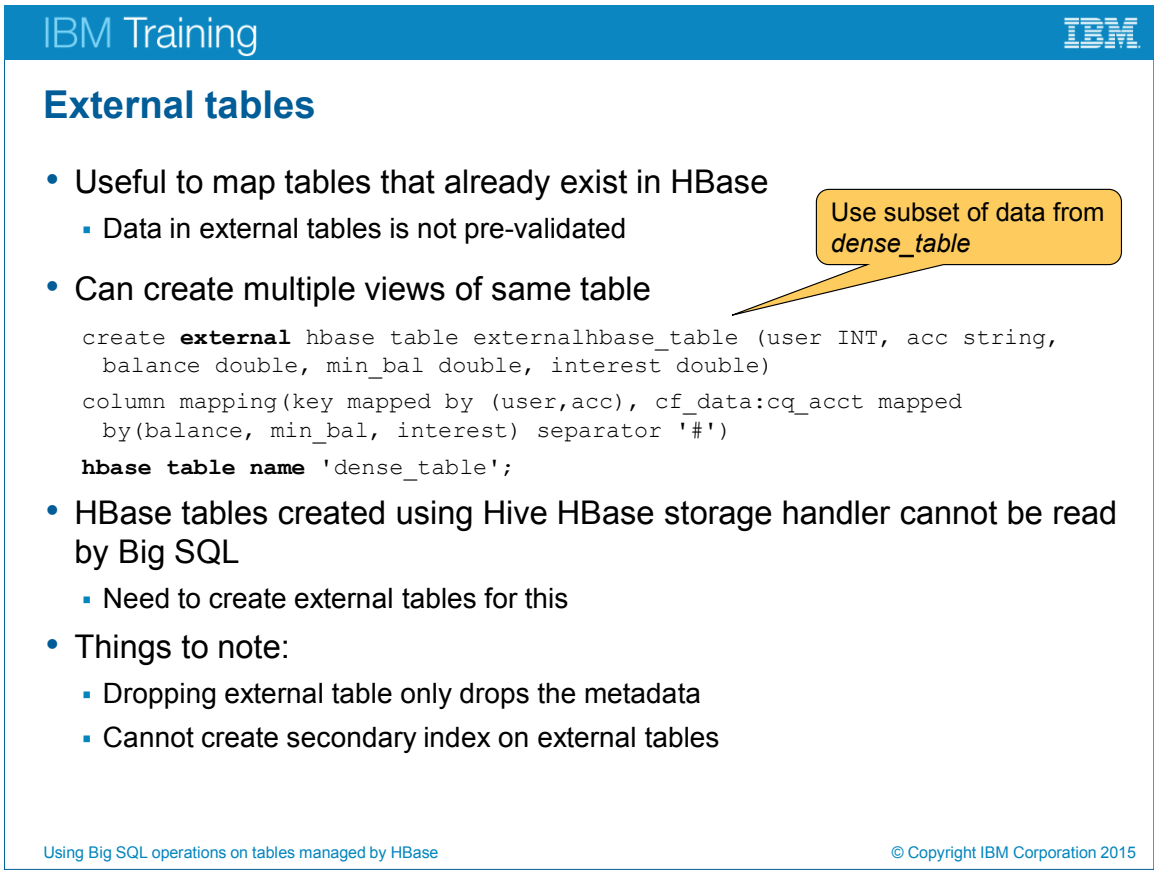
```
CREATE HBASE TABLE MIXED_ENCODING
(
C1   INT, C2   INT, C3   INT,
C4   VARCHAR(10), C5   DECIMAL(5,2),
C6   SMALLINT
)
COLUMN MAPPING
(
KEY MAPPED BY (C1, C2, C3) ENCODING BINARY,
CF1:COL1 MAPPED BY (C4, C5) SEPARATOR '|',
CF2:COL1 MAPPED BY (C6) ENCODING BINARY
);
```



Binary encoding: pros and cons

Queries on data encoded as binary have faster response times. As shown in the example, numeric data, including negative numbers, is collated correctly with binary encoding.

The downside is you get data encoded by Big SQL logic and might not be portable as it is.



External tables

Big SQL allows user to map existing data in HBase tables.

The data in external tables is not validated at creation time, that is, if a column in external table contains data with separators incorrectly defined, the query results would be unpredictable.

The create table statement allows for you to specify a different name for BIG SQL table through the *hbase* table name clause. Using external tables, you can also create multiple views of the same HBase table. For instance, one table can map too few columns and another table to another set of columns.

Another place where external tables can be used is to map tables, created using the Hive HBase storage handler. These cannot be directly read using a Big SQL storage handler.

Some things to note about external tables. They are not owned by Big SQL and hence cannot be dropped via Big SQL. Secondary indexes cannot be created via Big SQL on external tables.

IBM Training

Options to speed up load

- Disable write-ahead log (WAL)
  - Data loss can happen if region server crashes

```
LOAD HBASE DATA INPATH 'tpch/ORDERS.TXT' DELIMITED
FIELDS TERMINATED BY '|' INTO TABLE ORDERS DISABLE WAL;
```

- Increase write buffer
  - set hbase.client.write.buffer=8388608;

Options to speed up load

By using the option to disable WAL (write-ahead log), writes into HBase can be sped up. However, this is not really a safe option. Turning off WAL can result in data loss if a region server crashes.

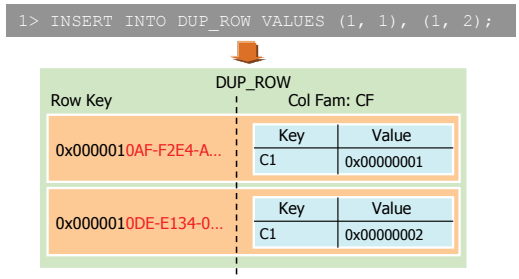
Another option to speed up a load is to increase the write buffer size.

IBM Training

Forcing unique row keys

- HBase requires unique row keys for each row
  - PUT (insert) with duplicate row key value creates new version of data - no error as with relational PK!
  - Modeling challenge for some RDBMS designs
- Big SQL's FORCE KEY UNIQUE allows for duplicate row keys
  - Big SQL transparently appends a "uniqifier" to the row key (36 bytes)

```
CREATE HBASE TABLE DUP_ROW
(
   C1 INT NOT NULL,
   C2 INT NOT NULL
)
COLUMN MAPPING
(
   KEY MAPPED BY (C1) FORCE KEY UNIQUE
   CF:C1 MAPPED BY (C2)
);
```



Using Big SQL operations on tables managed by HBase

Forcing unique row keys

As you know, HBase requires unique row keys for each row in an HBase table. Indeed, if you issue 3 HBase write operations with the same row key value, you won't end up with 3 different rows. You won't even get an error message. Instead, Hbase will simply version your data and treat only the most recent-written record as the current row for that row key. That may not be what you want (or expect).

So what are your options? A simple one involves using Big SQL's CREATE HBASE TABLE … FORCE KEY UNIQUE clause. As you can see in this example, you specify this clause as part of the row key mapping specification. This instructs Big SQL to append additional data to the key values to ensure that each input record results in a unique value (and therefore a new row in the HBase table), shown in the example at right. This additional data won't be visible to users who query the table.

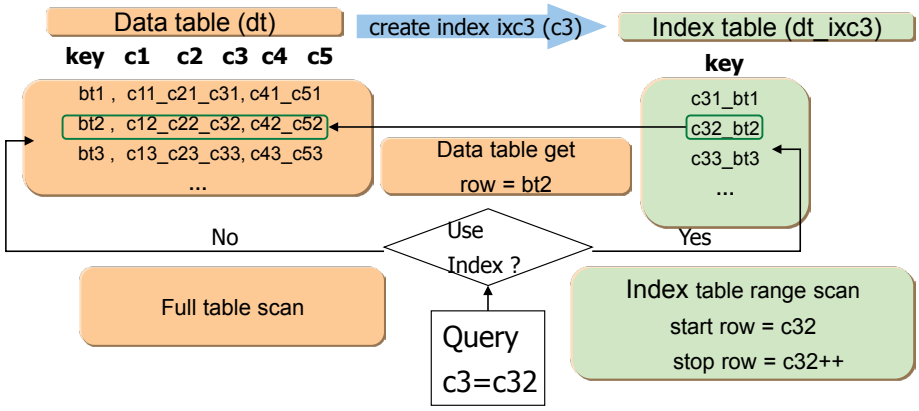
A downside to this approach is that additional storage space is consumed.

IBM Training

Secondary indexes (1 of 2)

- Defined on SQL column(s) not mapped to HBase row key
- Optimizer can leverage secondary indexes automatically
- Secondary indexes maintained automatically

```
create hbase table dt(id int,c1
string,c2 string,c3 string,c4
string,c5 string)
column mapping (key mapped by
(id), f:a mapped by (c1,c2,c3),
f:b mapped by (c4,c5));
create index ixc3 on dt (c3);
```



Secondary indexes (1 of 2)

Big SQL supports creation of secondary indexes which are self-maintaining. By self-maintaining, I refer to the fact that the index will get updated when the base data table is updated.

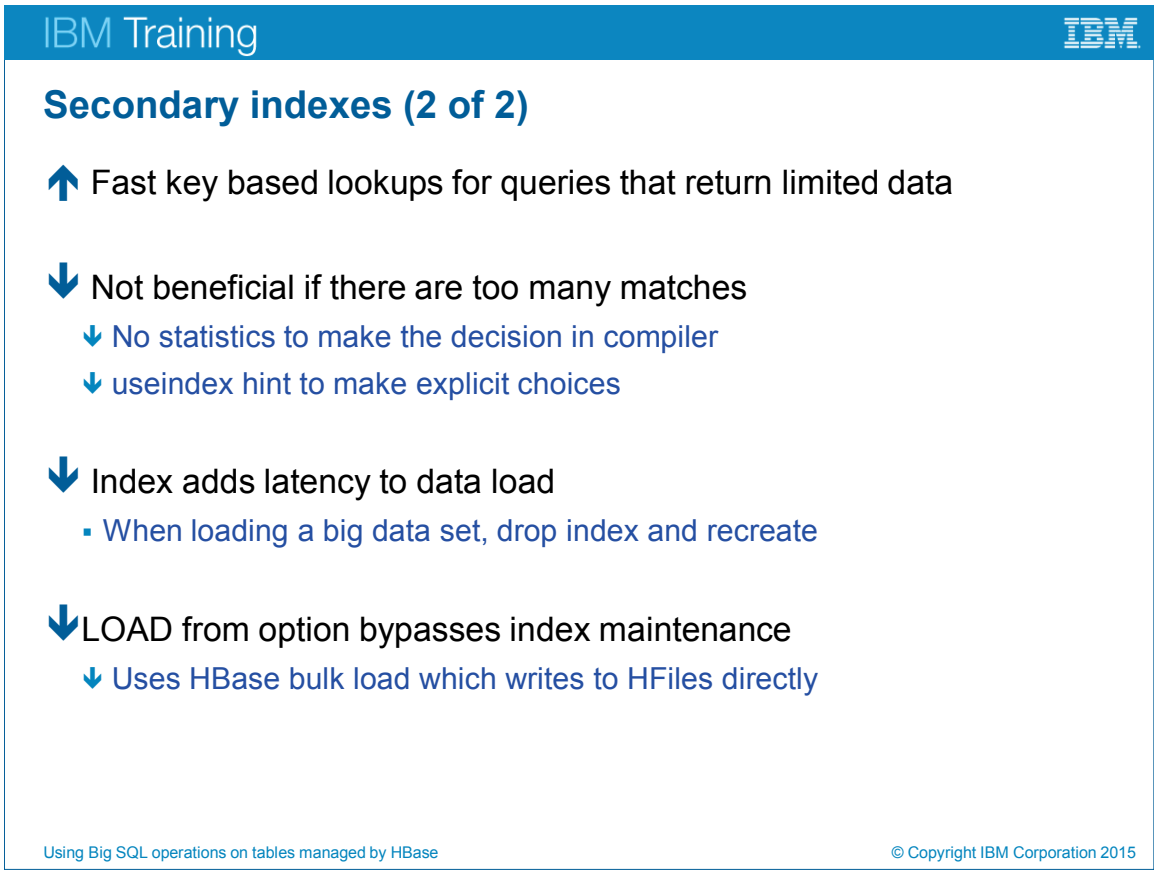
In HBase, the row key provides the same data retrieval benefits as a primary index. So, when you create a secondary index, use elements that are different from the row key.

Secondary indexes allow you to have a secondary way to read an HBase table. They provide a way to efficiently access records by means of some piece of information other than the primary key. Secondary indexes require additional cluster space and processing because the act of creating a secondary index requires both space and processing cycles to update.

The index is stored as another HBase table. At creation time, it is populated using a map reduce index builder. Once created, the index is maintained by a coprocessor. When new data is loaded into the base table, the index coprocessor synchronously updates the index table.

In the example, an index is created on column c3 which is part of a dense column in data table dt. This creates a new table to store index data. The index table stores the column value and row key it appears in. For simplicity, it is shown using separators. The internal representation is different.

When Big SQL gets a query with a predicate on c3, it will automatically use the index. In this case, the index table is scanned for all matching rows that start with value of predicate c3, in this case c32. From the matching row(s), the row key(s) of base table are extracted and get requests are batched and sent to data table. In this case, it is a single get request which can quickly get the matching row.



Secondary indexes (2 of 2)

If a query returns limited data, the use of an index converts a possible full table scan into two key lookups, a range scan on the index table and *Gets* on the base table.

Indexes can dramatically improve lookups on HBase tables when the index can be selective. However, accessing an index requires a lookup on the index table, followed by a lookup on the base table, which might not be co-located. This action can become costly when scanning many rows. In that case, it might be more efficient to do a MapReduce job to scan the base table rather than an index lookup.

Currently, there are no statistics for the index. Hence the decision of whether or not the index is useful cannot be made with accuracy. If an index is defined and there are predicates that match the index, the query assumes that it is beneficial to use the index. If the query ends up with a wide range of rows, it is up to the user to provide a *useindex='false'* hint. Similarly, when there are multiple indexes on the same column, the user can explicitly specify which index to use by employing this hint.

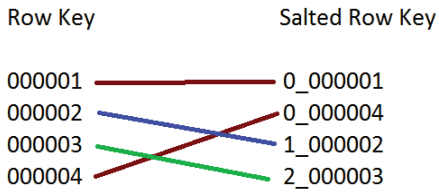
As the index maintenance is synchronous, it adds some overhead to data inserts. After the index is created, if a lot of data is going to be inserted into the data table, it is better to drop the index and recreate it. Index creation uses MapReduce and is faster.

Currently, there is no support to disable or repair indexes.

IBM Training

Salting

- Adds prefix to row key
- Helps minimize hot spots in HBase regions
  - Common occurrence with sequential row key values (writes often hit same Region Server, inhibiting parallelism)
- Logical example (prefix routes salted rows to different regions):



- CREATE HBASE TABLE . . . ADD SALT

```
CREATE HBASE TABLE mysalt1 ( rowkey INT, c0 INT)
COLUMN MAPPING ( KEY MAPPED BY (rowkey), f:q MAPPED BY (c0)
) ADD SALT;
```


Salting

Salting is technique used natively in HBase to minimize hot spots in regions, which often occur when writing sequential row key values to HBase tables over time. Such writes tend to hit the same Region Server. By “salting” or prefixing the row key values, you can help ensure that new rows are distributed across different regions (and Region Servers).

The final statement on this chart illustrates how you can create a Big SQL table that uses HBase salting, prefixing row key values with the output of a hash function. Although not shown here, you can also specify the number of salt buckets in which to put the prefixed row keys. By default, Big SQL uses the number of region servers that exist at the time of the CREATE TABLE as the number of buckets.

IBM Training

Accessmode hint

- Will run the query locally in Big SQL server
  - Useful to avoid map reduce overhead
- Very important for HBase point queries
  - This is not detected currently by compiler
  - Specify accessmode='local' hint when getting a limited set of data from HBase
- Specify at query level

```
select o_orderkey from orders /*+ accessmode='local' +*/ where
o_custkey=1 and o_orderkey=454791;
```

- Specify at session level
  - *set force local on*
  - set force commands override query level hints

Accessmode hint

The *accessmode* hint is important for HBase. It avoids MapReduce overhead. Combined with point queries, they ensure sub-second response time without being affected by the total data size.

There are multiple ways to specify the *accessmode* hint, as a query hint or at the session level.

Note

A session level hint take precedence. If *set force local off;* is run in a session, all subsequent queries always use MapReduce even if an explicit *accessmode='local'* hint is specified.

IBM Training

HBase hints

- rowcachesize (default=2000)
  - Used as scan cache setting
  - Also used to determine number of get requests to batch in index lookups
- colbatchsize (default=100)
- useindex ('false' to avoid index usage)

```
select o_orderkey from orders /*+ rowcachesize=10000 +*/ where
  o_custkey>5000
go -m discard
1450136 rows in results(first row: 22.67s; total: 27.46s)
```

Log `HBase scan details:{... , caching=10000, ...}`

- rowcachesize can also be set using the set command:
  - set hbase.client.scanner.caching=10000;

Using Big SQL operations on tables managed by HBase

HBase hints

Big SQL queries can provide specific hints like the row and column batch sizes that are to be used when fetching the data for a particular table. So if you have a query that is narrow in scope, you can tell it to get a large batch of rows, allowing for a more efficient execution.

rowcachesize can also be set with the set command:

set hbase.client.scanner.caching=10000

IBM Training

Checkpoint

1. What are the differences between one to one mapping and one to many mapping?
2. Upsert is updating a row if it exists, otherwise insert the value. What can be a potential confusion with this?
3. What is one advantage of using index? What is a disadvantage?

Checkpoint

IBM Training

Checkpoint solutions

1. What are the differences between one to one mapping and one to many mapping?
   - One to one mapping maps each HBase entity to a single column.
   - One to many mapping maps each HBase entity to multiple columns.
2. Upsert is updating a row if it exists, otherwise insert the value. What can be a potential confusion with this?
   - Multiple rows with the same key only shows up as a single row.
3. What is one advantage of using index? What is a disadvantage?
   - Quick, key-based lookup for queries with limited data
   - Not beneficial if there are too many matches.

Checkpoint solutions

IBM Training

Demonstration 1

Working with data in HBase tables using Big SQL

At the end of this demonstration, you will be able to:

- Create an HBase table using Big SQL
- Query an HBase table using Big SQL

Demonstration 1: Working with data in HBase tables using Big SQL

Demonstration 1: Working with data in HBase tables using Big SQL

Purpose:
In this demonstration, you will learn the basics of using HBase natively and with Big SQL, IBM's industry-standard SQL interface for data stored in its Hadoop-based platform. HBase is an open source key-value data storage mechanism commonly used in Hadoop environments. It features a column-oriented data model that enables programmers to efficiently retrieve data by key values from very large data sets. It also supports certain data modification operations, readily accommodates sparse data, and supports various kinds of data. Indeed, HBase doesn't have primitive data types – all user data is stored as byte arrays. With BigInsights, programmers can use SQL to query data in Big SQL tables managed by HBase.

Although HBase provides useful data storage and retrieval capabilities, it lacks a rich query language. Fortunately, IBM's Big SQL technology, a component of BigInsights, enables programmers to use industry-standard SQL to create, populate, and query Big SQL tables managed by HBase. Native HBase data retrieval operations (such as GET and SCAN) can be performed on these tables, too. This lab introduces you to the basics of using Big SQL with HBase

Estimated time: 60 minutes

User/Password: **biadmin/biadmin**
root/dalvm3
db2inst1/ibm2blue

Services Password: **ibm2blue**

Task 1. Ensure BigInsights is running.

You may skip this task if your environment is still up and running. Refer to past demonstrations if you need to do any of these.

1. Ensure BigInsights is running via Ambari.
2. Ensure you can access the BigInsights Home page
3. Ensure you can access the Big SQL page
4. For this demonstration, make sure HBase is running as well. If not, start it up via the Ambari console.

Task 2. Using Big SQL to create and query HBase tables.

To get started, you will create a Big SQL table named `reviews` to capture information about product reviews. A script is provided with the statements you will need to run:

The 1_Big SQL HBase statements.sql file is located in the */home/biadmin/labfiles/bigsql/Big_SQL_HBase* folder.

1. Use either JSqsh or the Big SQL console to create the table. You will use the **bigsql** user id and **ibm2blue** password. Create the table with this statement.

```
CREATE HBASE TABLE IF NOT EXISTS BIGSQLLAB.REVIEWS (
REVIEWID        varchar(10) primary key not null,
PRODUCT     varchar(30),
RATING      int,
REVIEWERNAME    varchar(30),
REVIEWERLOC     varchar(30),
COMMENT     varchar(100),
TIP         varchar(100)
)
COLUMN MAPPING
(
key     mapped by (REVIEWID),
summary:product mapped by (PRODUCT),
summary:rating  mapped by (RATING),
reviewer:name   mapped by (REVIEWERNAME),
reviewer:location  mapped by (REVIEWERLOC),
details:comment    mapped by (COMMENT),
details:tip    mapped by (TIP)
);
```

   This statement creates a Big SQL table named REVIEWS in the default BIGSQL schema and instructs Big SQL to use HBase as the underlying storage manager. The COLUMN MAPPING clause specifies how SQL columns are to be mapped to HBase columns in column families. For example, the SQL REVIEWERNAME column is mapped to the HBase column family:column of 'reviewer:name'.

For simplicity, this example uses a 1:1 mapping of SQL columns to HBase columns. It also uses a single SQL column as the HBase row key. As you'll see in a subsequent lab, you may want (or need) to map multiple SQL columns to one HBase column or to the HBase row key. Big SQL supports doing so.

Also for simplicity, you have accepted various defaults for this table, including default HBase column family property settings (for VERSIONS and others) and binary encoding for storage. Finally, note that the SQL REVIEWID column was defined as the SQL primary key. This is an informational constraint that can be useful for query optimization. However, it isn't actively enforced.

2. Verify that the table was created in the Hadoop file system. If necessary, open up a new terminal window and list the contents of the HBase data directory and confirm that the *bigsql.reviews* subdirectory exists.

```
hdfs dfs -ls /apps/hbase/data/data/default
```

The root HBase directory is determined at installation. The examples in this lab were developed for an environment in which HBase was configured to store data in /apps/hbase/data/data/default. If your HBase configuration is different, adjust the commands as needed to match your environment.

3. List the contents of your table's subdirectory. Observe that this subdirectory contains another subdirectory with a system-generated name.

```
hdfs dfs -ls
/apps/hbase/data/data/default/bigsql.reviews
```

```
[biadmin@ibmclass Desktop]$ hdfs dfs -ls /apps/hbase/data/data/default/bigsql.reviews
Found 3 items
drwxr-xr-x   - hbase hdfs          0 2015-08-14 11:33 /apps/hbase/data/data/default/bigsql.reviews/.tabledesc
drwxr-xr-x   - hbase hdfs          0 2015-08-14 11:33 /apps/hbase/data/data/default/bigsql.reviews/.tmp
drwxr-xr-x   - hbase hdfs          0 2015-08-14 11:33 /apps/hbase/data/data/default/bigsql.reviews/42a3277d4abf48ae2c9d37cec2551342
```

4. List the contents of the . . . /bigsql.reviews subdirectory with the system-generated name. (In the example above, this is the . . . / 42a3277d4abf48ae2c9d37cec2551342 subdirectory.) Adjust the path specification below to match your environment.

```
hdfs dfs -ls
/apps/hbase/data/data/default/bigsql.reviews/42a3277d4ab
f48ae2c9d37cec2551342
```

```
/apps/hbase/data/data/default/bigsql.reviews/42a3277d4abf48ae2c9d37cec2551342/.regioninfo
/apps/hbase/data/data/default/bigsql.reviews/42a3277d4abf48ae2c9d37cec2551342/details
/apps/hbase/data/data/default/bigsql.reviews/42a3277d4abf48ae2c9d37cec2551342/reviewer
/apps/hbase/data/data/default/bigsql.reviews/42a3277d4abf48ae2c9d37cec2551342/summary
```

Note that there are additional subdirectories for each of the 3 column families specified in the COLUMN MAPPING clause of your CREATE HBASE TABLE statement.

5. Back to your Big SQL execution environment. Insert a row into your Big SQL reviews table. Note that we are using the default bigsql schema here.

```
insert into reviews values
('198','scarf','2','Bruno',null,'Feels cheap',null);
```

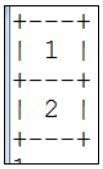
Note that this INSERT statement looks the same as any other SQL statement. You didn't need to insert one HBase cell value at a time, and you didn't need to understand the underlying HBase table structure.

6. Insert another row into your table, specifying values for only a subset of its columns.

```
insert into reviews (reviewid, product, rating,
reviewername) values ('298','gloves','3','Beppe');
```

7. Use SQL to count the number of rows stored in your table, verifying that 2 are present.

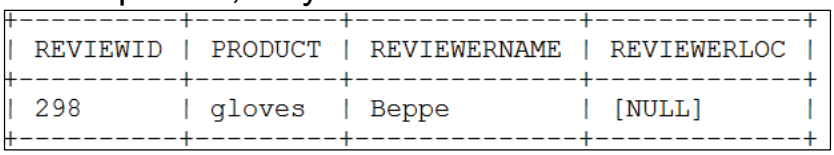
```
select count(*) from reviews;
```



8. Execute a simple query to return specific information about reviews of products rated 3 or higher.

```
select reviewid, product, reviewername, reviewerloc
from reviews
where rating >= 3;
```

As expected, only 1 row is returned.



Again, note that your SELECT statement looks like any other SQL SELECT - you didn't need to add any special code because the underlying storage manager is HBase.

9. Verify that you can work directly with the *bigsql.reviews* table from HBase. Launch the HBase shell. From the HBase home directory (such as `/usr/iop/current/hbase-client/bin`), issue this command:

```
hbase shell
```

10. Ignore any informational messages that may appear. Verify that the shell launched successfully and that your screen appears similar to this:

```
[biadmin@ibmclass bin]$ ./hbase shell
2015-08-14 11:49:33,678 INFO  [main] Configuration.deprecation: hadoop.native.lib is deprecated. Instead, use io.native.lib.available
HBase Shell; enter 'help<RETURN>' for list of supported commands.
Type "exit<RETURN>" to leave the HBase Shell
Version 0.98.8_IBM_4-hadoop2, rUnknown, Fri Mar 27 21:53:57 PDT 2015

hbase(main):001:0>
```

11. Scan the table:

```
scan 'bigsql.reviews'
```

```
hbase(main):001:0> scan 'bigsql.reviews'
SLF4J: Class path contains multiple SLF4J bindings.
SLF4J: Found binding in [jar:file:/usr/iop/4.0.0.0/hadoop/lib/slf4j-log4j12-1.7.5.jar!/org/slf4j/impl/StaticLoggerBinder.class]
SLF4J: Found binding in [jar:file:/usr/iop/4.0.0.0/zookeeper/lib/slf4j-log4j12-1.6.1.jar!/org/slf4j/impl/StaticLoggerBinder.class]
SLF4J: See http://www.slf4j.org/codes.html#multiple_bindings for an explanation.
ROW                                   COLUMN+CELL
 \x00198\x00                          column=details:comment, timestamp=1439570623068, value=\x00Feels cheap\x00
 \x00198\x00                          column=details:tip, timestamp=1439570623068, value=\x01
 \x00198\x00                          column=reviewer:location, timestamp=1439570623068, value=\x01
 \x00198\x00                          column=reviewer:name, timestamp=1439570623068, value=\x00Bruno\x00
 \x00198\x00                          column=summary:product, timestamp=1439570623068, value=\x00scarf\x00
 \x00198\x00                          column=summary:rating, timestamp=1439570623068, value=\x00\x80\x00\x00\x02
 \x00298\x00                          column=details:comment, timestamp=1439570671430, value=\x01
 \x00298\x00                          column=details:tip, timestamp=1439570671430, value=\x01
 \x00298\x00                          column=reviewer:location, timestamp=1439570671430, value=\x01
 \x00298\x00                          column=reviewer:name, timestamp=1439570671430, value=\x00Beppe\x00
 \x00298\x00                          column=summary:product, timestamp=1439570671430, value=\x00gloves\x00
 \x00298\x00                          column=summary:rating, timestamp=1439570671430, value=\x00\x80\x00\x00\x03
2 row(s) in 0.3280 seconds
```

As you would expect, the final line of output reports that there are 2 rows in your table. In case you're curious, \x00 is both the non-null marker and also the terminator used for variable length binary encoded values.

Task 3. Creating views over Big SQL HBase tables.

You can create views over Big SQL tables stored in HBase just as you can create views over Big SQL tables that store data in the Hive warehouse or in simple DFS files. Creating views over Big SQL HBase tables is straightforward, as you'll see in this task.

1. From your Big SQL query execution environment, create a view based on a subset of the `reviews` table that you created earlier.

```
create view testview as
select reviewid, product, reviewername, reviewerloc
from reviews
where rating >= 3;
```

Note that this view definition looks like any other SQL view definition. You didn't need to specify any syntax unique to Hadoop or HBase.

2. Query the view.

```
select reviewid, product, reviewername from testview;
```

```
+----------+---------+--------------+
| REVIEWID | PRODUCT | REVIEWERNAME |
+----------+---------+--------------+
| 298      | gloves  | Beppe        |
+----------+---------+--------------+
1 row in results(first row: 0.50s; total: 0.50s)
```

Task 4. Loading data into a Big SQL HBase table.

In this task, you'll explore how to use the Big SQL LOAD command to load data from a file into a Big SQL table managed by HBase. To do so, you will use sample data shipped with Big SQL that is typically installed with Big SQL client software. By default, this data is installed at /usr/ibmpacks/bigsql/4.0/bigsql/samples/data.
The sample data represents data exported from a data warehouse that tracks sales of outdoor products. It includes a series of FACT and DIMENSION tables. In this task, you will create 1 DIMENSION table and load sample data from 1 file into it.

1. Create a Big SQL table in HBase named sls_product_dim.

```
-- product dimension table
CREATE HBASE TABLE IF NOT EXISTS sls_product_dim
 ( product_key INT PRIMARY KEY NOT NULL
 , product_line_code INT NOT NULL
 , product_type_key INT NOT NULL
 , product_type_code INT NOT NULL
 , product_number INT NOT NULL
 , base_product_key INT NOT NULL
 , base_product_number INT NOT NULL
 , product_color_code INT
 , product_size_code INT
 , product_brand_key INT NOT NULL
 , product_brand_code INT NOT NULL
 , product_image VARCHAR(60)
 , introduction_date TIMESTAMP
 , discontinued_date TIMESTAMP
 )
COLUMN MAPPING
(
key                    mapped by (PRODUCT_KEY),
data:line_code         mapped by (PRODUCT_LINE_CODE),
data:type_key          mapped by (PRODUCT_TYPE_KEY),
data:type_code         mapped by (PRODUCT_TYPE_CODE),
data:number            mapped by (PRODUCT_NUMBER),
```

```
    data:base_key          mapped by (BASE_PRODUCT_KEY),
    data:base_number       mapped by (BASE_PRODUCT_NUMBER),
    data:color             mapped by (PRODUCT_COLOR_CODE),
    data:size              mapped by (PRODUCT_SIZE_CODE),
    data:brand_key         mapped by (PRODUCT_BRAND_KEY),
    data:brand_code        mapped by (PRODUCT_BRAND_CODE),
    data:image             mapped by (PRODUCT_IMAGE),
    data:intro_date        mapped by (INTRODUCTION_DATE),
    data:discon_date       mapped by (DISCONTINUED_DATE)
);
```

The HBase specification of this Big SQL statement placed nearly all SQL columns into a single HBase column family named 'data'. As you know, HBase creates physical files for each column family; this is something you should take into consideration when designing your table's structure. It's often best to keep the number of column families per table small unless your workload involves many queries over mutually exclusive columns. For example, if you knew that PRODUCT_IMAGE data was rarely queried or frequently queried alone, you might want to store it separately (i.e., in a different column family:column).

There's something else worth noting about this table definition: the HBase columns have shorter names than the SQL columns. HBase stores full key information (row key, column family name, column name, and timestamp) with the key value. This consumes disk space. If you keep the HBase column family and column names short and specify longer, more meaningful names for the SQL columns, your design will minimize disk consumption and remain friendly to SQL programmers.

Load data into the table.

2. This statement will return a warning message providing details on the number of rows loaded, etc.

```
load hadoop using file url
'file:///usr/ibmpacks/bigsql/4.0/bigsql/samples/data/GOS
ALESDW.SLS_PRODUCT_DIM.txt' with SOURCE PROPERTIES
('field.delimiter'='\t') INTO TABLE SLS_PRODUCT_DIM
overwrite;
```

3. Count the number of rows in the table to verify that 274 are present.

```
-- total rows in SLS_PRODUCT_DIM = 274
 select count(*) from SLS_PRODUCT_DIM;
```

4. Optionally, query the table.

```
select product_key, introduction_date,
product_color_code
from sls_product_dim
where product_key > 30000
fetch first 5 rows only;
```

```
+-------------+-------------------------+--------------------+
| PRODUCT_KEY | INTRODUCTION_DATE       | PRODUCT_COLOR_CODE |
+-------------+-------------------------+--------------------+
|       30001 | 1995-02-15 00:00:00.000 |                908 |
|       30002 | 1995-02-15 00:00:00.000 |                906 |
|       30003 | 1995-02-15 00:00:00.000 |                924 |
|       30004 | 1995-02-15 00:00:00.000 |                923 |
|       30005 | 1995-02-15 00:00:00.000 |                923 |
+-------------+-------------------------+--------------------+
5 rows in results(first row: 0.69s; total: 0.69s)
```

Task 5. Dropping tables created in this lab.

1. Drop the reviews table and the sls_product_dim table.

```
drop table reviews;
drop table sls_product_dim;
```

2. Verify that these tables no longer exist. For example, query each table and confirm that you receive an error message indicating that the table name you provided is undefined (SQLCODE -204, SQLSTATE 42704).

```
select count(*) from reviews;
```

Results:
In this demonstration, you learned the basics of using HBase natively and with Big SQL, IBM's industry-standard SQL interface for data stored in its Hadoop-based platform.

This demonstration introduced you to the basics of using Big SQL with HBase.

IBM Training

Unit summary

- Describe the basic function of HBase
- Issuing basic HBase commands
- Using Big SQL to create and query HBase tables
- Mapping HBase columns to Big SQL

Unit summary

IBM Training

Using Spark operations on tables managed by Big SQL

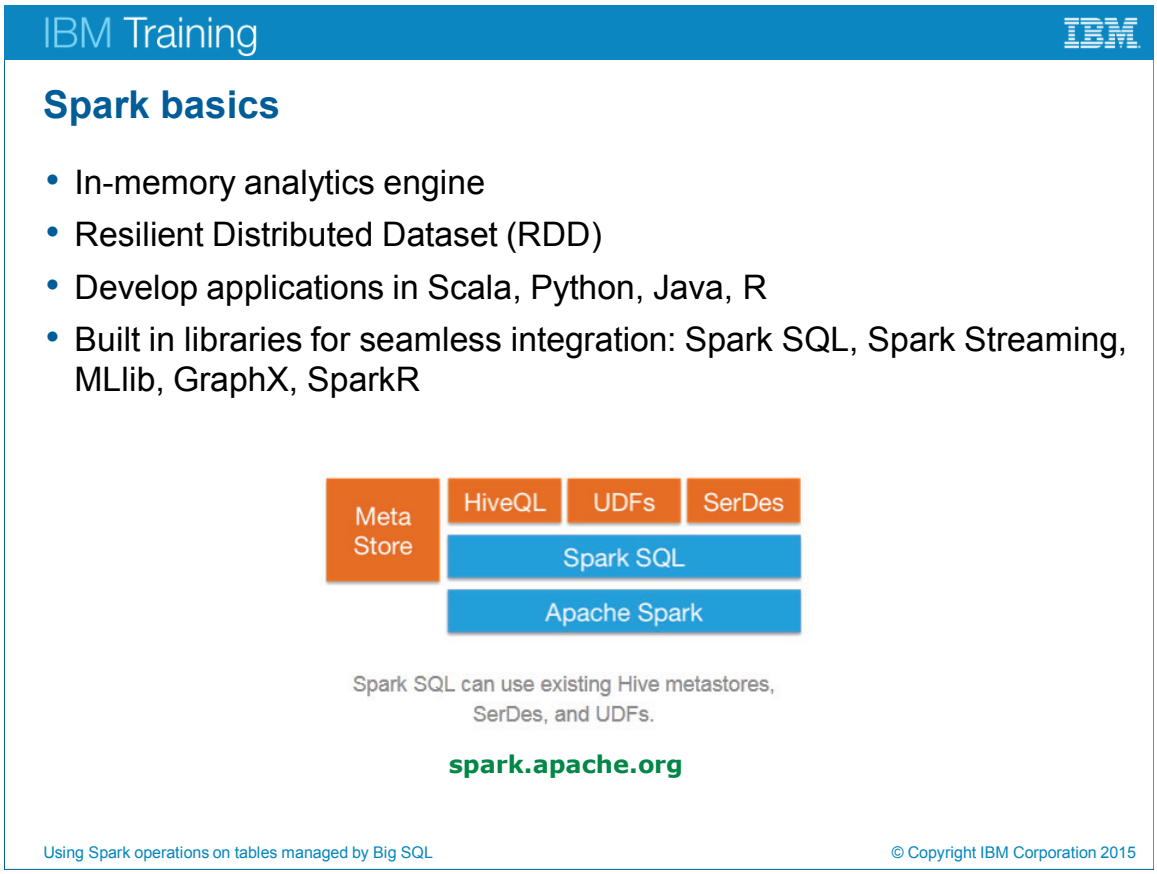
IBM BigInsights v4.0

IBM Training

Unit objectives

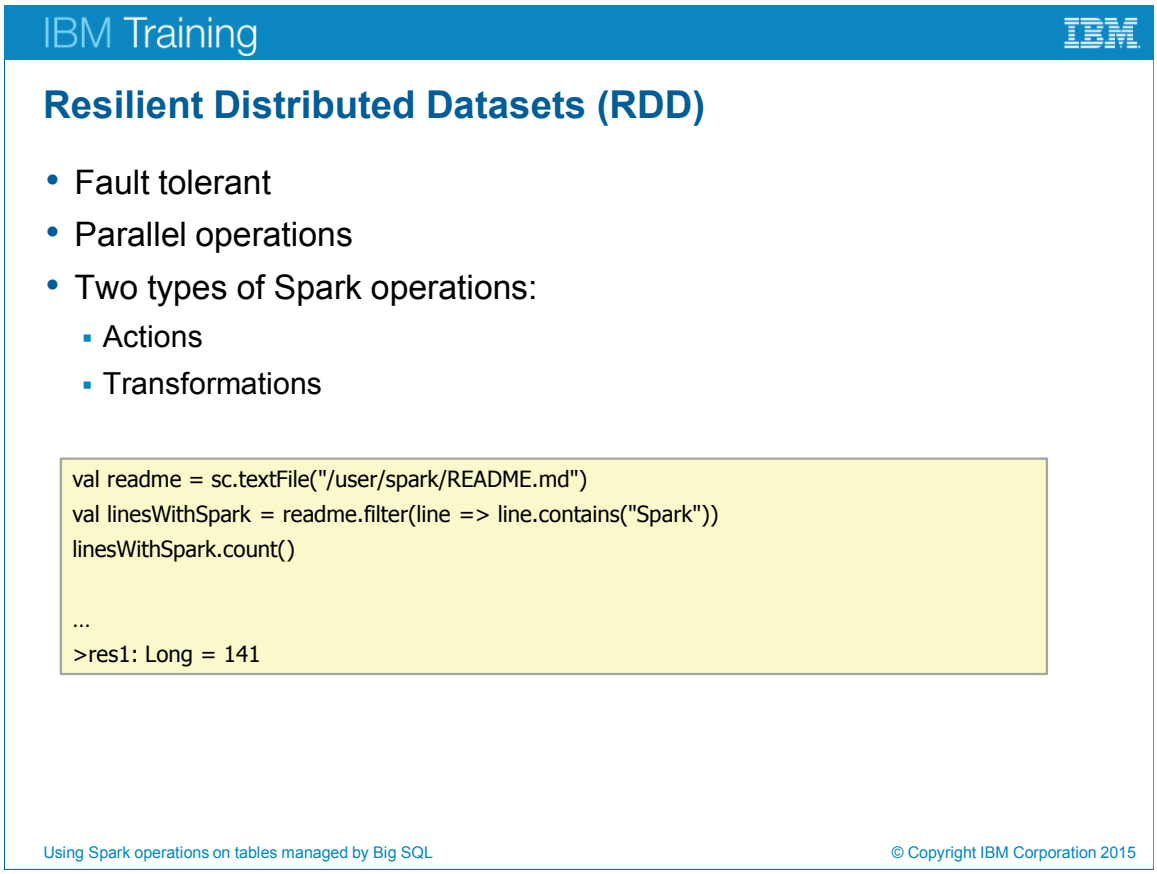
- Describe the purpose and role of Spark
- Querying and manipulating Big SQL data through Spark

Unit objectives



Spark basics

This unit assumes you know and understand how to use Spark, but I will touch briefly on the basics of Spark. Basically, Spark is an computing engine for a large scale data set. It utilizes in-memory computations for speed and can work with a large number of different use cases. Spark's underlying abstraction is a Resilient Distributed Dataset. A somewhat recent addition to Spark are DataFrames, which we will be using through this unit. DataFrames go hand in hand with Spark SQL, which is a library that sites on top of the Spark Core. You can develop your applications in Scala, Python, Java, and also new to Spark, is the R language. Of course, there are other libraries that are not within the scope of this course, but I just listed them here.



Resilient Distributed Datasets (RDD)

A little bit about RDDs. These are the fault tolerant collection of elements that can be operated on in parallel. There are two types of operations. Actions are operations that returns some sort of value back to the caller. Transformations basically updates the direct acyclic graphs of the RDD that Spark maintains for its fault tolerance. Essentially transformations doesn't actually do anything to the current state of the application. It is not until an Action is called, that ALL the transformations since the last action, leading up to the called actions are executed. These are called lazy evaluations, where no work is done until an action is called.

Here I show a very simple example, of reading in a README file and then just counting the number of lines that contains word Spark. The **filter** operation is a transformation, so nothing gets returned. It isn't until the **count** operation that the value of 141 was returned to the caller.

IBM Training

Spark SQL and DataFrames

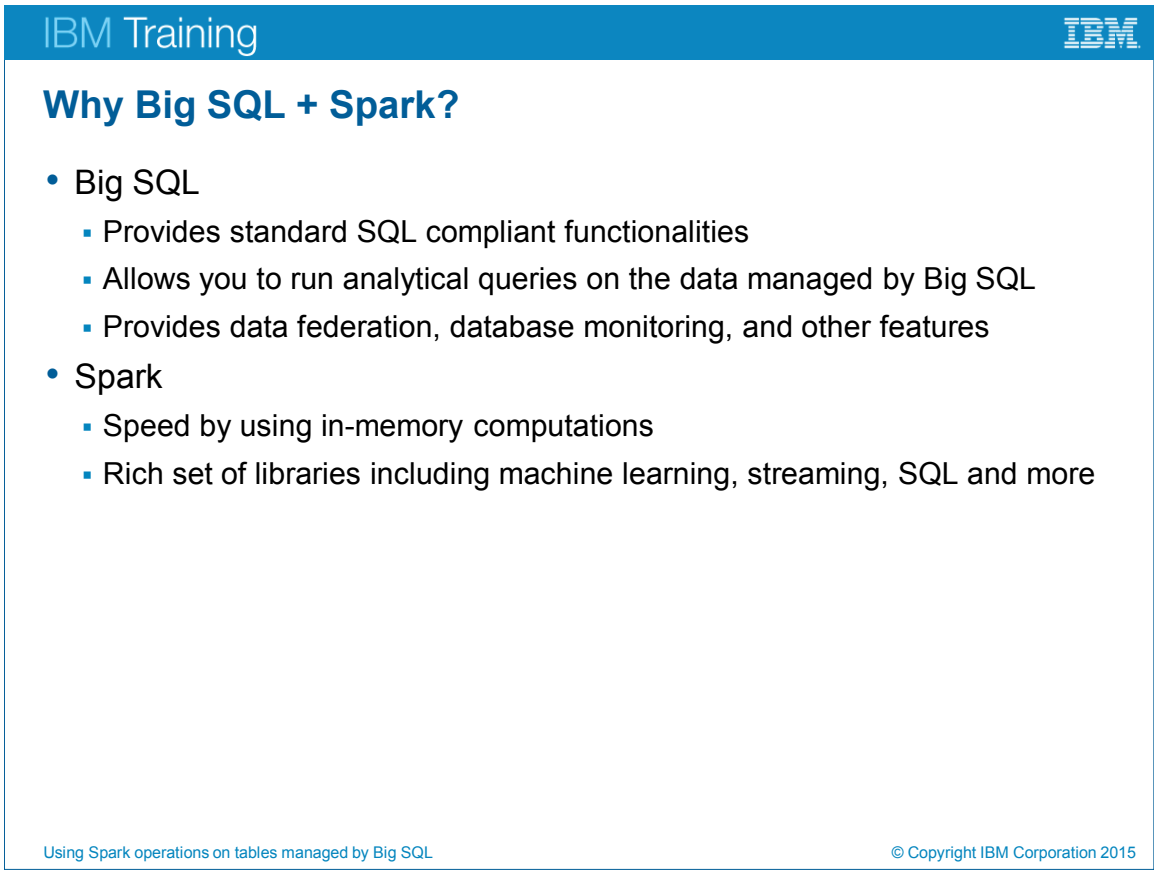
- **Spark SQL:** Spark module for structured data processing
- **DataFrames:** Distributed collection of data organized into named columns
  - Thought of as a table in a relational database
  - Created from sources such as:
    - Structured data files
    - Tables in Hive
    - External databases
    - Existing RDDs

```
val sc: SparkContext // An existing SparkContext.
val sqlContext = new org.apache.spark.sql.SQLContext(sc)

// creating DataFrames
val df = sqlContext.read.json("examples/src/main/resources/people.json")

// Displays the content of the DataFrame to stdout
df.show()
```

Using Spark operations on tables managed by Big SQL

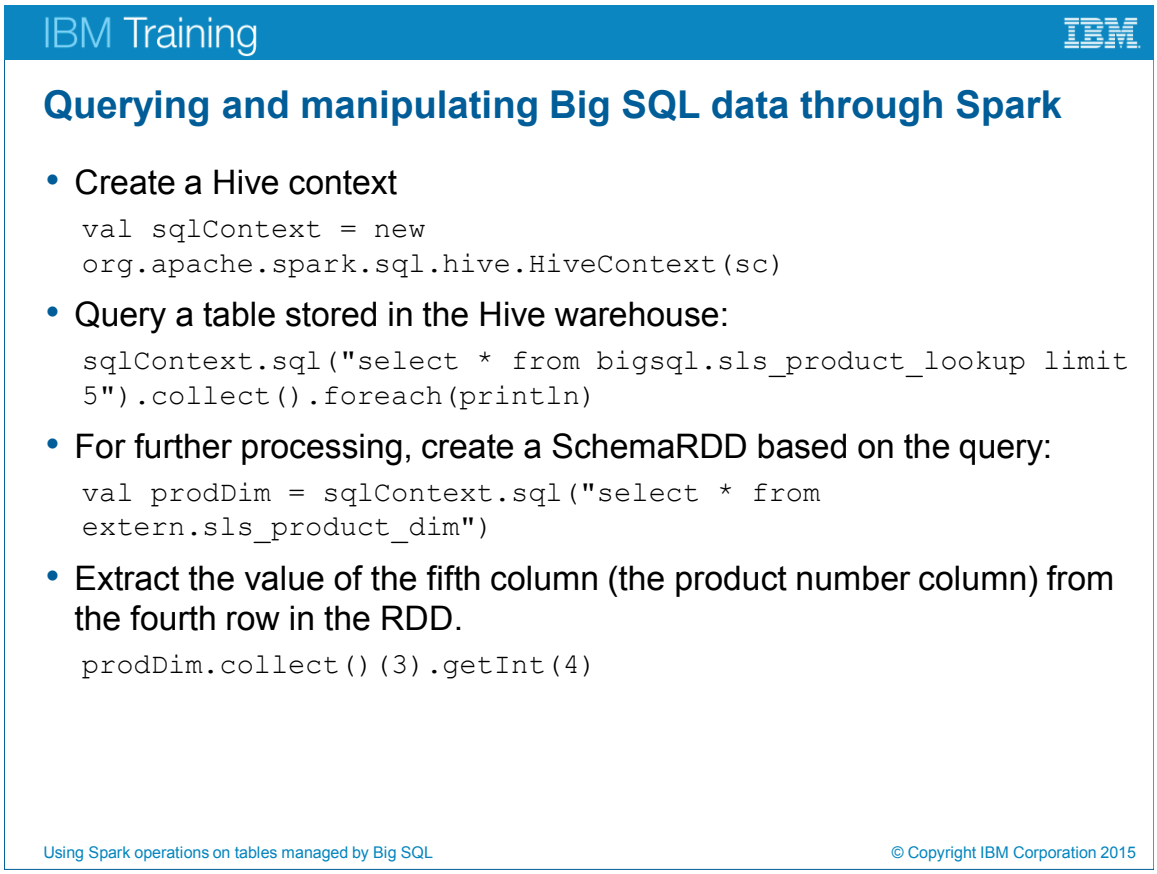
Spark SQL and DataFrames

The abstraction of Spark SQL is DataFrames. You can think of DataFrames as a table from a relational database. You can create DataFrames from structured data files, Hive tables, external databases, or existing RDDs. Because Big SQL tables are stored in the Hive catalogs, which are just essentially a logical view of the data residing in the HDFS, you can seamlessly integrate it with Spark to run Spark operations in it. This does not limit you to just Spark SQL. In fact, once you set this up, you can take advantage of all of the Spark libraries to perform other types of operations that Big SQL was not design to support such as machine learning or graph computations algorithms.



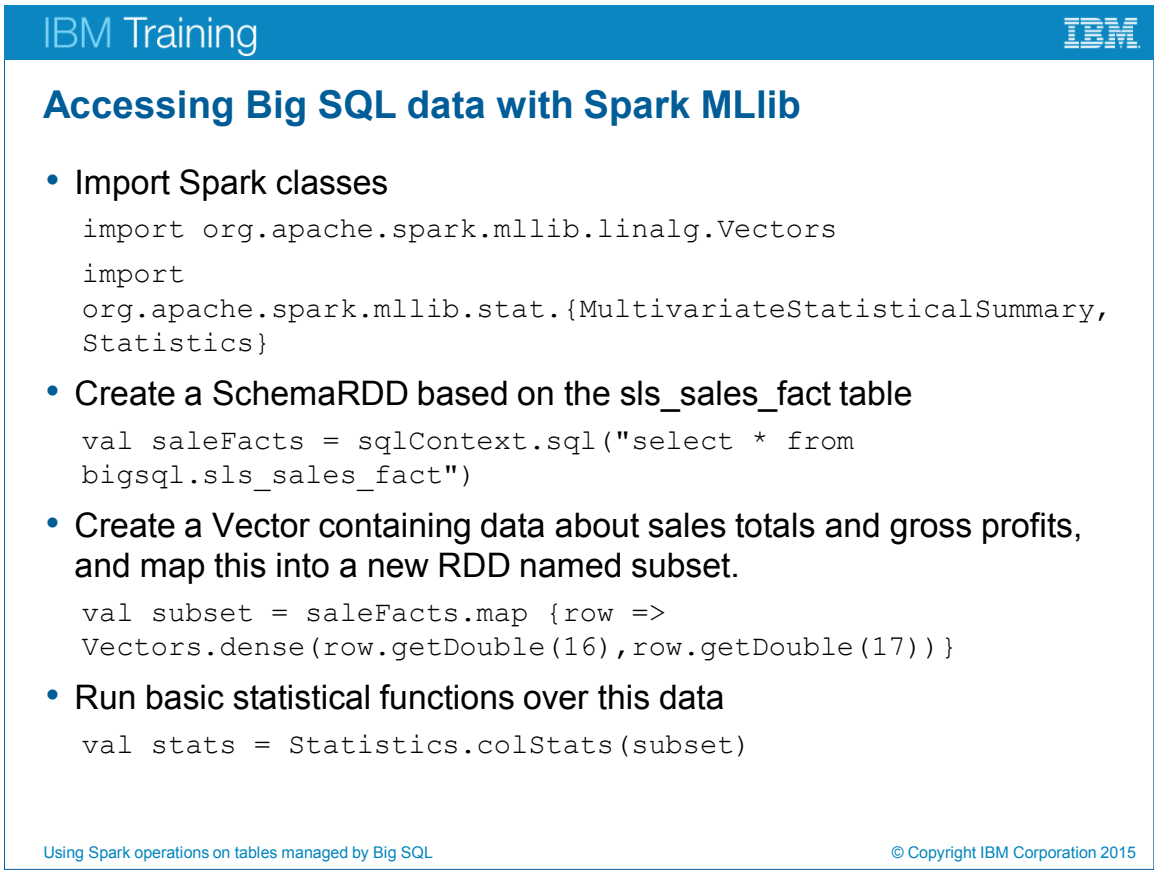
Why Big SQL + Spark?

Using Big SQL and Spark will give you the advantage and the capabilities of both. With Big SQL, as you are well aware by now, having access to a compliant SQL language makes it easy for you to run analytical queries against the data. What's more, the data resides directly on HDFS o Hive warehouse so there is no proprietary storage format allowing interoperability with many of technologies, such as Spark. Spark gives you access to operations and libraries, including machine learning, streaming, SQL and more. In the remainder of this unit, you will see, in particular, how to use Spark SQL and Spark MLlib on the exact same data that is managed by Big SQL tables.



Querying and manipulation Big SQL data through Spark

To use Spark with Big SQL, you first need to create a Hive Context from within the Spark shell. The Hive Context is created from the default Spark context. This is what you will use to access the data, which is managed by Big SQL. To query the table, you provide the query statement and additional Spark operations to print out the results. If you wish to do further processing on the data, then save the results to a SchemaRDD. Note that, for most other operations, you would use DataFrames, but for Hive, it is still recommended at the time of the development of this course, to use SchemaRDD. Refer to the Spark documentation for the latest best practice. Finally, the rest of the examples show how you are continue to use Spark operations and Spark SQL to get more out of your data.



Accessing Big SQL data with Spark MLlib

Spark provides additional libraries that you can use on the data managed by Big SQL. Here is an example of using some machine learning algorithms to find some basic statistics on the data. While some of these algorithms can probably be done in Big SQL with a lot of query manipulation, you can see here how easy and simple it is to perform basic statistical functions when you become familiar with the library. Again, this is just one example of what you can do once you set up Spark to access the same Big SQL data. All the Spark libraries are then at your disposal.

IBM Training

Checkpoint

1. List at least one of the data sources that Spark supports?
2. Which data source allows both Spark and Big SQL to work together?
3. What is one reason why you would want to use Spark with your Big SQL tables?

Checkpoint

IBM Training

Checkpoint solutions

1. List at least one of the data sources that Spark supports?
   - Structured data files
   - Tables in Hive
   - External databases
   - Existing RDDs
2. Which data source allows both Spark and Big SQL to work together?
   - The Hive warehouse
3. What is one reason why you would want to use Spark with your Big SQL tables?
   - Spark can provide additional analytical capabilities through the use of its libraries, including machine learning, streaming, graph computations, etc.

Checkpoint solutions

IBM Training

Demonstration 1

Using Spark operations on tables managed by Big SQL

At the end of this demonstration, you will be able to:

- Work with data in Big SQL tables using the Spark shell.
- Create a Spark Schema RDD (resilient distributed dataset) from data in Big SQL tables.
- Query and extract data from Big SQL tables using Spark SQL.
- Expose data in Big SQL columns as a Spark MLlib data type (Vector) and invoke a simple MLlib statistical function over this data.

Demonstration 1: Using Spark operations on tables managed by Big SQL

Demonstration 1: Using Spark operations on tables managed by Big SQL

Purpose:
In this demonstration you will explore how Spark programmers can work with data managed by Big SQL. As you may know, Big SQL is a popular component of several IBM BigInsights offerings. It enables SQL professionals to query data stored in Hadoop using ISO SQL syntax; it also provides database monitoring, query federation, and other features. Apache Spark, part of IBM's Open Platform for Apache Hadoop and BigInsights, is a fast, general-purpose engine for processing Big Data, including data managed by Hadoop. Particularly appealing to many Spark programmers are built-in and third-party libraries for machine learning, streaming, SQL, and more.

Given the popularity of both Big SQL and Spark, it's reasonable to expect organizations to want to deploy and use both technologies. This demonstration introduces you to one way in which organizations can integrate these technologies, namely, by creating, populating, and manipulating Big SQL tables stored in HDFS directories or the Hive warehouse and then accessing data from these tables through Spark SQL and Spark MLlib.

Estimated time: 45 minutes

User/Password: **biadmin/biadmin**
root/dalvm3

Services Password: **ibm2blue**

Task 1. Creating and populating your Big SQL sample tables.

Create two tables in the Hive warehouse using your default schema (which will be "bigsql" if you connected into your database as that user). The first table is part of the PRODUCT dimension and includes information about product lines in different languages. The second table is the sales FACT table, which tracks transactions (orders) of various products. The statements below create both tables in a TEXTFILE format. Within each row, fields are separated by tabs ("\t"). Furthermore, each line is terminated by new line character ("\n").

Both of these tables should have been created in one of the earlier demonstrations. Go back and create them now if you haven't done so already. You will be working with these two tables in the default *bigsql* schema.

```
sls_product_line_lookup
sls_sales_fact
```

The 1_Big SQL Spark statements.sql file is located in the */home/biadmin/labfiles/bigsql/Big_SQL_Spark* folder.

Query the tables to verify that the expected number of rows was loaded into each table.

1. Execute each query below and compare the results with the number of rows specified in the comment line preceding each query.

```
-- total rows in SLS_PRODUCT_LINE_LOOKUP = 5
select count(*) from SLS_PRODUCT_LINE_LOOKUP;

-- total rows in SLS_SALES_FACT = 446023
select count(*) from SLS_SALES_FACT;
```

Next you will create 1 externally managed Big SQL table - i.e., a table created over a user directory that resides outside of the Hive warehouse. This user directory will contain all the table's data in files. Creating such a table effectively layers a SQL schema over existing DFS data (or data that you may later upload into the target DFS directory).

2. Open a new terminal.
3. From the command line, issue this command to switch to the root user ID temporarily:

```
su root
```

4. Create a directory structure in your distributed file system for the source data file for the product dimension table. Ensure public read/write access to this directory structure. (If desired, alter the DFS information as appropriate for your environment.)

```
hdfs dfs -mkdir /user/bigsql_spark_lab
hdfs dfs -mkdir /user/bigsql_spark_lab/sls_product_dim
hdfs dfs -chmod -R 777 /user/bigsql_spark_lab
```

5. Upload the source data file (the Big SQL sample data file named GOSALESDW.SLS_PRODUCT_DIM.txt) into the target DFS directory. Change the local and DFS directories information below to match your environment.

```
hdfs dfs -copyFromLocal
/usr/ibmpacks/bigsql/4.0/bigsql/samples/data/GOSALESDW.SLS
_PRODUCT_DIM.txt
/user/bigsql_spark_lab/sls_product_dim/SLS_PRODUCT_DIM.txt
```

6. List the contents of the DFS directory and verify that your sample data file is present.

```
hdfs dfs -ls /user/bigsql_spark_lab/sls_product_dim
```

7. Return to your Big SQL query execution environment (JSqsh or the Big SQL console).
8. Create an external Big SQL table for the sales product dimension (extern.sls_product_dim).
   Note that the LOCATION clause references the DFS directory into which you copied the sample data.

```
-- product dimension table
CREATE EXTERNAL HADOOP TABLE IF NOT EXISTS
extern.sls_product_dim
 ( product_key INT NOT NULL
 , product_line_code INT NOT NULL
 , product_type_key INT NOT NULL
 , product_type_code INT NOT NULL
 , product_number INT NOT NULL
 , base_product_key INT NOT NULL
 , base_product_number INT NOT NULL
 , product_color_code INT
 , product_size_code INT
 , product_brand_key INT NOT NULL
 , product_brand_code INT NOT NULL
 , product_image VARCHAR(60)
 , introduction_date TIMESTAMP
 , discontinued_date TIMESTAMP
 )
 ROW FORMAT DELIMITED
 FIELDS TERMINATED BY '\t'
 LINES TERMINATED BY '\n'
 location '/user/bigsql_spark_lab/sls_product_dim';
```

9. Verify that you can query the table.

```
--total rows in EXTERN.SLS_PRODUCT_DIM = 274
select count(*) from EXTERN.SLS_PRODUCT_DIM;
```

Task 2. Setting up your Spark environment.

Now that you've created and populated the sample Big SQL tables required for this demonstration, it's time to experiment with accessing their data. In this module, you'll adjust the level of detail returned by the Spark shell so that only essential messages are returned. The Spark shell can be verbose, so suppressing informational messages will help you focus on the tasks at hand. Next, you'll launch the Spark shell.

1. Open up a new terminal, or use an existing one.
2. Switch to the **root** user, password **dalvm3**, if required.
3. Switch to the **spark** user.
4. As the *spark* user, change directories to Spark home:

```
cd /home/spark
```

5. Create a new file named **log4j.properties** using your favorite editor. Instructions in this demonstration are based on the vi editor.

```
vi log4j.properties
```

6. In vi, enter insert mode. Type

```
i
```

7. Cut and paste the following content into the file:

```
# Set everything to be logged to the console
log4j.rootCategory=ERROR, console
log4j.appender.console=org.apache.log4j.ConsoleAppender
log4j.appender.console.target=System.err
log4j.appender.console.layout=org.apache.log4j.PatternLa
yout
log4j.appender.console.layout.ConversionPattern=%d{yy/MM
/dd HH:mm:ss} %p %c{1}: %m%n

# Settings to quiet third party logs that are too
verbose
log4j.logger.org.eclipse.jetty=WARN
log4j.logger.org.eclipse.jetty.util.component.AbstractLi
feCycle=ERROR
log4j.logger.org.apache.spark.repl.SparkIMain$exprTyper=
INFO
log4j.logger.org.apache.spark.repl.SparkILoop$SparkILoop
Interpreter=INFO
```

8. To exit the file, press **Esc** and then enter.

```
:wq
```


9. Verify that your new log4j.properties file was successfully created and contains the right content.

```
more log4j.properties
```

10. Launch the Spark shell, which will enable you to issue Scala statements and expressions:

```
spark-shell
```

Task 3. Querying and manipulating Big SQL data through Spark.

In this task, you'll issue Scala commands and expressions from the Spark shell to retrieve Big SQL data. Specifically, you'll use Spark SQL to query data in Big SQL tables. You'll model your result sets from your queries as SchemaRDDs, a specific type of resilient distributed dataset (RDD). SchemaRDDs consist of Row objects and a schema describing each column in the row. For details on RDDs and SchemaRDDs, visit the Spark web site.

1. From the Spark shell, establish a Hive context named sqlContext:

```
val sqlContext = new
org.apache.spark.sql.hive.HiveContext(sc)
```

```
scala> val sqlContext = new org.apache.spark.sql.hive.HiveContext(sc)
sqlContext: org.apache.spark.sql.hive.HiveContext = org.apache.spark.sql.hive.Hi
veContext@c349c1d
```

The Hive context enables you to find tables in the Hive meta store (HCatalog) and issue HiveQL queries against these tables.

2. Experiment with querying a Big SQL table stored in the Hive warehouse.

```
sqlContext.sql("select * from bigsql.sls_product_lookup
limit 5").collect().foreach(println)
```

Let's examine this statement briefly. Using the Hive context established previously, we issue a simple query (using HiveQL syntax) to retrieve 5 rows from the *sls_product_lookup* table in the *bigsql* Hive schema. We call other Spark functions to collect the result and print each record.

```
scala> sqlContext.sql("select * from bigsql.sls_product_lookup limit 5").collect
().foreach(println)
[1110,CS,Vak na vodu Kuchtík,Lehký, skladný vak na tekutiny. Široké hrdlo usnadn
í plnění. Objem 10 litrů.]
[1110,DA,Sahara Vandtaske,Sammentrykkelig letvægtstaske til væsker. Bred åbning
til nem påfyldning. Indeholder 10 liter.]
[1110,DE,TrailChef Wasserbeutel,Leichter, zusammenfaltbarer Beutel zum einfachen
 Transport von Flüssigkeiten. Breite Öffnung zum einfachen Einfüllen. Fassungsve
rmögen 10 Liter.]
[1110,EL,Δοχείο υγρών Μαρμίτα Σεφ,Ελαφρύ, πτυσσόμενο δοχείο για εύκολη μεταφορά
υγρών.  Ευρύ στόμιο για εύκολη πλήρωση.  Χωρητικότητα 10 λίτρα.]
[1110,EN,TrailChef Water Bag,Lightweight, collapsible bag to carry liquids easil
y.  Wide mouth for easy filling.  Holds 10 liters.]
```

3. Issue a similar query against an externally managed Big SQL table (i.e., a Big SQL table created over an HDFS directory.

```
sqlContext.sql("select * from extern.sls_product_dim
limit 5").collect().foreach(println)
```

```
scala> sqlContext.sql("select * from extern.sls_product_dim limit 5").collect().
foreach(println)
[30001,991,951,951,1110,1,1,908,808,701,701,P01CE1CG1.jpg,1995-02-15 00:00:00.0,
null]
[30002,991,951,951,2110,2,2,906,807,701,701,P02CE1CG1.jpg,1995-02-15 00:00:00.0,
null]
[30003,991,951,951,3110,3,3,924,825,701,701,P03CE1CG1.jpg,1995-02-15 00:00:00.0,
null]
[30004,991,951,951,4110,4,4,923,804,701,701,P04CE1CG1.jpg,1995-02-15 00:00:00.0,
null]
[30005,991,951,951,5110,5,5,923,823,701,701,P05CE1CG1.jpg,1995-02-15 00:00:00.0,
null]
```

4. To make the result set readily available for further processing in Spark, create a SchemaRDD based on a query. In this case, we'll query the externally managed table.

```
val prodDim = sqlContext.sql("select * from
extern.sls_product_dim")
```

```
scala> val prodDim = sqlContext.sql("select * from extern.sls_product_dim")
prodDim: org.apache.spark.sql.SchemaRDD =
SchemaRDD[0] at RDD at SchemaRDD.scala:108
== Query Plan ==
== Physical Plan ==
HiveTableScan [product_key#0,product_line_code#1,product_type_key#2,product_type
_code#3,product_number#4,base_product_key#5,base_product_number#6,product_color_
code#7,product_size_code#8,product_brand_key#9,product_brand_code#10,product_ima
ge#11,introduction_date#12,discontinued_date#13], (MetastoreRelation extern, sls
_product_dim, None), None
```

5. Extract the value of the fifth column (the product number column) from the fourth row in the RDD. Since array elements begin at 0, issue the following statement:

```
prodDim.collect()(3).getInt(4)
```

```
scala> prodDim.collect()(3).getInt(4)
res3: Int = 4110
```

Let's step through the logic of this statement briefly. The *collect()* function returns an array containing all elements in the referenced RDD. Since array indexing begins at 0, specifying (3) indicates that we want to work only with the fourth row. The getInt(4) function retrieves the fifth column for this row. Given the Big SQL table definition, this is the value for the PRODUCT_NUMBER column (an integer).

6. Experiment with using the Spark map() function to create a new RDD named prodNum based on prodDim. In this case, the transformation performed by the map() function will be simple – it will simply extract the product numbers from prodDim. (Recall that product numbers reside in the fifth field of prodDim and that array indexing begins at zero.)

```
val prodNum = prodDim.map (row => row.getInt(4))
```

```
scala> val prodNum = prodDim.map (row => row.getInt(4))
prodNum: org.apache.spark.rdd.RDD[Int] = MappedRDD[5] at map at <console>:16
```

7. Use the count() function to verify that prodNum contains 274 records.

```
prodNum.count()
```

```
scala> prodNum.count()
res4: Long = 274
```

8. Optionally, validate these results through Big SQL.

   a. Open a second terminal window and launch JSqsh.

   b. Connect to your Big SQL database.

   c. Count the number of product number records in the sls_product_dim table.

```
select count(product_number) from
extern.sls_product_dim;
```

   d. Verify that 274 rows are present. Note that this matches the count of the prodNum RDD you defined in Scala.

Task 4. Accessing Big SQL data with Spark MLlib.

Now that you understand how to use Spark SQL to work with data managed by Big SQL tables, let's explore how to use other Spark technologies to manipulate this data. In this task, you'll transform Big SQL data into a data type commonly used with Spark's machine learning library (MLlib). Then you'll invoke a simple MLlib function on that data.

1. If necessary, return to the Spark shell that you launched in the previous task.
2. Import Spark classes that you'll be using shortly.

```
import org.apache.spark.mllib.linalg.Vectors

import
org.apache.spark.mllib.stat.{MultivariateStatisticalSumm
ary, Statistics}
```

```
scala> import org.apache.spark.mllib.linalg.Vectors
import org.apache.spark.mllib.linalg.Vectors

scala> import org.apache.spark.mllib.stat.{MultivariateStatisticalSummary, Stati
stics}
import org.apache.spark.mllib.stat.{MultivariateStatisticalSummary, Statistics}
```

3. Create a SchemaRDD named saleFacts based on data in the bigsql.sls_sales_fact table.

```
val saleFacts = sqlContext.sql("select * from
bigsql.sls_sales_fact")
```

```
scala> val saleFacts = sqlContext.sql("select * from bigsql.sls_sales_fact")
saleFacts: org.apache.spark.sql.SchemaRDD =
SchemaRDD[80] at RDD at SchemaRDD.scala:108
== Query Plan ==
== Physical Plan ==
HiveTableScan [order_day_key#271,organization_key#272,employee_key#273,retailer_
key#274,retailer_site_key#275,product_key#276,promotion_key#277,order_method_key
#278,sales_order_key#279,ship_day_key#280,close_day_key#281,quantity#282,unit_co
st#283,unit_price#284,unit_sale_price#285,gross_margin#286,sale_total#287,gross_
profit#288], (MetastoreRelation bigsql, sls_sales_fact, None), None
```

4. Count the records in saleFacts, verifying that 446023 are present.

```
saleFacts.count()
```

5. Create a Vector containing data about sales totals and gross profits, and map this into a new RDD named subset.

```
val subset = saleFacts.map {row =>
Vectors.dense(row.getDouble(16),row.getDouble(17))}
```

6. Run basic statistical functions over this data.

```
val stats = Statistics.colStats(subset)
```

This operation may take several minutes to complete, depending on your machine resources. If the operation appears to be hung, press Enter on your keyboard to see if it's completed.

7. Print various statistical results.

```
println(stats.mean)

println(stats.variance)

println(stats.max)
```

```
scala> println(stats.mean)
[10507.92396098396,4315.550979837396]

scala> println(stats.variance)
[3.29316420180777E8,5.02623271828062E7]

scala> println(stats.max)
[363575.08,163736.73]
```

8. Optionally, validate one of these statistical results through Big SQL.

   a. If necessary, open a terminal window and launch JSqsh.

   b. Connect to your Big SQL database.

   c. Issue these queries to determine the maximum sales total and maximum gross profit values in the bigsql.sls_sales_fact table:

```
select max(sale_total) from  bigsql.sls_sales_fact;

select max(gross_profit) from bigsql.sls_sales_fact;
```

```
[bdvs1052.svl.ibm.com][bigsql] 1> select max(sale_total) from  bigsql.sls_sales_fact;
+--------------+
|            1 |
+--------------+
| 363575.08000 |
+--------------+
1 row in results(first row: 0.44s; total: 0.44s)
[bdvs1052.svl.ibm.com][bigsql] 1> select max(gross_profit) from bigsql.sls_sales_fact;
+--------------+
|            1 |
+--------------+
| 163736.73000 |
+--------------+
1 row in results(first row: 0.47s; total: 0.47s)
```

Note that these values equal the results computed by the Spark MLlib function that you invoked for the maximum values in this data set.

In this lab, you explored one way of using Spark to work with data in Big SQL tables stored in the Hive warehouse or DFS directories. From the Spark shell, you established a Hive context, queried Big SQL tables using Hive's query syntax, performed basic Spark transactions and actions on the data, and even applied a simple statistical function from Spark MLlib on the data.

Results:
In this demonstration you explored how Spark programmers can work with data managed by Big SQL. You were introduced you to one way in which organizations can integrate these technologies, namely, by creating, populating, and manipulating Big SQL tables stored in HDFS directories or the Hive warehouse and then accessing data from these tables through Spark SQL and Spark MLlib.

IBM Training

Unit summary

- Describe the purpose and role of Spark
- Querying and manipulating Big SQL data through Spark

Unit summary

IBM Training