Getting started with BigInsights 4.2 - BigSQL

June, 2017

Contents

Lab 1 Overview

In this hands-on lab, you'll learn how to work with key Big SQL, a component of IBM's big data platform based on Apache Hadoop. Big SQL is included with several IBM BigInsights offerings.

Big SQL enables IT professionals to create tables and query data in BigInsights using familiar SQL statements. To do so, programmers use standard SQL syntax and, in some cases, SQL extensions created by IBM to make it easy to exploit certain Hadoop-based technologies. Big SQL shares query compiler technology with DB2 (a relational DBMS) and, as such, offers a wide breadth of SQL capabilities.

Organizations interested in Big SQL often have considerable SQL skills in-house, as well as a suite of SQL-based business intelligence applications and query/reporting tools. The idea of being able to leverage existing skills and tools — and perhaps reuse portions of existing applications — can be quite appealing to organizations new to Hadoop. Indeed, some companies with large data warehouses built on relational DBMS systems are looking to Hadoop-based platforms as a potential target for offloading "cold" or infrequently used data in a manner that still allows for query access. In other cases, organizations turn to Hadoop to analyze and filter non-traditional data (such as logs, sensor data, social media posts, etc.), ultimately feeding subsets or aggregations of this information to their relational warehouses to extend their view of products, customers, or services.

1.1. What you'll learn

After completing all exercises in this lab guide, you'll know how to

- Inspect the status of your Big SQL service through Apache Ambari, a Web-based management tool included with the IBM Open Platform for Apache Hadoop.
- Create a connection to your Big SQL server from a command line environment (JSqsh).
- Execute Big SQL statements and commands.
- Create Big SQL tables stored in the Hive warehouse and in user-specified directories of your Hadoop Distributed File System (HDFS).
- Load data into Big SQL tables.
- Query big data using Big SQL projections, restrictions, joins, and other operations.
- Gather statistics about your tables and explore data access plans for your queries.
- Create and execute SQL-based scalar and table user-defined functions.
- Work with Big SQL web tooling to explore database metrics and perform other tasks.

1.2. About your environment

This lab requires a BigInsights 4.2 environment in which Big SQL is installed and running.

Examples in this lab use are based on a sample environment with the configuration shown in the tables below. **If your environment is different, modify the sample code and instructions as needed to match your configuration.**

	User	Password
Root account	root	passw0rd
Big SQL Administrator	bigsql	passw0rd
Ambari Administrator	admin	admin
Knox Gateway account	guest	guest-password

Property	Value
Host name	IP on each machine
Ambari port number	8080
Big SQL database name	bigsql
Big SQL port number	32051
Big SQL installation directory	/usr/ibmpacks/bigsql
JSqsh installation directory	/usr/ibmpacks/common-utils/current/jsqsh
Big SQL samples directory	/usr/ibmpacks/bigsql/4.2.0.0/bigsql/samples/data
BigInsights Home	https://bigi01.localdomain:8443/gateway/default/BigInsightsWeb/index.html

You can find all the resources on GitHub:

https://github.com/fjcanobailen/biginsights

About the screen captures, sample code, and environment configuration

Screen captures in this lab depict examples and results that may vary from what you see when you complete the exercises. In addition, some code examples may need to be customized to match your environment.

Lab 2 Exploring your Big SQL service through Ambari

Administrators can monitor, launch, and inspect aspects of their Big SQL service through Ambari, an open source Web-based tool for managing Hadoop clusters. In this section, you will learn how to

- Launch Ambari.
- Inspect the overall status of your cluster.
- Inspect the configuration and status of your Big SQL service.
- Identify the software repository (build source) of your Big SQL service.

[Allow **30 minutes** to complete this section.]

2.1. Inspecting your cluster status

In this exercise, you will launch Apache Ambari and verify that a minimal set of services are running so that you can begin working with Big SQL. You will also learn how to stop and start a service.

__1. Launch a Web browser.

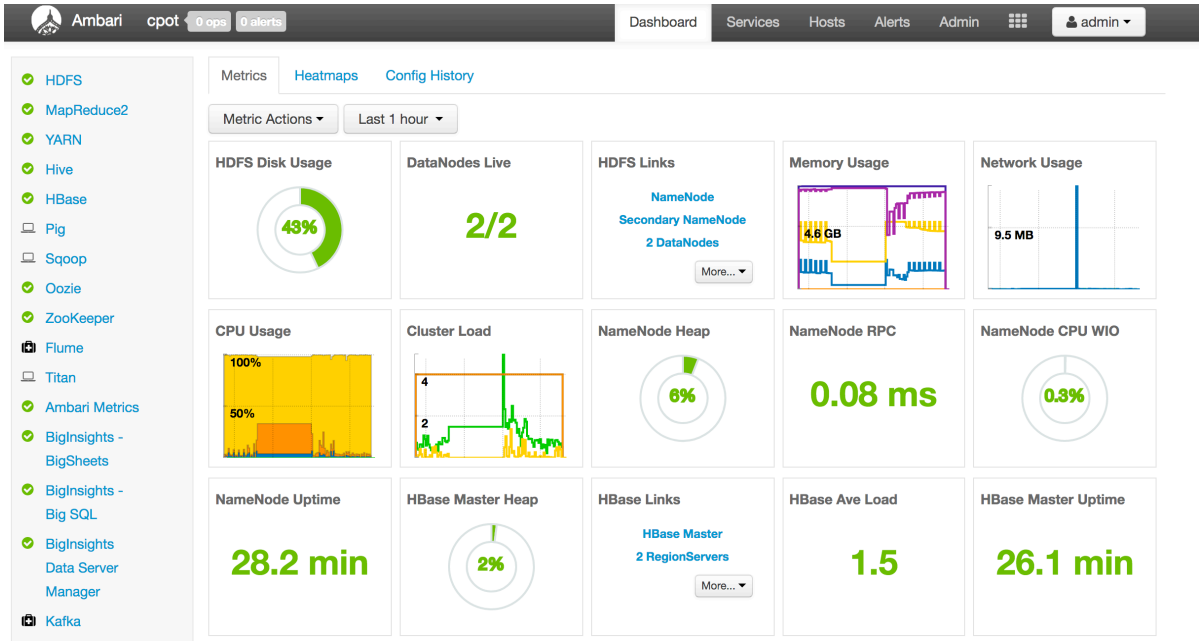
__2. Enter the URL for your Ambari service, which was configured at installation. For example, if the host running your Ambari server is *bigi01.localdomain* and the server was installed at its default port of 8080, you would enter

```
http://bigi01.localdomain:8080
```

__3. When prompted, enter the Ambari administrator ID and password. (By default, this is `admin/admin`).

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`

__4. Verify that the Ambari console appears similar to this:

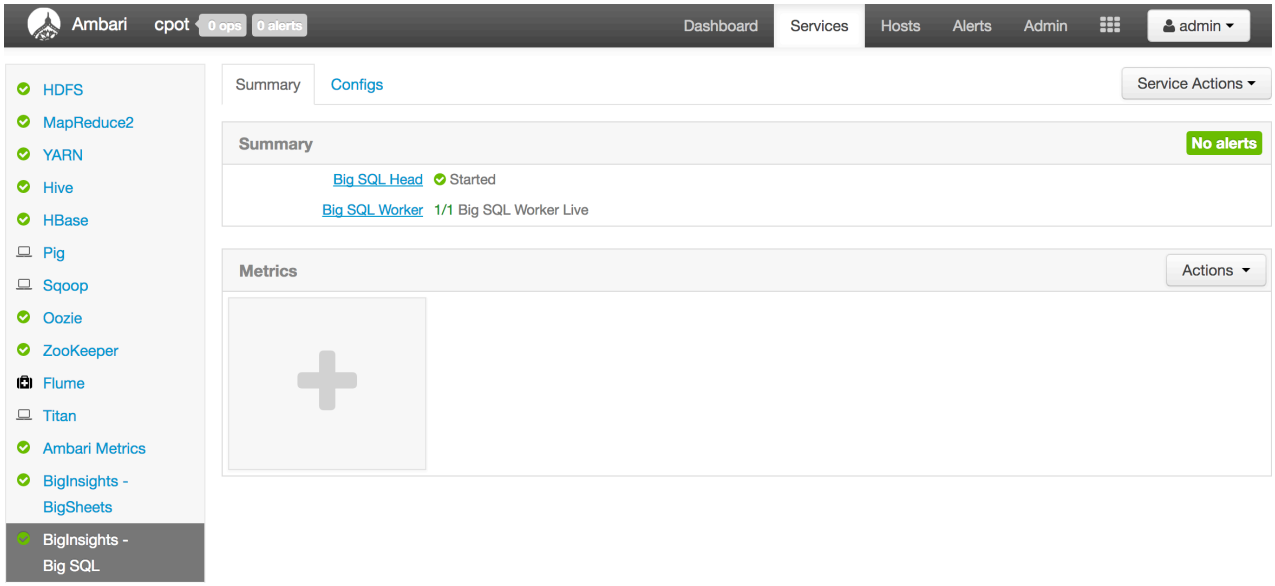


__5. If necessary, click the Dashboard tab at the top of the screen and inspect the overall status of services that have been installed. The previous screen capture was taken from a system in which all open source components provided in the IBM Open Platform for Apache Hadoop had been started. The Big SQL service was also started.

2.2. Exploring your Big SQL service

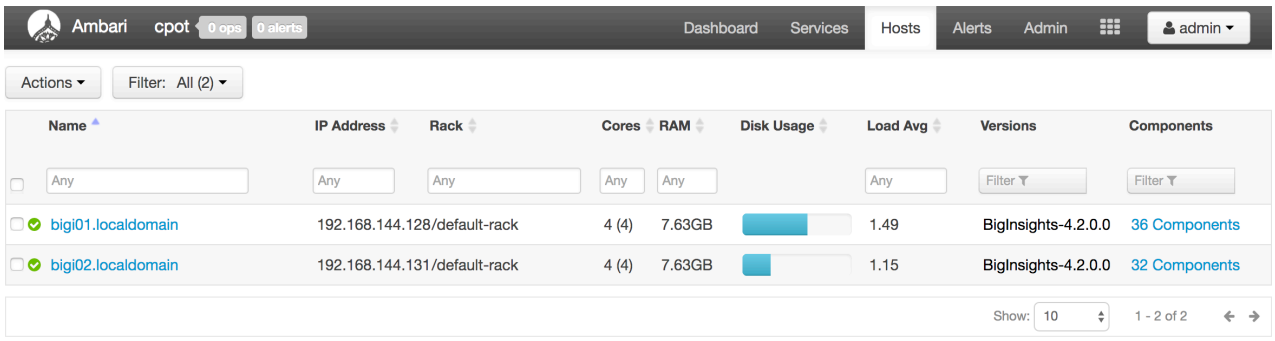
Let's explore the configuration of your Big SQL service, including the nodes on which various Big SQL artifacts have been installed. This is important for subsequent exercises, as you need to know where Big SQL client software resides and where the Big SQL Head Node resides. Big SQL client software includes JSqsh, a command-line interface that you'll use in a subsequent lesson. To connect to Big SQL and issue commands or queries, you need to specify the JDBC URL of the Big SQL Head Node.

__1. In Ambari, click on the BigInsights - Big SQL service to display details about it.

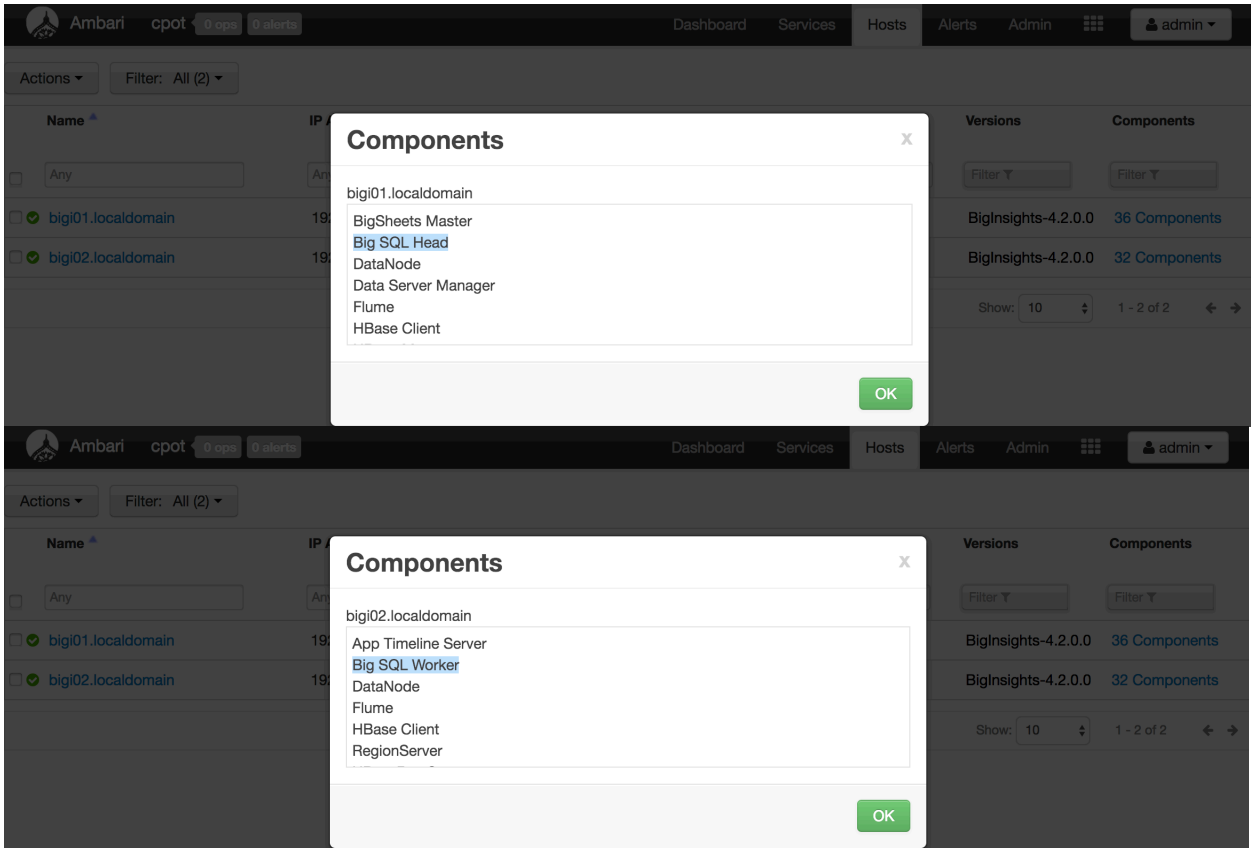


__2. Examine the overview information presented in the Summary tab. In the previous screen capture, such as example of a cluster, you'll note that there is a Big SQL worker installed and running.

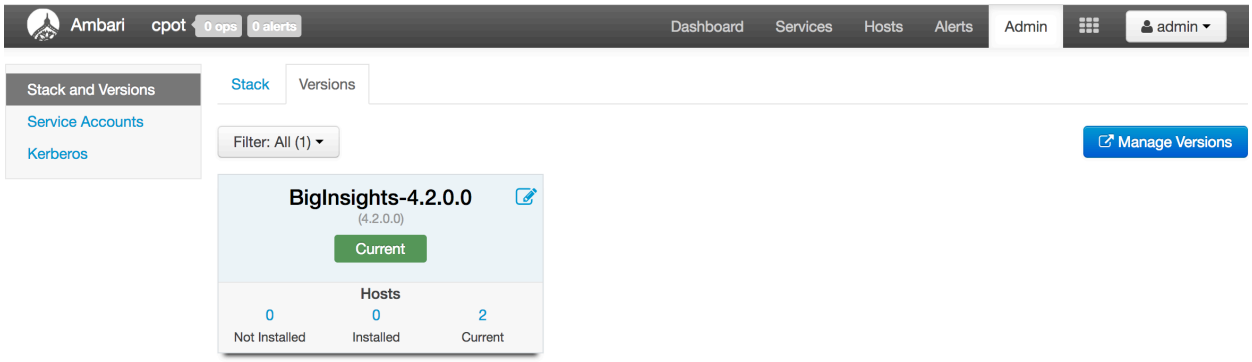
__3. Click on the Hosts tab towards the upper right of your screen. A summary of the nodes in your cluster is displayed. The image below was taken from a 2-node cluster.



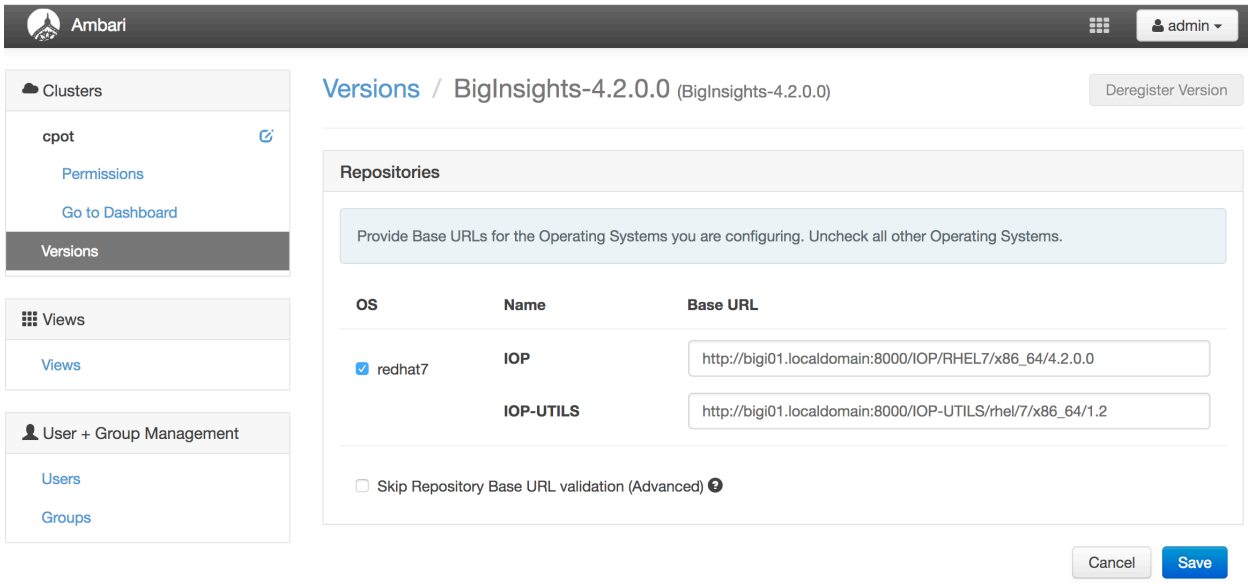
__4. Expand the components information for each node to inspect the installed services. In particular, note which node contains the Big SQL Head node and which node(s) contain the Big SQL Workers.



__5. Optionally, explore the software repository associated with your Big SQL installation. In the upper right corner, click Admin > Stack and Versions. Scroll to the bottom of the displayed page and note the repositories associated with the software you have installed on your cluster. This information indicates the build levels for your core services (IBM Open Platform for Apache Hadoop), core utilities, and IBM value packs.



Admin (White Button) > Manage Ambari > Versions:


Ambari
admin
Clusters
cpot
Permissions
Go to Dashboard
Versions
Views
Views
User + Group Management
Users
Groups
Versions / BigInsights-4.2.0.0 (BigInsights-4.2.0.0)
Deregister Version
Repositories
Provide Base URLs for the Operating Systems you are configuring. Uncheck all other Operating Systems.
OS
Name
Base URL
redhat7
IOP
http://bigi01.localdomain:8000/IOP/RHEL7/x86_64/4.2.0.0
IOP-UTILS
http://bigi01.localdomain:8000/IOP-UTILS/rhel/7/x86_64/1.2
Skip Repository Base URL validation (Advanced)
Cancel
Save

Lab 3 Exploring Big SQL web tooling: Data Server Manager (DSM)

Big SQL provides web tools that you can use to inspect database metrics, issue SQL statements, and perform other functions. These tools are part of IBM Data Server Manager (DSM), a component you can install and configure on your Big SQL Head Node. This lab introduces you to a few features of DSM.

After completing this lab, you will know how to:

- Launch the BigInsights Home page and the Big SQL web tooling (DSM).
- Execute Big SQL queries and inspect result sets from DSM.
- Inspect metrics and monitoring information collected for your Big SQL database.

Prior to beginning this lab, you will need access to a BigInsights cluster in which BigInsights Home, Big SQL, DSM, and Ambari are running. You also need to have created and populated the sample tables presented in a prior lab on Querying Structured Data.

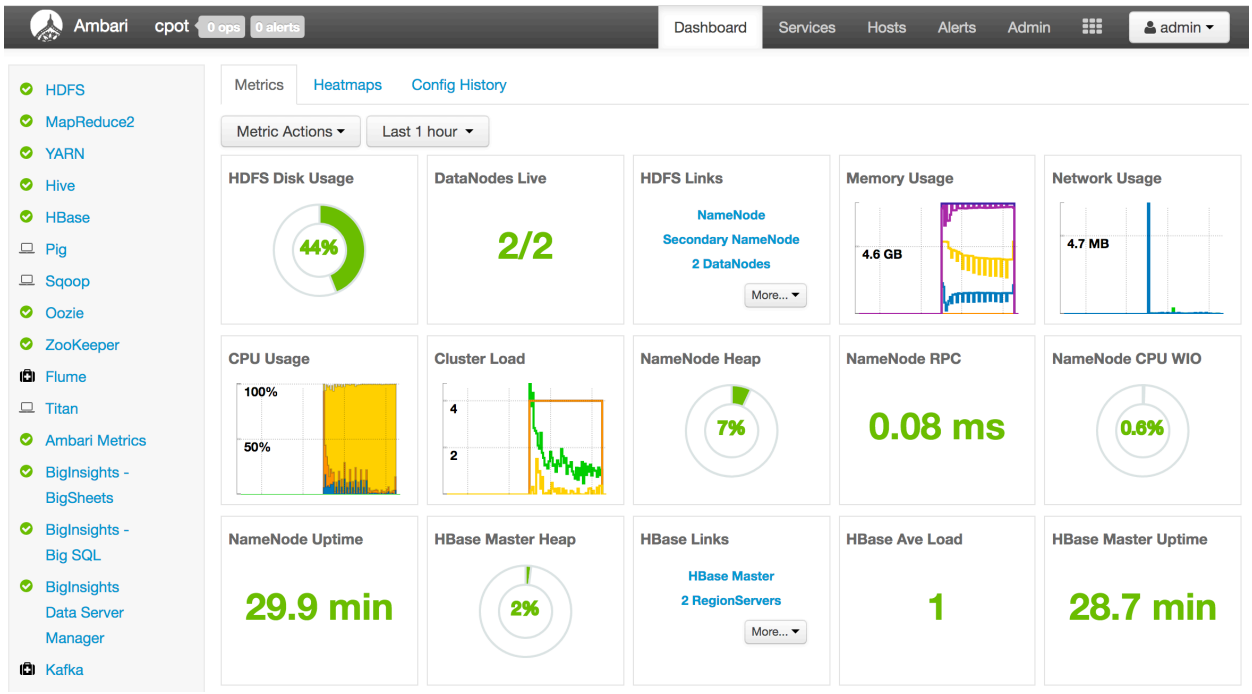
[Allow **30 minutes** to complete this lab.]

For additional information about DSM capabilities, please visit IBM's DSM site at http://www-03.ibm.com/software/products/en/ibm-data-server-manager. If you have questions or comments about this lab, please post them to the forum on Hadoop Dev at https://developer.ibm.com/answers?community=hadoop.

3.1. Launching the BigInsights Home and Big SQL web tooling

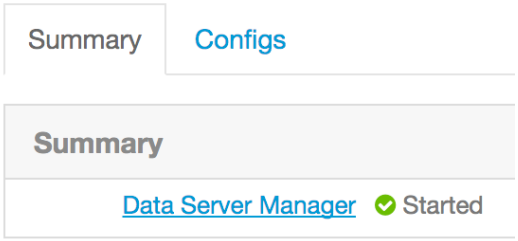
Big SQL web tooling is accessed through a link in the BigInsights Home page. BigInsights Home is a component provided. In this exercise, you'll verify that BigInsights Home and Big SQL web tooling (DSM) are installed and running on your cluster. Then you'll launch the Home page and the web tooling.

__1. Launch Ambari and sign into its console. If necessary, consult an earlier lab for details on how to do this.

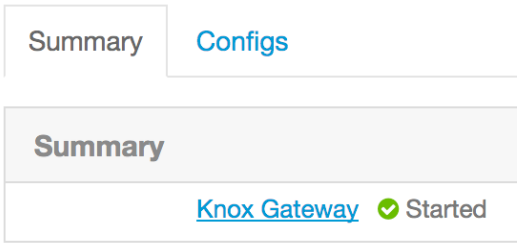


__2. From the Ambari Dashboard, inspect the list of services in the left pane. Verify that BigInsights Home and BigInsights – Big SQL services are running, as well as all pre-requisite services (e.g., HDFS, MapReduce2, Hive, and Knox).

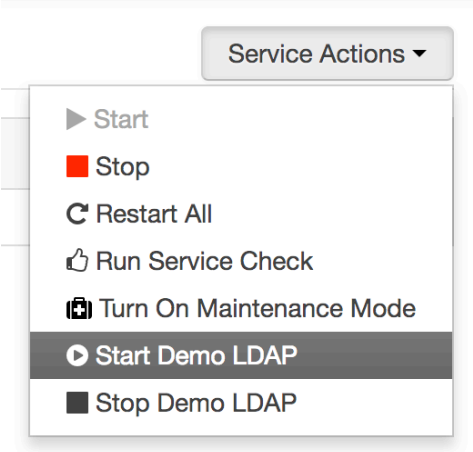
__3. Click on the BigInsights Data Server Manager. Verify that all underlying components are running.



__4. Click on the Knox. Verify that all underlying components are running.



__5. On Knox service view. Click on Service Actions and Start Demo LDAP.

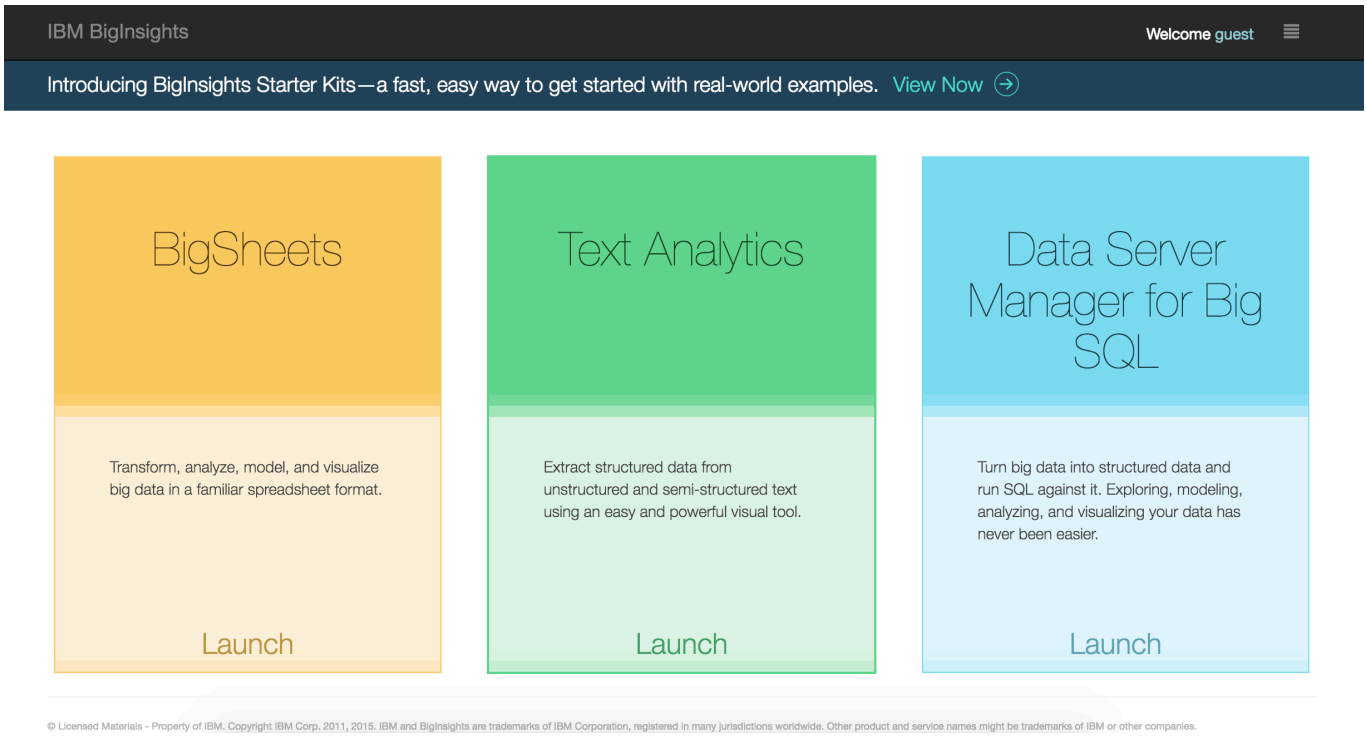


__6. Launch BigInsights Home, providing the appropriate URL based on your installation's configuration. Assuming you installed BigInsights with Knox and accepted default installation values, the BigInsights Home URL is similar to the link shown below. **Substitute the location of the Knox gateway on your cluster for the italicized text in this example**.

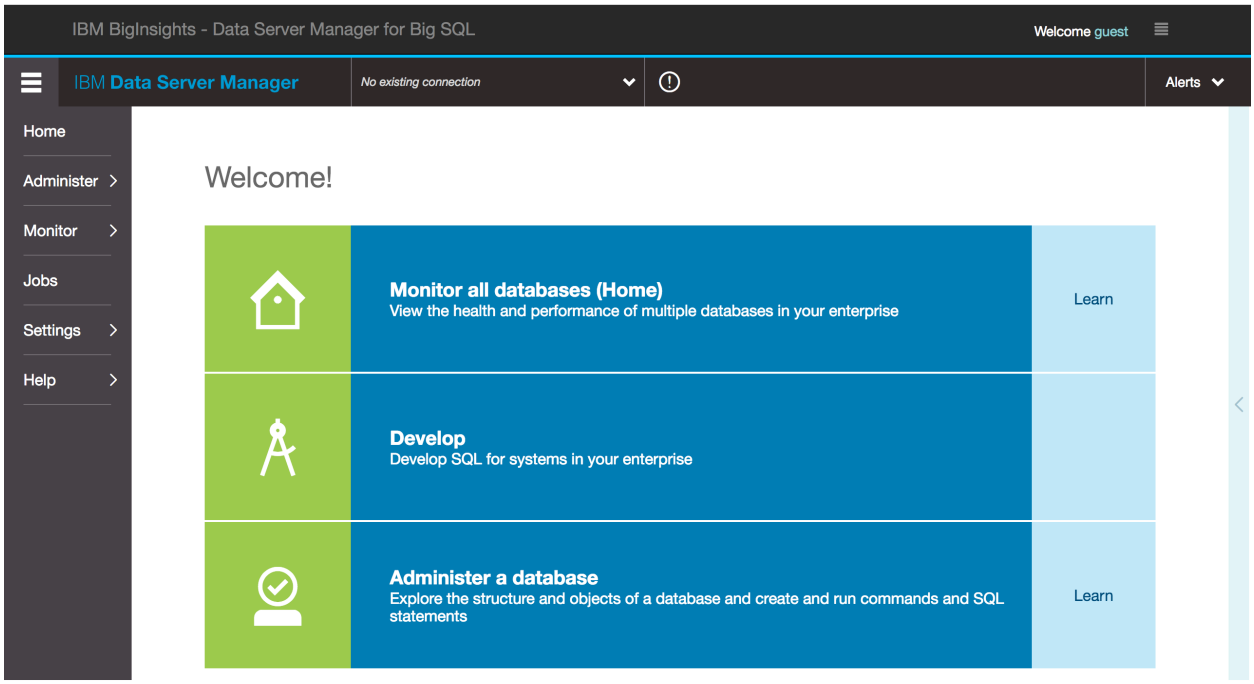
`https://bigi01.localdomain:8443/gateway/default/BigInsightsWeb/index.html`

__7. When prompted, enter a valid user ID and password for the Knox gateway. (Defaults are guest / guest-password).

__8. Verify that BigInsights Home displays an item for Data Server Manager (DSM). Depending on the size of your browser window and other BigInsights components installed on your cluster, you may need to scroll through the BigInsights Home page to locate the DSM section.



__9. Click the Launch button in the Data Server Manager for Big SQL box.



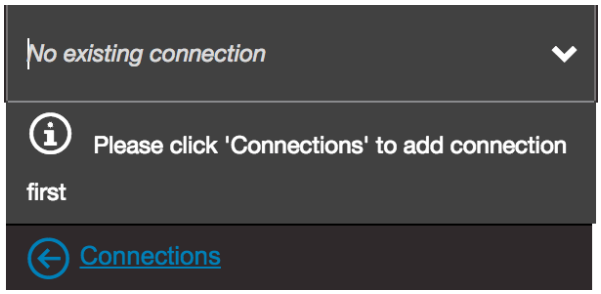
3.2. Issuing queries and inspecting results

In this exercise, you will work with the SQL Editor to execute a query and inspect the results.

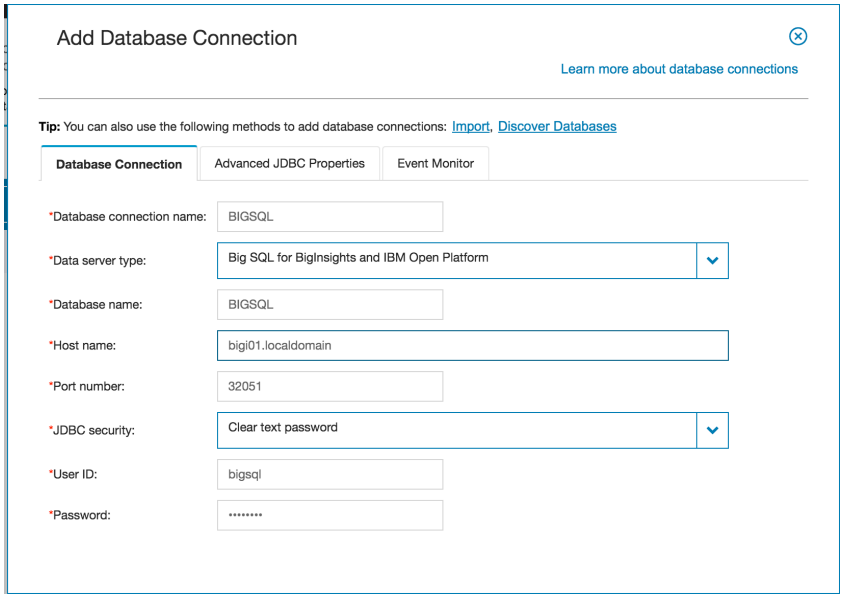
With the Big SQL web tooling launched, click the SQL Editor link at left.

__1. Add Database Connection.

Click on Connections:



Add Connection:



Click on Test Connection:

Click on OK

__2. Upload the files into the HDFS File System via shell:

```
su bgsql
```

Check that you have create /user/bigsql_lab directory:

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

If other case, create the directory as hdfs session and give permissions:

```
hdfs dfs -mkdir /user/bigsql_lab
```

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

__3. Give the permissions to the folder **/user/bigsql_lab/external**. For doing that, go to your terminal and type the following commands as root session:

```
su – hdfs
```

```
hdfs dfs -mkdir /user/bigsql_lab/external

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

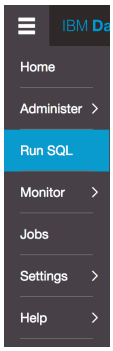
__4. Upload files:

```
hdfs dfs -copyFromLocal /home/bigsql/data.csv /user/bigsql_lab

hdfs dfs -copyFromLocal /home/bigsql/external.txt /user/bigsql_lab/external
```

__5. Back to the DSM on your browser, in the empty query box to the left of the database connection information, paste the following query:

Select BIGSQL connection:

Click on Run SQL:

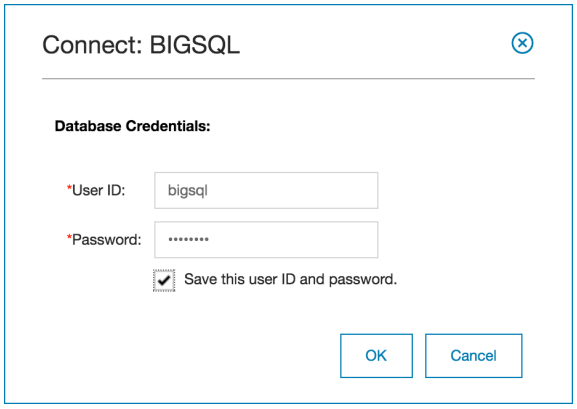


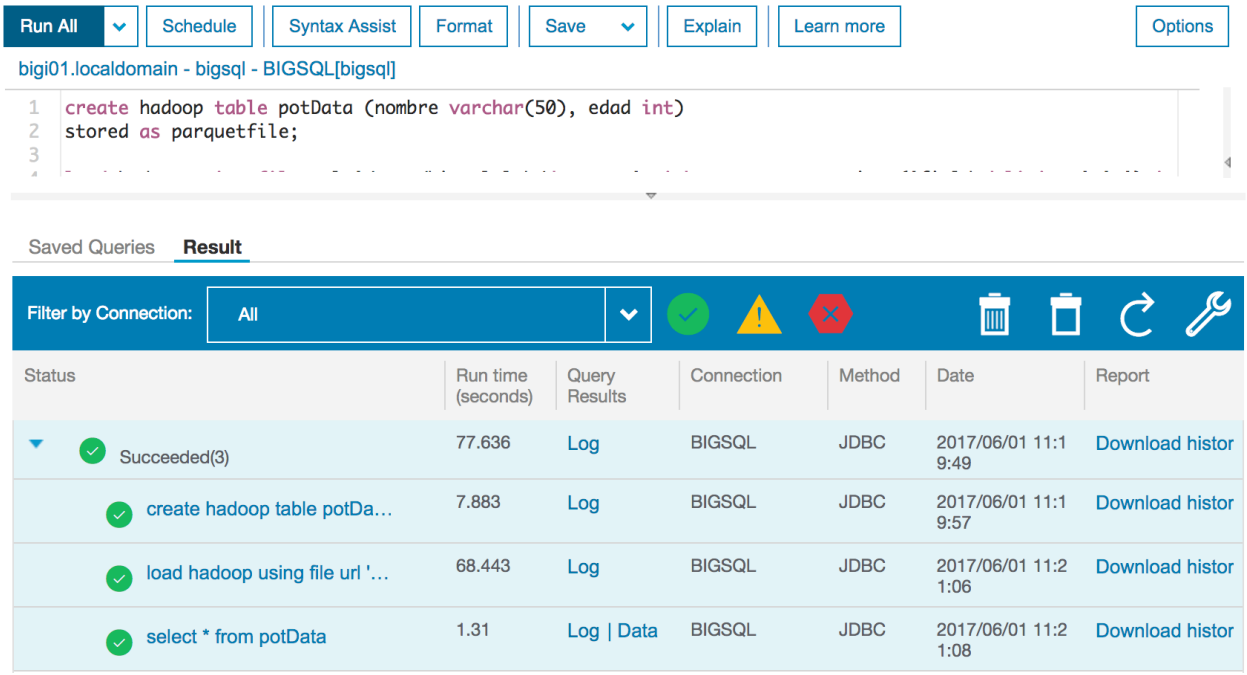
Paste queries:

```
create hadoop table potData (nombre varchar(50), edad int)
stored as parquetfile;

load hadoop using file url '/user/bigsql_lab/data.csv' with source properties
('field.delimiter'=',') into table potData overwrite with load properties
('rejected.records.dir'='/tmp/rejected_records/AUX','max.rejected.records'=1,
'num.map.tasks'=30);

select * from potData;
```

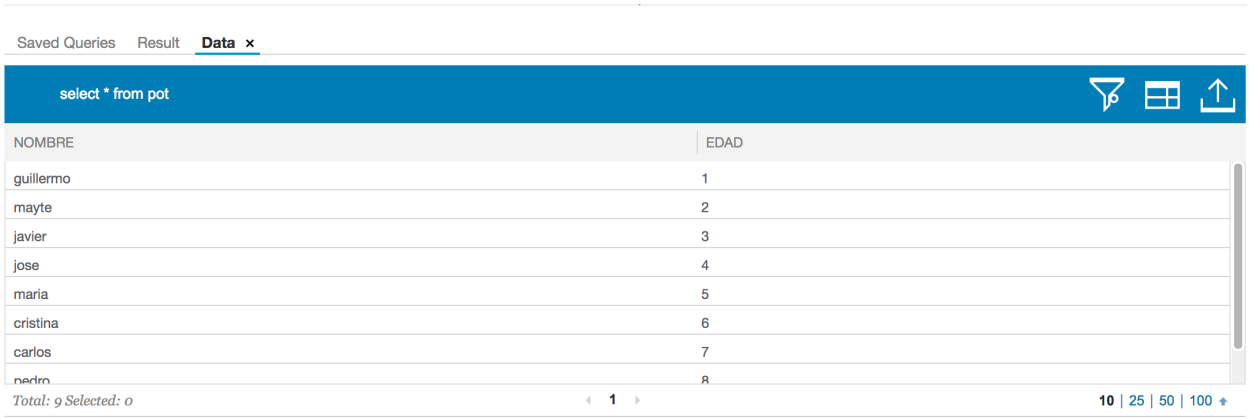
__6. Click Run All. Status information about the operation is shown in the lower panel.

If prompt this message, complete it:

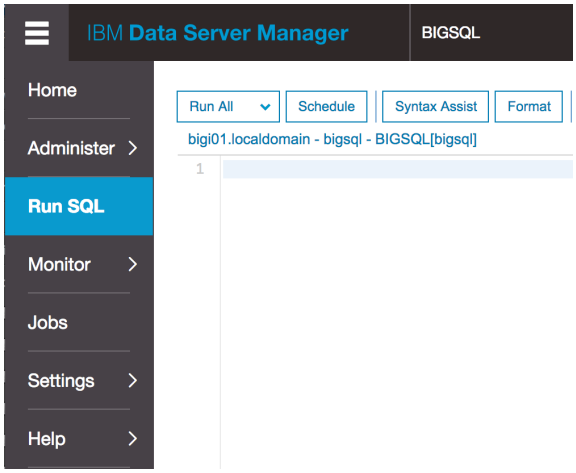




__7. When the operation completes, click on Data link latest query and adjust the size of the lower pane (if needed) and inspect the query results, a subset of which is shown below.



__8. Clear Query Box.



__9. Run queries and review data results:

From inserts to parquet file:

```
create hadoop table potInsert (nombre varchar(50), edad int)
stored as parquetfile;

insert into potInsert values ('paco', 10),( 'arancha', 20),( 'pedro', 30),(
'nacho', 15);

select * from potInsert;
```

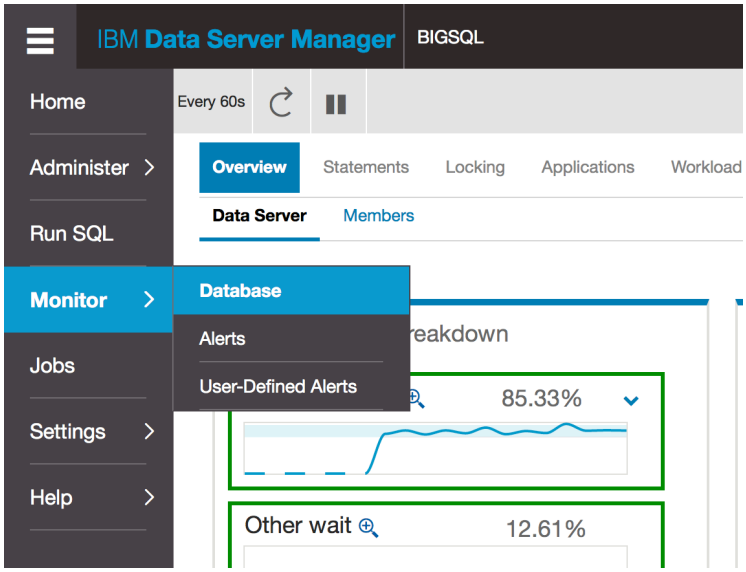
From file:

```
create external hadoop table potExternal (nombre varchar(50), edad int)
location '/user/bigsql_lab/external'
row format delimited fields terminated by '|';

select * from potExternal;
```

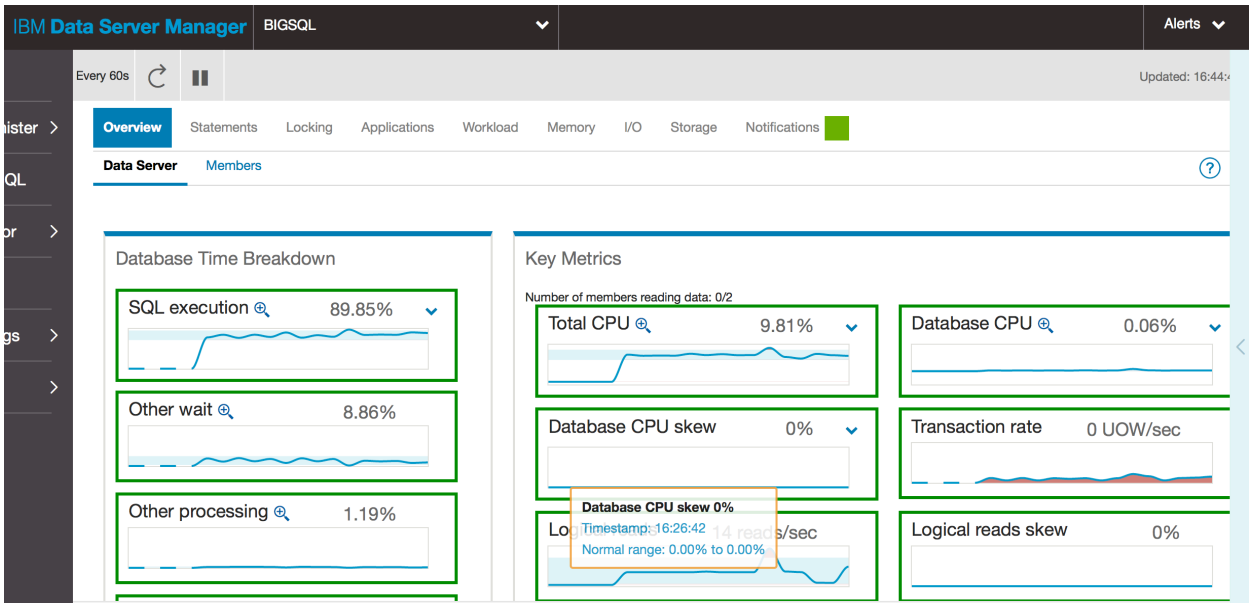
3.3. Examining database metrics

Administrators can use the Big SQL web tooling to display metrics about their database and inspect the overall health of their environment. This exercise shows you how to launch this facility and begin exploring some metrics. For further details, consult the product Knowledge Center or online help information.

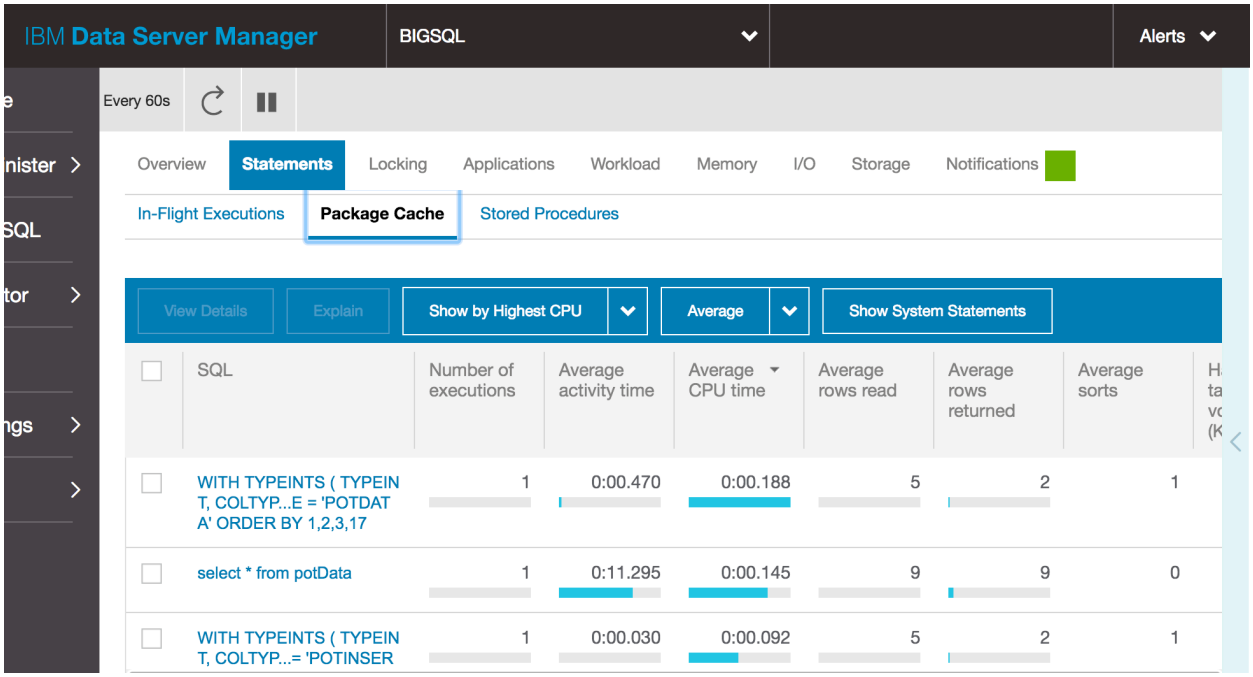
__1. With the Big SQL web tooling launched and your database connection active (based on your completion of the previous lab exercise), click in Monitor > Database tab.



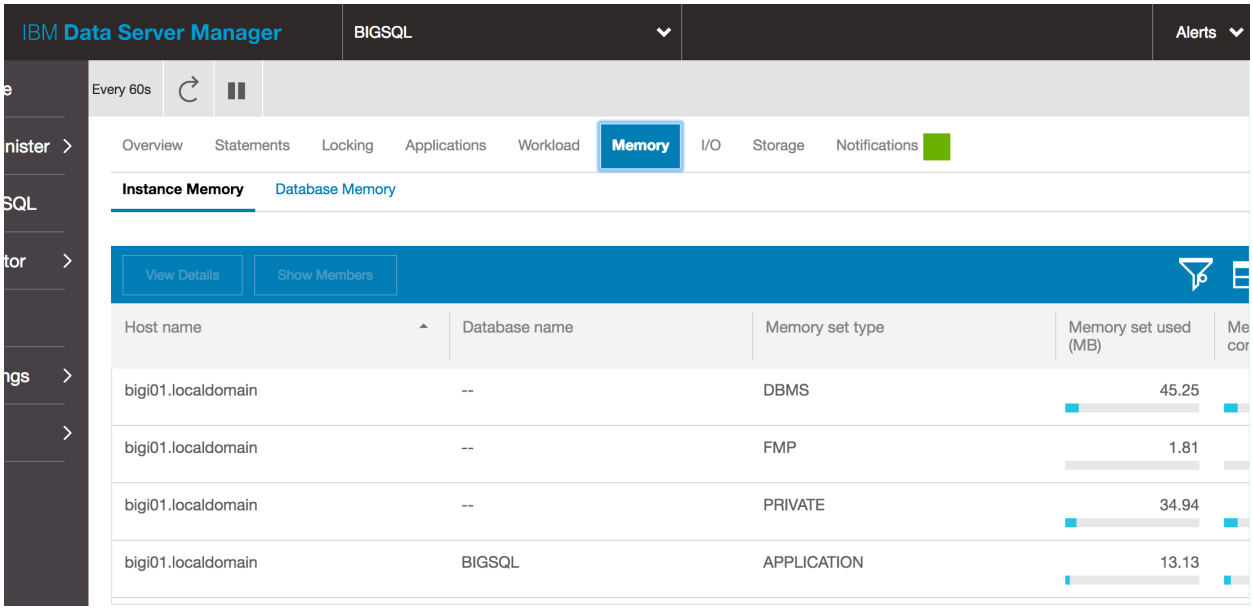
__2. Inspect the Overview information presented. (If necessary, click on the Overview tab.)



__3. Examine details about SQL execution and other aspects like the history in Statements tab.



__4. Investigate the Memory:



3.4. Compatibility with HIVE

Just to show the compatibility with HIVE, take your terminal and type the following commands, as root session:

```
> su - hive
```

Execute hive client driver

```
> hive
```

Now, you can check that the previous databases, tables and data are available by HIVE:

```
hive> show databases;
hive> use bigsql;
hive> show tables;
hive> describe potdata;
hive> select * from potdata;
```

Lab 4 Understanding and influencing data access plans

As you may already know, query optimization significantly influences runtime performance. In this lab, you'll learn how to use DSM and the ANALYZE TABLE command to collect statistics about your data so that the Big SQL query optimizer can make well-informed decisions when choosing between various options for data access. Collecting and maintaining accurate statistics is highly recommended when dealing with large volumes of data (but less critical for this sample lab).

Next, you will use the Big SQL EXPLAIN feature to examine the data access plan the query optimizer chose for a given query. Performance specialists often consult data access plans to help them tune their environments.

After completing this lab, you will know how to:

- Collect meta data (statistics) about your data.
- Collect and review data access plans for your queries.

Prior to beginning this lab, you must have created and populated several tables with data as described in an earlier lab on Querying Structured Data.

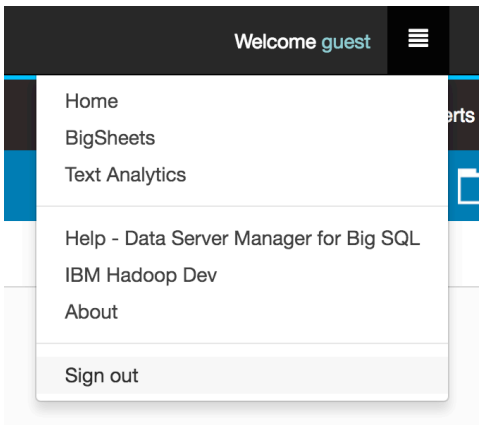
[Allow **30** minutes to complete this lab.]

Please post questions or comments about this lab to the forum on Hadoop Dev at https://developer.ibm.com/answers?community=hadoop.

4.1. DSM

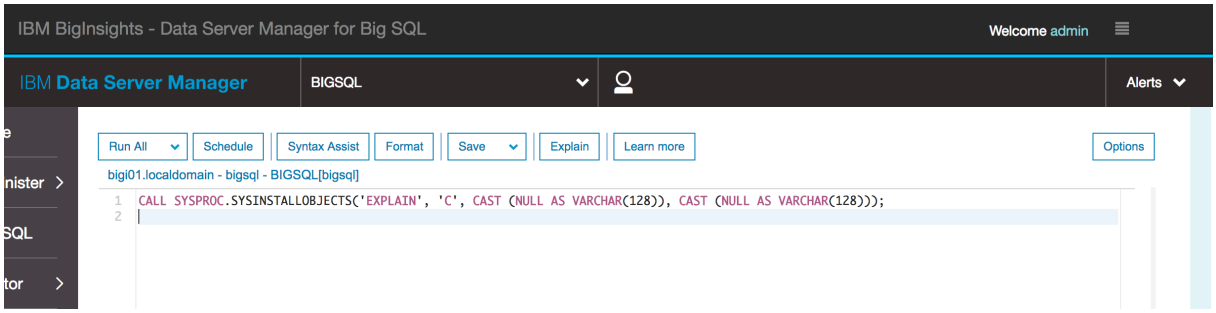
__1. To create the necessary EXPLAIN tables to hold information about your query plans, call the SYSINSTALLOBJECTS procedure as admin user. In this invocation, the tables will be created only for your user account. By casting a NULL in the last parameter, a single set of EXPLAIN tables can be created in schema SYSTOOLS, which can be used for all users.

Login on DSM as admin/passw0rd. If you followed previous Lab, you need to logout:



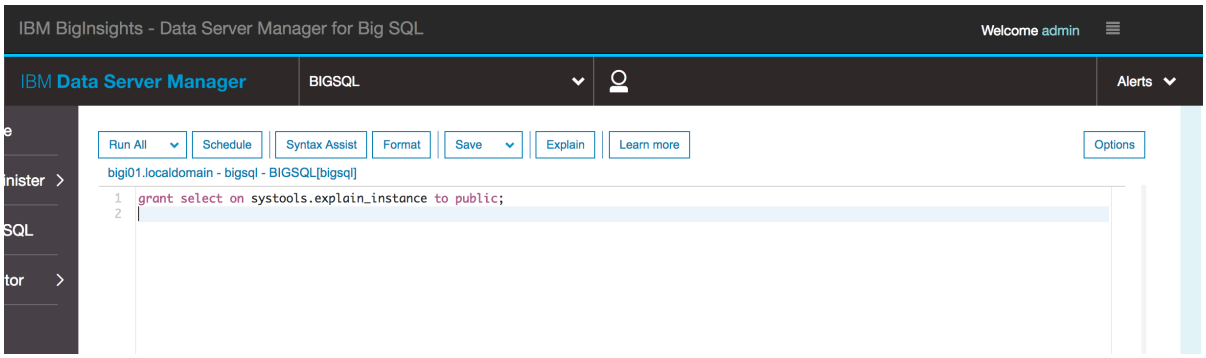
As admin user, run the query:

```
CALL SYSPROC.SYSINSTALLOBJECTS('EXPLAIN', 'C', CAST (NULL AS VARCHAR(128)), CAST (NULL
AS VARCHAR(128)));
```



__2. Authorize all Big SQL users to read data from the SYSTOOLS.EXPLAIN_INSTANCE table created by the stored procedure you just executed.

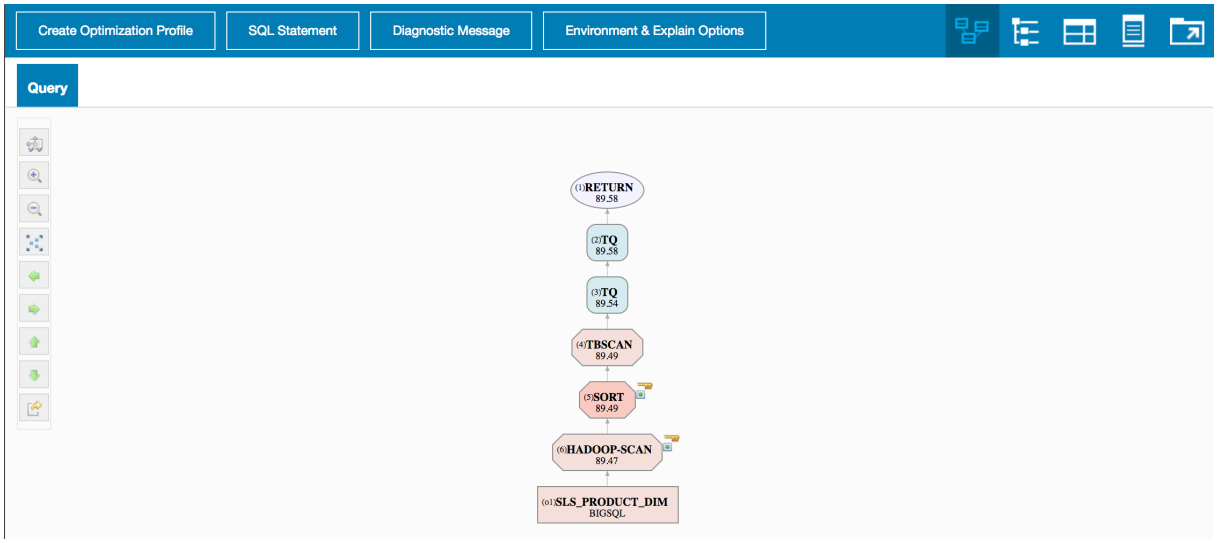
```
grant select on systools.explain_instance to public;
```



__3. Login on DSM as guest/passw0rd.

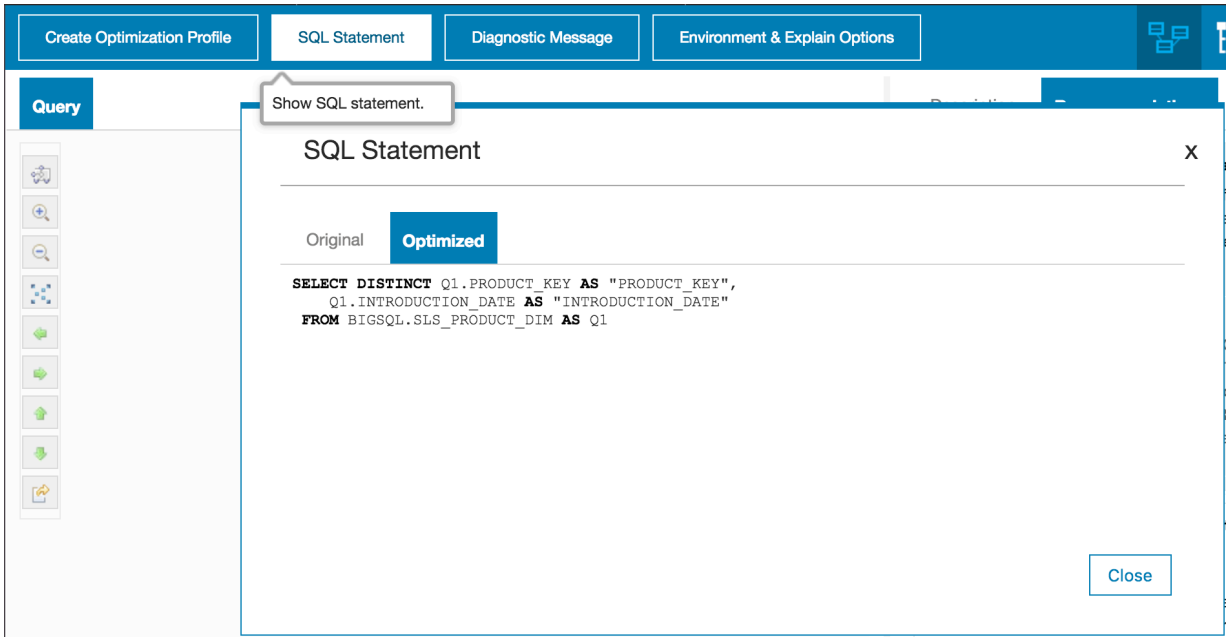
__4. Click on Run SQL, paste the query and click on Explain

```
select distinct product_key, introduction_date from sls_product_dim;
```

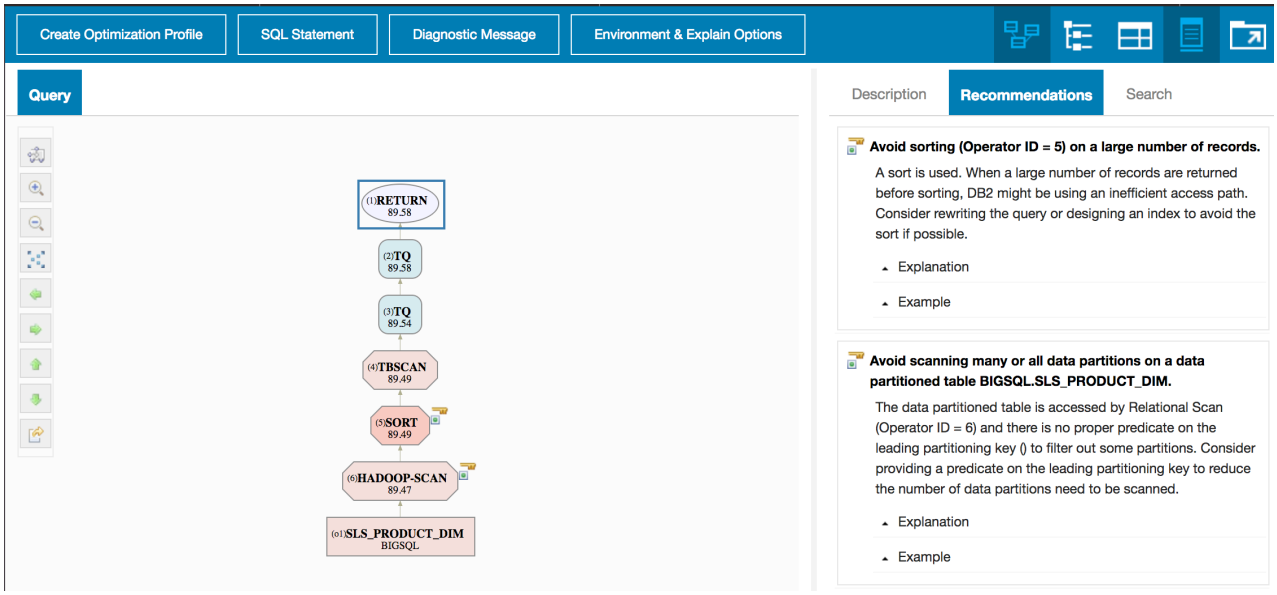


__5. Inspect the Original Statement and Optimized Statement sections of the plan. Sometimes, the optimizer will decide to rewrite the query in a more efficient manner-- for example, replacing IN lists with JOINS. In this case, the Optimized Statement show that no further optimization has been done.

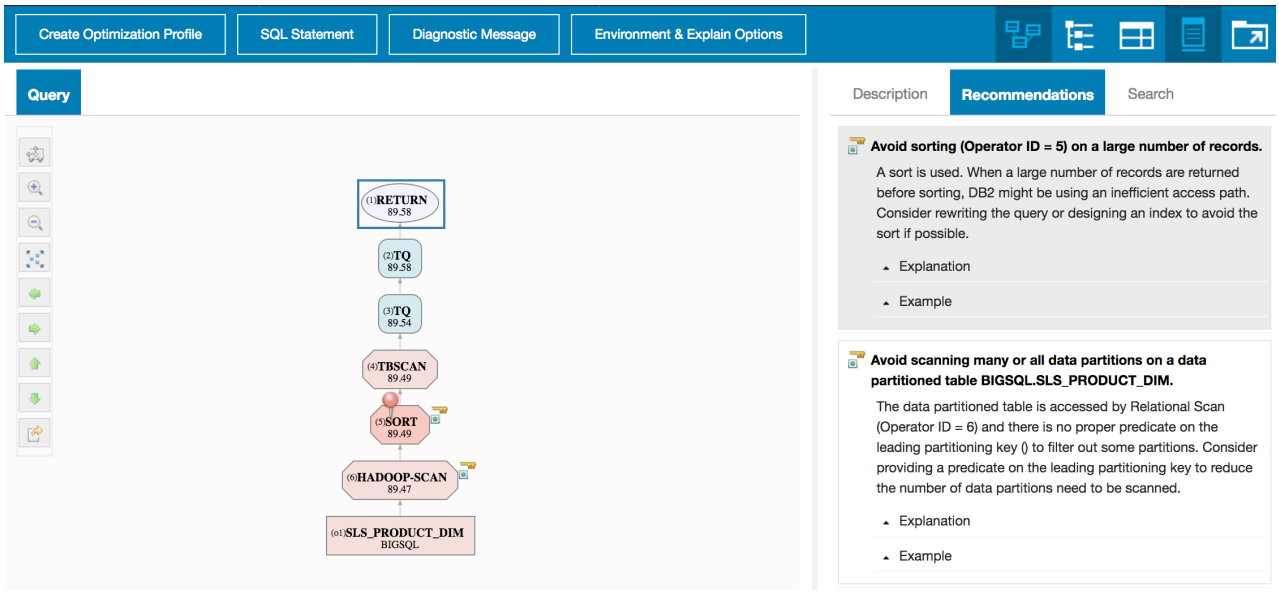
Click on SQL Statement link and review Original VS Optimized query.



__6. Now, if you click on any node and expand right frame, you will see a Recommendations tab, click on it and review first recommendation.

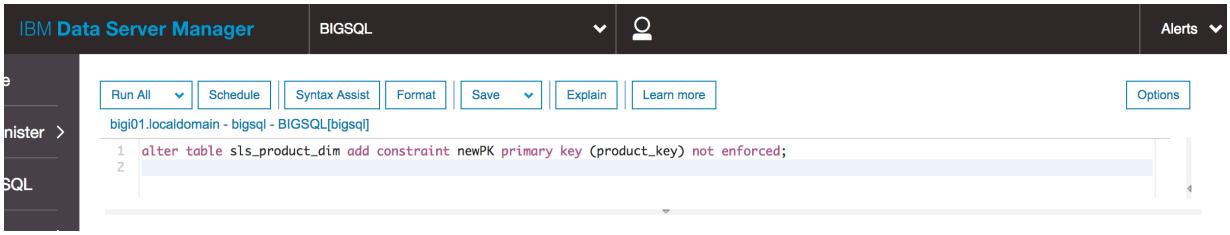


__7. If you click on the recommendation, you will sea a pointer on the Sort node.



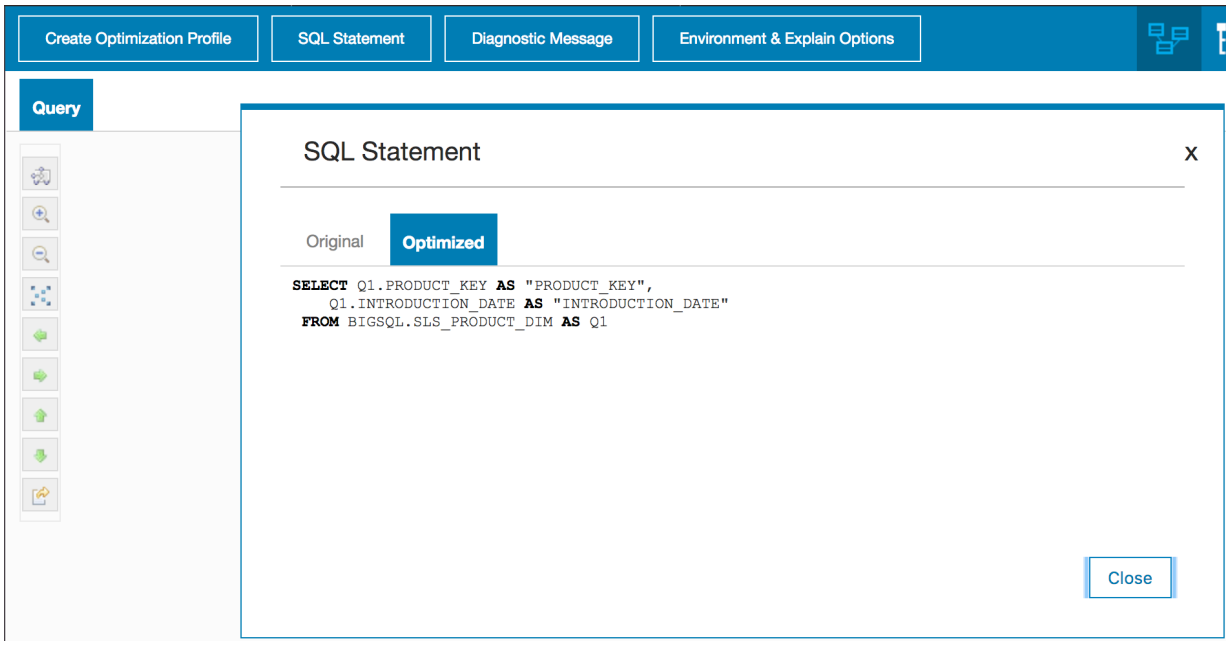
__8. Next, for BigSQL an index is a constraint so alter the table to include an informational primary key constraint on one of the table's columns. From your query execution environment, execute the following alter command:

```
alter table sls_product_dim add constraint newPK primary key (product_key)
not enforced;
```

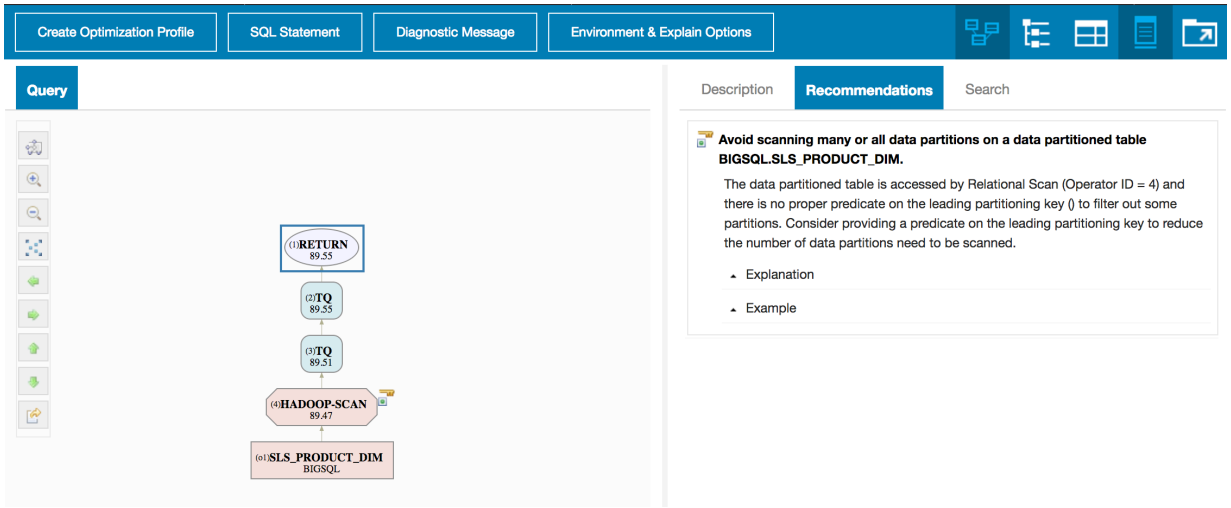


This will alter the table to have a non-enforced PK constraint.

__9. Run previous select query and compare the original and optimized statements. Observe that the DISTINCT clause was removed from the optimized query. Because of the primary key constraint that you added, the optimizer determined that the DISTINCT clause (to eliminate duplicates) was unnecessary.



__10. Similarly, inspect the new Access Plan. Observe that no SORT operation is included and that there are fewer operations in total and the followed recomentation is not in Recommendations tab.



4.2. [Optional] JSqsh (Review Lab 5)

4.2.1. Collecting statistics with the ANALYZE TABLE command

The ANALYZE TABLE command collects statistics about your Big SQL tables. These statistics influence query optimization, enabling the Big SQL query engine to select an efficient data access path to satisfy your query.

__11. If needed, launch JSqsh and connect to your Big SQL database.

__12. Collect statistics for the tables you created in an earlier lab on Querying Structured Data. **By default, BigSQL 4.2 performs a predefined analyze over a prefixed sample, if you want to run over the full data, issue each of the following commands individually, allowing the operation to complete.** Depending on your machine resources, this may take several minutes or more.

```
ANALYZE TABLE sls_sales_fact COMPUTE STATISTICS
FOR COLUMNS product_key, order_method_key;

ANALYZE TABLE sls_product_dim COMPUTE STATISTICS
FOR COLUMNS product_key, product_number, product_line_code,
product_brand_code;

ANALYZE TABLE sls_product_lookup COMPUTE STATISTICS
FOR COLUMNS product_number, product_language;

ANALYZE TABLE sls_order_method_dim COMPUTE STATISTICS
FOR COLUMNS order_method_key, order_method_en;

ANALYZE TABLE sls_product_line_lookup COMPUTE STATISTICS
FOR COLUMNS product_line_code, product_line_en;
```

```
ANALYZE TABLE sls_product_brand_lookup COMPUTE STATISTICS
FOR COLUMNS product_brand_code;
```

ANALYZE TABLE syntax

It's best to include FOR COLUMNS and a list of columns to the ANALYZE TABLE command. Choose those columns found in your WHERE, ORDER BY, GROUP BY and DISTINCT clauses.

4.2.2. Understanding your data access plan (EXPLAIN)

The EXPLAIN feature enables you to inspect the data access plan selected by the Big SQL optimizer for your query. Such information is highly useful for performance tuning. This exercise introduces you to EXPLAIN, a Big SQL feature that stores meta data in a set of EXPLAIN tables.

__1. If necessary, launch your query execution environment and connect to your Big SQL database.

__2. To create the necessary EXPLAIN tables to hold information about your query plans, call the SYSINSTALLOBJECTS procedure. In this invocation, the tables will be created only for your user account. By casting a NULL in the last parameter, a single set of EXPLAIN tables can be created in schema SYSTOOLS, which can be used for all users.

```
CALL SYSPROC.SYSINSTALLOBJECTS('EXPLAIN', 'C', CAST (NULL AS VARCHAR(128)), CAST (NULL
AS VARCHAR(128)));
```

__3. Authorize all Big SQL users to read data from the SYSTOOLS.EXPLAIN_INSTANCE table created by the stored procedure you just executed.

```
grant select on systools.explain_instance to public;
```

__4. Capture the data access plan for a query. One way to do this is by prefixing the query with the command `EXPLAIN PLAN WITH SNAPSHOT FOR`. In this way, the query isn't executed, but the access plan is saved in the EXPLAIN tables. Run this command:

```
explain plan with snapshot for
select distinct product_key, introduction_date
from sls_product_dim;
```

Information about the data access strategy for this query is stored in the EXPLAIN tables, which you'll explore shortly. There are various tools to view the "explained" access plan. For example, you could use the Data Studio, IBM Query Tuning perspective and Query Tuner Project. In this lab, you will use a DB2 utility called `db2exfmt`, executed from the bash shell.

__5. If necessary, open a terminal window.

__6. Invoke `db2profile` to set up your environment.

```
. ~bigsql/sqllib/db2profile
```

__7. Invoke `db2exfmt` to retrieve the plan. Supply appropriate input parameters, including the name of your Big SQL database (e.g., `-d bigsql`), the user ID and password under which you executed the explain plan query (e.g., `-u bigsql passw0rd`), and an output file for the plan itself (e.g., `-o query1.exp`). Adjust the example below as needed for your environment.

```
db2exfmt -d bigsql -u bigsql passw0rd -o query1.exp
```

__8. When prompted for additional information (such as an EXPLAIN timestamp), accept defaults and hit Enter. Allow the operation to complete.

```
[bigsql@bigi01 ~]$ db2exfmt -d bigsql -u bigsql passw0rd -o query1.exp
DB2 Universal Database Version 11.1, 5622-044 (c) Copyright IBM Corp. 1991, 2015
Licensed Material - Program Property of IBM
IBM DATABASE 2 Explain Table Format Tool

Connecting to the Database as user bigsql.
Connect to Database Successful.
Using SYSTOOLS schema for Explain tables.
Enter up to 26 character Explain timestamp (Default -1) ==>
Enter up to 128 character source name (SOURCE_NAME, Default %%) ==>
Enter source schema (SOURCE_SCHEMA, Default %%) ==>
Enter section number (0 for all, Default 0) ==>
Output is in query1.exp.
Executing Connect Reset -- Connect Reset was Successful.
```

__9. Investigate the contents of the query1.exp file. For example, type

```
more query1.exp
```

Use the space bar to scroll forward through the output one page at a time, and enter b to page backward.

__10. Inspect the Original Statement and Optimized Statement sections of the plan. Sometimes, the optimizer will decide to rewrite the query in a more efficient manner-- for example, replacing IN lists with JOINS. In this case, the Optimized Statement show that no further optimization has been done.

```
Original Statement:
-------------------

select distinct product_key, introduction_date
from sls_product_dim

Optimized Statement:
--------------------
SELECT
  DISTINCT Q1.PRODUCT_KEY AS "PRODUCT_KEY",
  Q1.INTRODUCTION_DATE AS "INTRODUCTION_DATE"
FROM
  BIGSQL.SLS_PRODUCT_DIM AS Q1
```

__11. Scroll to the `Access Plan` section. Notice the SORT operation and the total number of operations for this plan.

```
Access Plan:
-----------
        Total Cost:             89.9366
        Query Degree:           4

       Rows
      RETURN
      (   1)
       Cost
        I/O
        |
        41
      MDTQ
      (   2)
      89.9366
         1
        |
      35.6008
      LMTQ
      (   3)
      89.9026
         1
        |
      35.6008
      TBSCAN
      (   4)
      89.8737
         1
        |
      35.6008
      SORT
      (   5)
      89.8736
         1
        |
      68.3333
```

__12. Next, alter the table to include an informational primary key constraint on one of the table's columns. From your query execution environment (e.g., JSqsh), execute the following alter command:

```
alter table sls_product_dim add constraint newPK primary key (product_key)
not enforced;
```

This will alter the table to have a non-enforced PK constraint.

__13. Now collect the plan information for the same query on the altered table:

```
explain plan with snapshot for
select distinct product_key, introduction_date
from sls_product_dim;
```

__14. From a terminal window, invoke `db2exfmt` again, providing a different output file name (such as `query2.exp`):

```
db2exfmt -d bigsql -u bigsql passw0rd -o query2.exp
```

__15. When prompted for additional information (such as an EXPLAIN timestamp), accept defaults and hit Enter. Allow the operation to continue.

```
DB2 Universal Database Version 11.1, 5622-044 (c) Copyright IBM Corp. 1991, 2015
Licensed Material - Program Property of IBM
IBM DATABASE 2 Explain Table Format Tool

Connecting to the Database as user bigsql.
Connect to Database Successful.
Using SYSTOOLS schema for Explain tables.
Enter up to 26 character Explain timestamp (Default -1) ==>
Enter up to 128 character source name (SOURCE_NAME, Default %%) ==>
Enter source schema (SOURCE_SCHEMA, Default %%) ==>
Enter section number (0 for all, Default 0) ==>
Output is in query2.exp.
Executing Connect Reset -- Connect Reset was Successful.
```

__16. Investigate the contents of the query2.exp file. For example, type

```
more query2.exp
```

__17. Compare the original and optimized statements. Observe that the DISTINCT clause was removed from the optimized query. Because of the primary key constraint that you added, the optimizer determined that the DISTINCT clause (to eliminate duplicates) was unnecessary.

```
Original Statement:
------------------

select distinct product_key, introduction_date
from sls_product_dim

Optimized Statement:
-------------------
SELECT
  Q1.PRODUCT_KEY AS "PRODUCT_KEY",
  Q1.INTRODUCTION_DATE AS "INTRODUCTION_DATE"
FROM
  BIGSQL.SLS_PRODUCT_DIM AS Q1
```

__18. Similarly, inspect the new Access Plan. Observe that no SORT operation is included and that there are fewer operations in total.

```
Access Plan:
-----------
        Total Cost:             89.9367
        Query Degree:           4

       Rows
      RETURN
      (   1)
       Cost
        I/O
        |
        205
      DTQ
      (   2)
      89.9367
         1
        |
      68.3333
      LTQ
      (   3)
      89.8986
         1
        |
      68.3333
      TBSCAN
      (   4)
      89.8684
         1
        |
```

Lab 5 [Optional] Using the Big SQL command line interface (JSqsh)

BigInsights supports a command-line interface for Big SQL through the Java SQL Shell (JSqsh, pronounced "jay-skwish"). JSqsh is an open source project for querying JDBC databases. In this section, you will learn how to

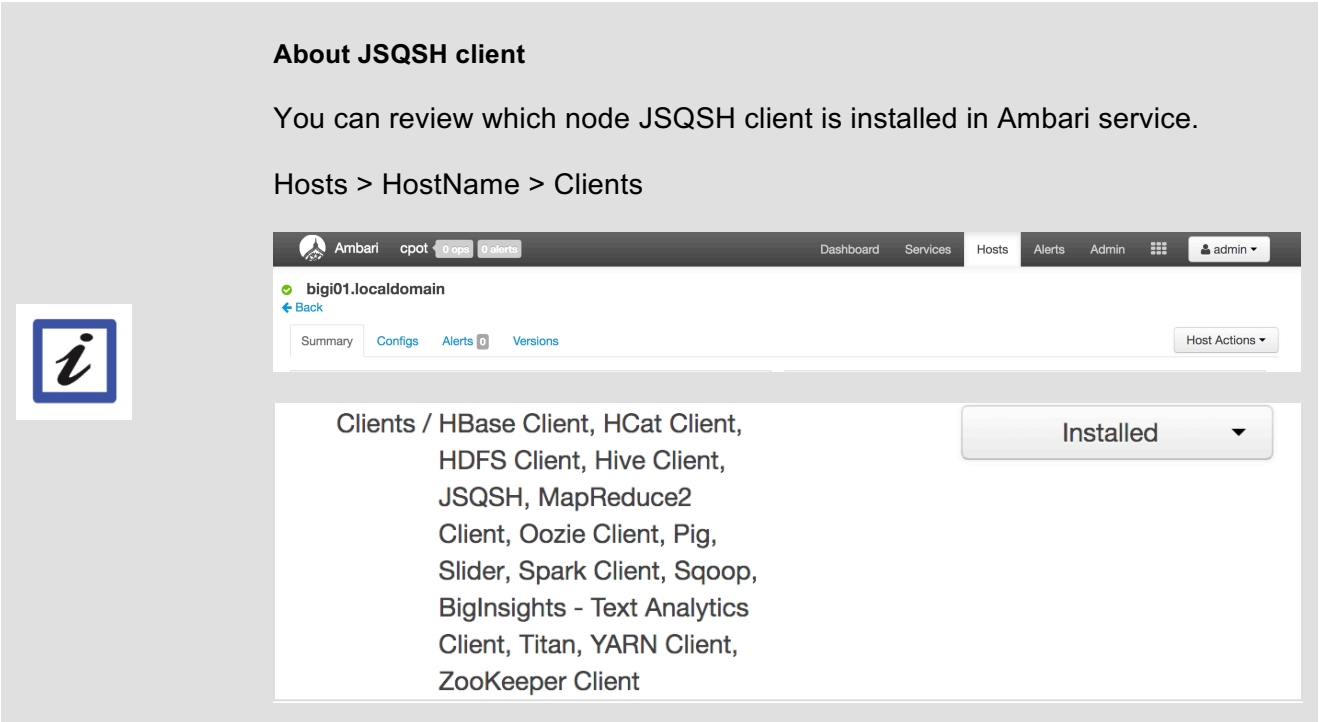
- We are going to use bigsql user. Login such as this user or do a su - bigsql
- Launch JSqsh.
- Issue Big SQL queries.
- Issue popular JSqsh commands to get help, retrieve your query history, and perform other functions.

[Allow **30 minutes** to complete this section].

5.1. Understanding JSqsh connections

To issue Big SQL commands from JSqsh, you need to define a connection to a Big SQL server.

__1. On a node in your cluster in which JSqsh has been installed, open a terminal window.



__2. Login as bigsql user

```
su bigsql
```

__3. Launch the JSqsh shell. If JSqsh was installed at its default location of /usr/ibmpacks/common-utils/current/jsqsh/bin, you would invoke the shell with this command:

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

__4. If this is your first time invoking the shell, a welcome screen will display. A subset of that screen is shown below.

```
Welcome to JSqsh 4.8
Type "\help" for help topics. Using JLine.
WELCOME TO JSQSH!

    It looks like this is the first time that you've run jsqsh (or
    you just typed '\help welcome' at the jsqsh prompt). If this is
    the first time you have run jsqsh, you will find that you have
    a shiney new directory called '.jsqsh' in your home directory
    and that this directory contains a couple of files that you
    should be aware of:

    drivers.xml - JSqsh comes pre-defined to understand how to use
       a fixed number of JDBC drivers, however this file may be
       used to teach it how to recognize other JDBC drivers. The
       file is pretty well commented, so hopefully it is enough
       to get you started.

    sqshrc - Everything contained in this file is executed just
       as if you had typed commands at the sqsh prompt and is
       the place where you can set variables and configure sqsh
       to your likings.

    JSqsh is intended to be self-documenting. If you would like
    to see this information again, then type '\help welcome' at
    the jsqsh prompt, or run '\help' to see a list of all help
    topics that are available.

You will now enter the jsqsh setup wizard.
Hit enter to continue:
```

__5. If prompted to press **Enter** to continue, do so.

__6. When prompted, If following screen appears, enter **C** to launch the connection wizard.

```
JSQSH SETUP WIZARD

Welcome to the jsqsh setup wizard! This wizard provides a (crude) menu
driven interface for managing several jsqsh configuration files. These
files are all located in $HOME/.jsqsh, and the name of the file being
edited by a given screen will be indicated on the title of the screen

Note that many wizard screens require a relative large console screen
size, so you may want to resize your screen now.

(C)onnection management wizard
   The connection management wizard allows you to define named connections
   using any JDBC driver that jsqsh recognizes. Once defined, jsqsh only
   needs the connection name in order to establish a JDBC connection

(D)river management wizard
   The driver management wizard allows you to introduce new JDBC drivers
   to jsqsh, or to edit the definition of an existing driver. The most
   common activity here is to provide the classpath for a given JDBC driver

Choose (Q)uit, (C)onnection wizard, or (D)river wizard: C
```

In the future, you can enter `\setup connections` in the JSqsh shell to invoke this wizard.

Inspect the list of drivers displayed by the wizard, and note the number for the `db2`

```
JSQSH CONNECTION WIZARD - (edits $HOME/.jsqsh/connections.xml
The following connections are currently defined:

     Name                 Driver     Host                            Port
---  -------------------- ---------- ------------------------------ ------
  1  bigsql               db2        bigi01.localdomain              32051

Enter a connection number above to edit the connection, or:
(B)ack, (Q)uit, or (A)dd connection: 1
```

About the driver selection

You may be wondering why this lab uses the DB2 driver rather than the Big SQL driver. In 2014, IBM released a common SQL query engine as part of its DB2 and BigInsights offerings. Doing so provided for greater SQL commonality across its relational DBMS and Hadoop-based offerings. It also brought a greater breadth of SQL function to Hadoop (BigInsights) users. This common query engine is accessible through the "DB2" driver listed. The Big SQL driver remains operational and offers connectivity to an earlier, BigInsights-specific SQL query engine. This lab focuses on using the common SQL query engine.

__7. At the prompt line, enter the number of the DB2 driver.

__8. The connection wizard displays some default values for the connection properties and prompts you to change them. (Your default values may differ from those shown below.)

```
JSQSH CONNECTION WIZARD - (edits $HOME/.jsqsh/connections.xml)

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 : 32051
3             server : bigi01.localdomain
4               user : bigsql
5           password :
6       Autoconnect : false

JDBC Driver Properties
----------------------
None

Enter a number to change a given configuration property, or
(T)est, (D)elete, (B)ack, (Q)uit, Add (P)roperty, or (S)ave:
```

__9. Change each variable as needed, one at a time. To do so, enter the variable number and specify a new value when prompted. For example, to change the value of the `password` variable (which is null by default)

(1) Specify variable number 5 and hit **Enter**.

(2) Enter the password value and hit **Enter.**

```
Enter a number to change a given configuration property, or
(T)est, (D)elete, (B)ack, (Q)uit, Add (P)roperty, or (S)ave: 5

Please enter a new value:
password: ********
```

(3) Inspect the variable settings that are displayed again to verify your change.

Repeat this process as needed for each variable that needs to be changed.

After making all necessary changes, the variables should reflect values that are accurate for your environment. In particular, the server property must correspond to the location of the Big SQL Head Node in your cluster.

Here is an example of a connection created for the `bigsql` user account (password `passw0rd`) that will connect to the database named `bigsql` at port `51000` on the bigi01.localdomain server.

```
JSQSH CONNECTION WIZARD - (edits $HOME/.jsqsh/connections.xml)

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 : 32051
3            server : bigi01.localdomain
4              user : bigsql
5          password : ********
6       Autoconnect : false

JDBC Driver Properties
----------------------
None

Enter a number to change a given configuration property, or
(T)est, (D)elete, (B)ack, (Q)uit, Add (P)roperty, or (S)ave: T
```

The Big SQL database name is defined during the installation of BigInsights. The default is `bigsql`. In addition, a Big SQL database administrator account is also defined at installation. This account has SECADM (security administration) authority for Big SQL. By default, that user account is `bigsql`.

__10. When prompted, enter **T** to test your configuration.

__11. Verify that the test succeeded, and hit **Enter**.

```
Enter a number to change a given configuration property, or
(T)est, (D)elete, (B)ack, (Q)uit, Add (P)roperty, or (S)ave: T

Attempting connection...
Succeeded!
```

__12. Save your connection. Enter s, name your connection `bigsql`, and hit **Enter**.

```
JSQSH CONNECTION WIZARD - (edits $HOME/.jsqsh/connections.xml
The following connections are currently defined:

     Name                 Driver     Host                           Port
---  -------------------- ---------- ------------------------------ ------
  1  bigsql               db2        bigi01.localdomain              32051

Enter a connection number above to edit the connection, or:
(B)ack, (Q)uit, or (A)dd connection:
```

__13. Finally, quit the connection wizard when prompted. (Enter **Q**.)

You must resolve any connection errors before continuing with this lab. If you have any questions, visit Hadoop Dev (https://developer.ibm.com/hadoop/) and review the product documentation or post a message to the forum.

5.2. Getting help for JSqsh

Now that you're familiar with JSqsh connections, you're ready to work further with the shell.

__1. From JSqsh, type `\help` to display a list of available help categories.

```
2> \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                 |
+----------+-----------------------------------------------+
```

__2. Optionally, type `\help commands` to display help for supported commands. A partial list of supported commands is displayed on the initial screen.

```
2> \help commands
Available commands. Use "\help <command>" to display detailed help for a given command
+--------------+-------------------------------------------------------------------------+
| Command      | Description                                                             |
+--------------+-------------------------------------------------------------------------+
| \alias       | Creates an alias                                                        |
| \buf-append  | Appends the contents of one SQL buffer into another                     |
| \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                                           |
| \go          | Executes the contents of the current buffer                             |
--More--
```

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

5.3. 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 you created in a previous lesson. Assuming you named your connection `bigsql`, enter this command:

```
\connect bigsql
```

__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. Next, cut and paste 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 an earlier lab exercise, you created a connection using the `bigsql` user ID, so your table is BIGSQL.TEST1.

We've intentionally created a very simple Hadoop table for this exercise so that you can concentrate on working with JSqsh. Later, you'll learn more about CREATE TABLE options supported by Big SQL, including the LOCATION clause of CREATE TABLE. In these examples, where LOCATION is omitted, the default Hadoop directory path for these tables are at `/…/hive/warehouse/<schema>.db/<table>`.

Big SQL enables users with appropriate authority to create their own schemas by issuing a command such as

```
create schema if not exists testschema;
```

Authorized users can then create tables in that schema as desired. Furthermore, users can also create a table in a different schema, and if it doesn't already exist it will be implicitly created.

__4. Display all tables in the current schema with the `\tables` command.

```
\tables
```

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

This screen capture was taken from an environment in which only 1 Big SQL table was created (BIGSQL.TEST1).

__5. Optionally, display information about tables and views created in other schemas, such as the SYSCAT schema used for the Big SQL catalog. 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 meta data 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 we 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)
```

In case you're wondering, `SYSCAT.COLUMNS` is one of a number of views supplied over system catalog tables automatically maintained for you by the Big SQL service.

__9. Issue a query that restricts the number of rows returned to 5. For example, 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.

__10. Leave JSqsh open so you can explore additional features in the next section.

5.4. [Optional] Exploring additional JSqsh commands

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

__1. Review the history of commands you recently executed in the JSqsh shell. Type `\history` and **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.

__2. Enter `!!` (two exclamation points, without spaces) to recall the previously run statement. In the example below, the previous statement selects the first 5 rows from `SYSCAT.TABLES.` To run the statement, type a semi-colon on the following line.

```
[bigi01.localdomain][bigsql] 1> !!
[bigi01.localdomain][bigsql] 1> select tabschema, tabname from syscat.tables fetch first 5 rows only
[bigi01.localdomain][bigsql] 2> ;
+-----------+-----------------------+
| TABSCHEMA | TABNAME               |
+-----------+-----------------------+
| BIGSQL    | TEST1                 |
| SYSCAT    | ATTRIBUTES            |
| SYSCAT    | AUDITPOLICIES         |
| SYSCAT    | AUDITUSE              |
| SYSCAT    | BUFFERPOOLDBPARTITIONS |
+-----------+-----------------------+
5 rows in results(first row: 0.035s; total: 0.038s)
```

__3. 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.

__4. Experiment with JSqsh's ability to support piping of output to an external program. 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:

```
[bigi01.localdomain][bigsql] 1> select tabschema, tabname from syscat.tables
[bigi01.localdomain][bigsql] 2> go | more
+-----------+-----------------------------------+
| TABSCHEMA | TABNAME                           |
+-----------+-----------------------------------+
| SYSIBMADM | SNAPTBSP                          |
| SYSIBMADM | SNAPTBSP_PART                     |
| SYSIBMADM | SNAPTBSP_QUIESCER                 |
| SYSIBMADM | SNAPTBSP_RANGE                    |
| SYSIBMADM | SNAPUTIL                          |
| SYSIBMADM | SNAPUTIL_PROGRESS                 |
| SYSIBMADM | TAB                               |
| SYSIBMADM | TBSP_UTILIZATION                  |
| SYSIBMADM | TOP_DYNAMIC_SQL                   |
| SYSIBMADM | USER_ALL_TABLES                   |
| SYSIBMADM | USER_ARGUMENTS                    |
| SYSIBMADM | USER_CATALOG                      |
| SYSIBMADM | USER_COL_COMMENTS                 |
| SYSIBMADM | USER_CONSTRAINTS                  |
| SYSIBMADM | USER_CONS_COLUMNS                 |
| SYSIBMADM | USER_DEPENDENCIES                 |
| SYSIBMADM | USER_ERRORS                       |
| SYSIBMADM | USER_INDEXES                      |
| SYSIBMADM | USER_IND_COLUMNS                  |
| SYSIBMADM | USER_IND_PARTITIONS               |
| SYSIBMADM | USER_OBJECTS                      |
--More--665 rows in results(first row: 0.005s; total: 0.014s)
```

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.

__5. Experiment with JSqsh's ability to redirect output to a local file rather than the console display. 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.

__6. Exit the shell:

```
quit
```

__7. From a terminal window, view the output file:

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

```
[bigsql@bigi01 ~]$ cat test1.out
+-----------+---------+-------+----------+--------+
| TABSCHEMA | COLNAME | COLNO | TYPENAME | LENGTH |
+-----------+---------+-------+----------+--------+
| BIGSQL    | COL1    |     0 | INTEGER  |      4 |
| BIGSQL    | COL2    |     1 | VARCHAR  |      5 |
+-----------+---------+-------+----------+--------+
```

__8. 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.

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

```
vim test.sql
```

__b. 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;
```

__c. Save your file (hit 'esc' to exit INSERT mode then type `:wq`) and return to the command line.

__d. 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/current/jsqsh/bin/jsqsh bigsql -i test.sql
```

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

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

```
[bigsql@bigi01 ~]$ /usr/ibmpacks/common-utils/current/jsqsh/bin/jsqsh bigsql -i test.sql
+-----------+------------------------+
| TABSCHEMA | TABNAME                |
+-----------+------------------------+
| BIGSQL    | TEST1                  |
| SYSCAT    | ATTRIBUTES             |
| SYSCAT    | AUDITPOLICIES          |
| SYSCAT    | AUDITUSE               |
| SYSCAT    | BUFFERPOOLDBPARTITIONS |
+-----------+------------------------+
5 rows in results(first row: 0.032s; total: 0.038s)
+-----------+-------------+-------+-----------+-----------+
| 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: 0.141s; total: 0.144s)
```

__9. Finally, clean up the your database. 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.

Lab 6 Querying structured data with Big SQL

In this lab, you will execute Big SQL queries to investigate data stored in Hadoop. Big SQL provides broad SQL support based on the ISO SQL standard. You can issue queries using JDBC or ODBC drivers to access data that is stored in Hadoop in the same way that you access relational databases from your enterprise applications. Multiple queries can be executed concurrently. The SQL query engine supports joins, unions, grouping, common table expressions, windowing functions, and other familiar SQL expressions.

This tutorial uses sales data from a fictional company sells outdoor products to retailers and directly to consumers through a web site. The firm maintains its data in a series of FACT and DIMENSION tables, as is common in relational data warehouse environments. In this lab, you will explore how to create, populate, and query a subset of the star schema database to investigate the company's performance and offerings. Note that Big SQL provides scripts to create and populate the more than 60 tables that comprise the sample GOSALESDW database. You will use fewer than 10 of these tables in this lab.

Prior to starting this lab, you must know how to connect to your Big SQL server and execute SQL from a supported tool **DSM** or **JSqsh**. If necessary, complete the prior lab on DSM or JSqsh before proceeding.

After you complete the lessons in this module, you will understand how 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.

[Allow **1.5 hours** to complete this lab.]

6.1. Creating sample tables and loading sample data

In this lesson, you will create several sample tables and load data into these tables from local files.

NOTE: This step has been created for you in order to waste time loading the data.

__1. 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 lab.

Subsequent examples in this section presume your sample data is in the `/usr/ibmpacks/bigsql/4.2.0.0/bigsql/samples/data` directory. This is the location of the data in typical Big SQL installations.

__2. If necessary, launch your query execution tool (e.g., DSM or JSqsh) and establish a connection to your Big SQL server following the standard process for your environment.

__3. Create several tables in your default schema. Issue each of the following CREATE TABLE statements one at a time, and verify that each completed successfully:

By JSqsh, to avoid character issues, you can:

1. Create a file (eg: create.sql)
2. Copy create queries
3. Execute JSqsh with the file as parameter

```
[bigsql@bigi01 ~]$ jsqsh bigsql -i queries.sql
0 rows affected (total: 0.569s)
0 rows affected (total: 0.222s)
0 rows affected (total: 0.223s)
0 rows affected (total: 0.228s)
0 rows affected (total: 0.258s)
0 rows affected (total: 0.169s)
0 rows affected (total: 0.228s)
0 rows affected (total: 0.201s)
```

```
-- 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_en    VARCHAR(90) NOT NULL
, country_en   VARCHAR(90) NOT NULL
, region_de    VARCHAR(90), country_de   VARCHAR(90), region_fr   VARCHAR(90)
, country_fr   VARCHAR(90), region_ja    VARCHAR(90), country_ja  VARCHAR(90)
, region_cs    VARCHAR(90), country_cs   VARCHAR(90), region_da   VARCHAR(90)
, country_da   VARCHAR(90), region_el    VARCHAR(90), country_el  VARCHAR(90)
, region_es    VARCHAR(90), country_es   VARCHAR(90), region_fi   VARCHAR(90)
, country_fi   VARCHAR(90), region_hu    VARCHAR(90), country_hu  VARCHAR(90)
, region_id    VARCHAR(90), country_id   VARCHAR(90), region_it   VARCHAR(90)
, country_it   VARCHAR(90), region_ko    VARCHAR(90), country_ko  VARCHAR(90)
, region_ms    VARCHAR(90), country_ms   VARCHAR(90), region_nl   VARCHAR(90)
, country_nl   VARCHAR(90), region_no    VARCHAR(90), country_no  VARCHAR(90)
, region_pl    VARCHAR(90), country_pl   VARCHAR(90), region_pt   VARCHAR(90)
, country_pt   VARCHAR(90), region_ru    VARCHAR(90), country_ru  VARCHAR(90)
, region_sc    VARCHAR(90), country_sc   VARCHAR(90), region_sv   VARCHAR(90)
, country_sv   VARCHAR(90), region_tc    VARCHAR(90), country_tc  VARCHAR(90)
, region_th    VARCHAR(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.

__4. Load data into each of these tables using sample data provided in files. **Change the SFTP and file path specifications in each of the following examples to match your environment.** Then, one at a time, issue each LOAD statement and verify that the operation completed successfully. LOAD returns a warning in JSqsh message providing details on the number of rows loaded, etc.

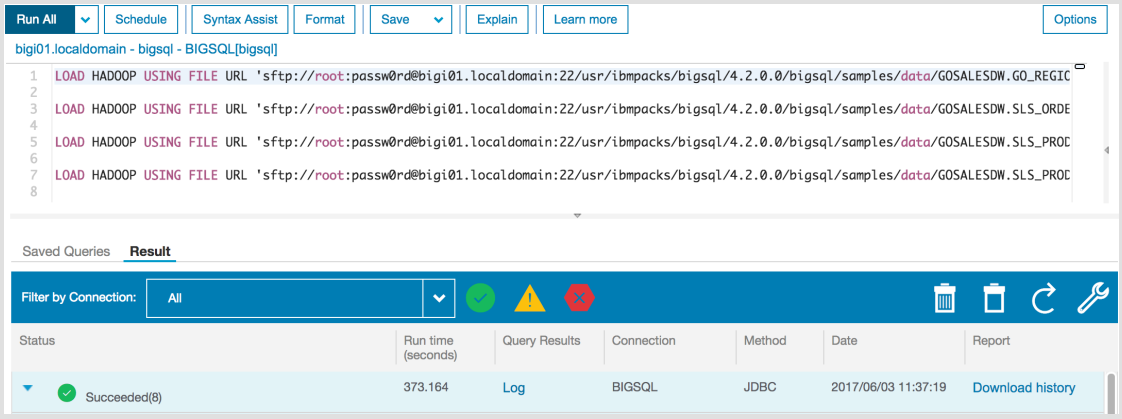
Example of warning message:

```
[bigi01.localdomain][bigsql] 1> load hadoop using file url 'sftp://root:passw0rd@bigi01.loca
ldomain:22/usr/ibmpacks/bigsql/4.2.0.0/bigsql/samples/data/GOSALESDW.GO_REGION_DIM.txt' with
 SOURCE PROPERTIES ('field.delimiter'='\t') INTO TABLE GO_REGION_DIM overwrite;
WARN [State: 0    ][Code: 5108]: Loading of data to a Hadoop table or processing of data in
an external table completed. Number of rows processed: "21". Number of source records: "21".
 If the source was a file, number of skipped lines: "0". Number of rejected source records:
"0". Job or file identifier: "job_1496217291001_0003".. SQLCODE=5108, SQLSTATE=0    , DRIVER
=3.71.27
0 rows affected (total: 1m7.931s)
```

You can use -i parameter:

```
[bigsql@bigi01 ~]$ jsqsh bigsql -i load.sql
WARN [State: 0    ][Code: 5108]: Loading of data to a Hadoop table or processing of data in
an external table completed. Number of rows processed: "21". Number of source records: "21".
 If the source was a file, number of skipped lines: "0". Number of rejected source records:
"0". Job or file identifier: "job_1496217291001_0005".. SQLCODE=5108, SQLSTATE=0    , DRIVER
=3.71.27
0 rows affected (total: 41.892s)
```

DSM completed succeeded:




```
load hadoop using file url
'sftp://yourID:yourPassword@bigi01.localdomain:22/usr/ibmpacks/bigsql/4.2.0.0/bigsql/s
amples/data/GOSALESDW.GO_REGION_DIM.txt' with SOURCE PROPERTIES
('field.delimiter'='\t') INTO TABLE GO_REGION_DIM overwrite;

load hadoop using file url
'sftp://yourID:yourPassword@bigi01.localdomain:22/usr/ibmpacks/bigsql/4.2.0.0/bigsql/s
amples/data/GOSALESDW.SLS_ORDER_METHOD_DIM.txt' with SOURCE PROPERTIES
('field.delimiter'='\t') INTO TABLE SLS_ORDER_METHOD_DIM overwrite;

load hadoop using file url
'sftp://yourID:yourPassword@bigi01.localdomain:22/usr/ibmpacks/bigsql/4.2.0.0/bigsql/s
amples/data/GOSALESDW.SLS_PRODUCT_BRAND_LOOKUP.txt' with SOURCE PROPERTIES
('field.delimiter'='\t') INTO TABLE SLS_PRODUCT_BRAND_LOOKUP overwrite;

load hadoop using file url
'sftp://yourID:yourPassword@bigi01.localdomain:22/usr/ibmpacks/bigsql/4.2.0.0/bigsql/s
amples/data/GOSALESDW.SLS_PRODUCT_DIM.txt' with SOURCE PROPERTIES
('field.delimiter'='\t') INTO TABLE SLS_PRODUCT_DIM overwrite;

load hadoop using file url
'sftp://yourID:yourPassword@bigi01.localdomain:22/usr/ibmpacks/bigsql/4.2.0.0/bigsql/s
amples/data/GOSALESDW.SLS_PRODUCT_LINE_LOOKUP.txt' with SOURCE PROPERTIES
('field.delimiter'='\t') INTO TABLE SLS_PRODUCT_LINE_LOOKUP overwrite;

load hadoop using file url
'sftp://yourID:yourPassword@bigi01.localdomain:22/usr/ibmpacks/bigsql/4.2.0.0/bigsql/s
amples/data/GOSALESDW.SLS_PRODUCT_LOOKUP.txt' with SOURCE PROPERTIES
('field.delimiter'='\t') INTO TABLE SLS_PRODUCT_LOOKUP overwrite;

load hadoop using file url
'sftp://yourID:yourPassword@bigi01.localdomain:22/usr/ibmpacks/bigsql/4.2.0.0/bigsql/s
amples/data/GOSALESDW.SLS_SALES_FACT.txt' with SOURCE PROPERTIES
('field.delimiter'='\t') INTO TABLE SLS_SALES_FACT overwrite;
```

```
load hadoop using file url
'sftp://yourID:yourPassword@bigi01.localdomain:22/usr/ibmpacks/bigsql/4.2.0.0/bigsql/s
amples/data/GOSALESDW.MRK_PROMOTION_FACT.txt' with SOURCE PROPERTIES
('field.delimiter'='\t') INTO TABLE MRK_PROMOTION_FACT overwrite;
```

Let's explore the LOAD syntax shown in these examples briefly. 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 (`bigi01.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. See the BigInsights Knowledge Center (product documentation) for details. In addition, you can load data directly from a remote relational DBMS via a JDBC connection. The Knowledge Center includes examples of that.

__5. 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.

```
-- 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;
```

6.2. Querying the data with Big SQL

Now you're ready to query your tables. Based on earlier exercises, you've already seen that you can perform basic SQL operations, including projections (to extract specific columns from your tables) and restrictions (to extract specific rows meeting certain conditions you specified). Let's explore a few examples that are a bit more sophisticated.

In this lesson, 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.

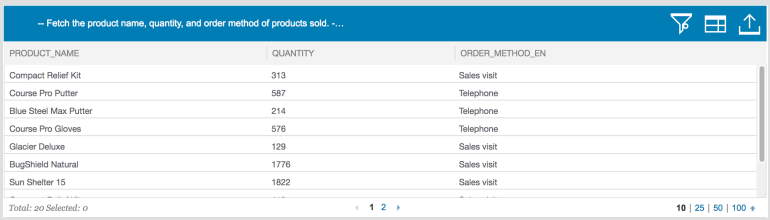
__1. Join data from multiple tables to return the product name, quantity and order method of goods that have been sold. For simplicity, limit the number of returns rows to 20. To achieve this, execute the following query:

```
-- 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;
```

Let's review a few aspects of this query briefly:

- 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.

DSM



JSqsh

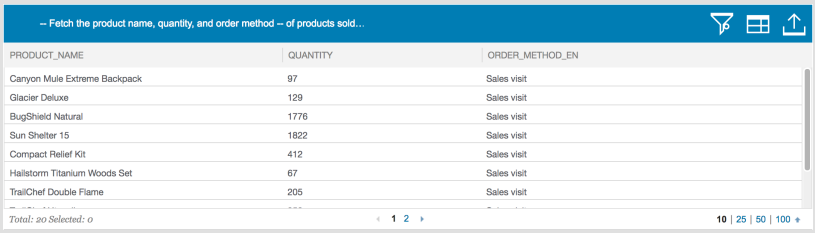
```
+------------------------------+----------+-----------------+
| Compact Relief Kit           |      313 | Sales visit     |
| Course Pro Putter            |      587 | Telephone       |
| Blue Steel Max Putter        |      214 | Telephone       |
| Course Pro Gloves            |      576 | Telephone       |
| 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:

DSM



JSqsh

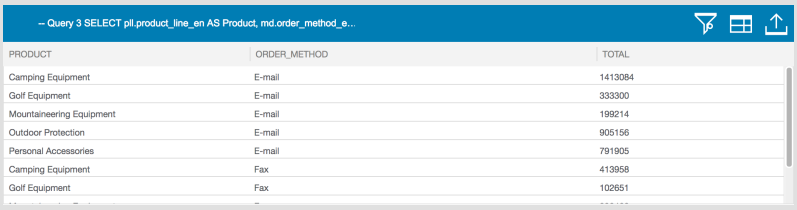
```
+-------------------------------+----------+-----------------+
| Canyon Mule Extreme Backpack  |       97 | Sales visit     |
| Glacier Deluxe                |      129 | Sales visit     |
| BugShield Natural             |     1776 | Sales visit     |
| Sun Shelter 15                |     1822 | 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.

DSM



JSqsh

```
+--------------------------+--------------+----------+
| PRODUCT                  | ORDER_METHOD |    TOTAL |
+--------------------------+--------------+----------+
| Camping Equipment        | E-mail       |  1413084 |
| Golf Equipment           | E-mail       |   333300 |
| Mountaineering Equipment | E-mail       |   199214 |
| Outdoor Protection       | E-mail       |   905156 |
| Personal Accessories     | E-mail       |   791905 |
| Camping Equipment        | Fax          |   413958 |
| Golf Equipment           | Fax          |   102651 |
| Mountaineering Equipment | Fax          |   292408 |
| Outdoor Protection       | Fax          |   311583 |
| Personal Accessories     | Fax          |   359414 |
| Camping Equipment        | Mail         |   348058 |
```

6.3. 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.

__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.

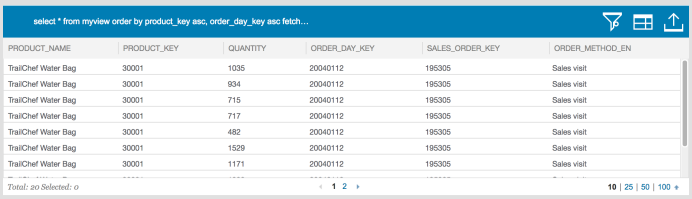
```
create view myview as
select product_name, sales.product_key, mkt.quantity,
      sales.order_day_key, sales.sales_order_key, order_method_en
from
      mrk_promotion_fact mkt,
      sls_sales_fact sales,
      sls_product_dim prod,
      sls_product_lookup pnumb,
      sls_order_method_dim meth
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:

DSM



JSqsh

```
+-----------------+-------------+----------+--------------+-----------------+-----------------+
| PRODUCT_NAME    | PRODUCT_KEY | QUANTITY | ORDER_DAY_KEY | SALES_ORDER_KEY | ORDER_METHOD_EN |
+-----------------+-------------+----------+--------------+-----------------+-----------------+
| TrailChef Water Bag |       30001 |     1035 |     20040112 |          195305 | Sales visit     |
| TrailChef Water Bag |       30001 |      934 |     20040112 |          195305 | Sales visit     |
| TrailChef Water Bag |       30001 |      715 |     20040112 |          195305 | Sales visit     |
| TrailChef Water Bag |       30001 |      717 |     20040112 |          195305 | Sales visit     |
| TrailChef Water Bag |       30001 |      482 |     20040112 |          195305 | Sales visit     |
| TrailChef Water Bag |       30001 |     1529 |     20040112 |          195305 | Sales visit     |
| TrailChef Water Bag |       30001 |     1171 |     20040112 |          195305 | Sales visit     |
| TrailChef Water Bag |       30001 |     1033 |     20040112 |          195305 | Sales visit     |
| TrailChef Water Bag |       30001 |     1390 |     20040112 |          195305 | Sales visit     |
| TrailChef Water Bag |       30001 |      968 |     20040112 |          195305 | Sales visit     |
| TrailChef Water Bag |       30001 |      714 |     20040112 |          195305 | Sales visit     |
| TrailChef Water Bag |       30001 |      634 |     20040112 |          195305 | Sales visit     |
| TrailChef Water Bag |       30001 |      891 |     20040112 |          195305 | Sales visit     |
| TrailChef Water Bag |       30001 |     1350 |     20040112 |          195305 | Sales visit     |
```

6.4. 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 exercise, 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.

__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;
```

6.5. [Optional] 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 lab 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/.

__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,
```

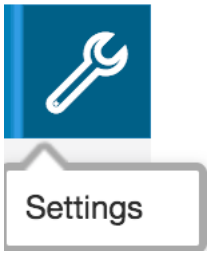
```
sls_order_method_dim meth
WHERE
pnumb.product_language='EN'
AND sales.product_key=prod.product_key
AND prod.product_number=pnumb.product_number
AND meth.order_method_key=sales.order_method_key
and sales.quantity > 5500;
```

__3. Query the table. Note that your SELECT statement does not need to be modified in any way because of the underlying file format.

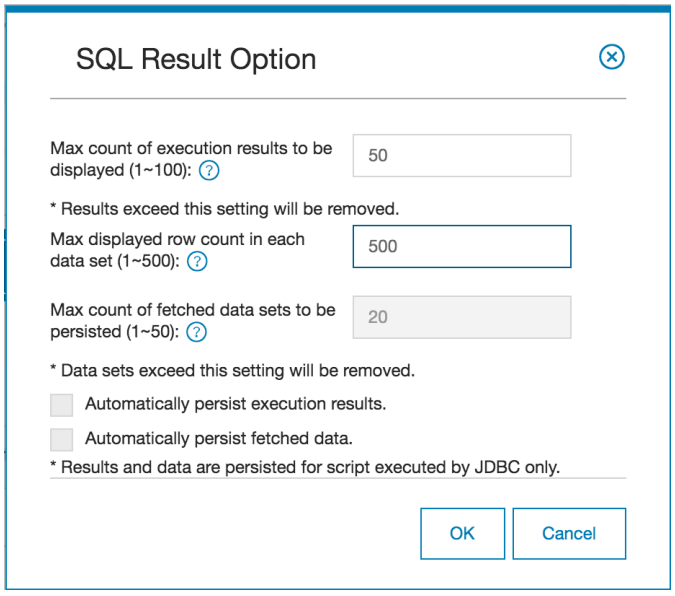
```
select * from big_sales_parquet;
```

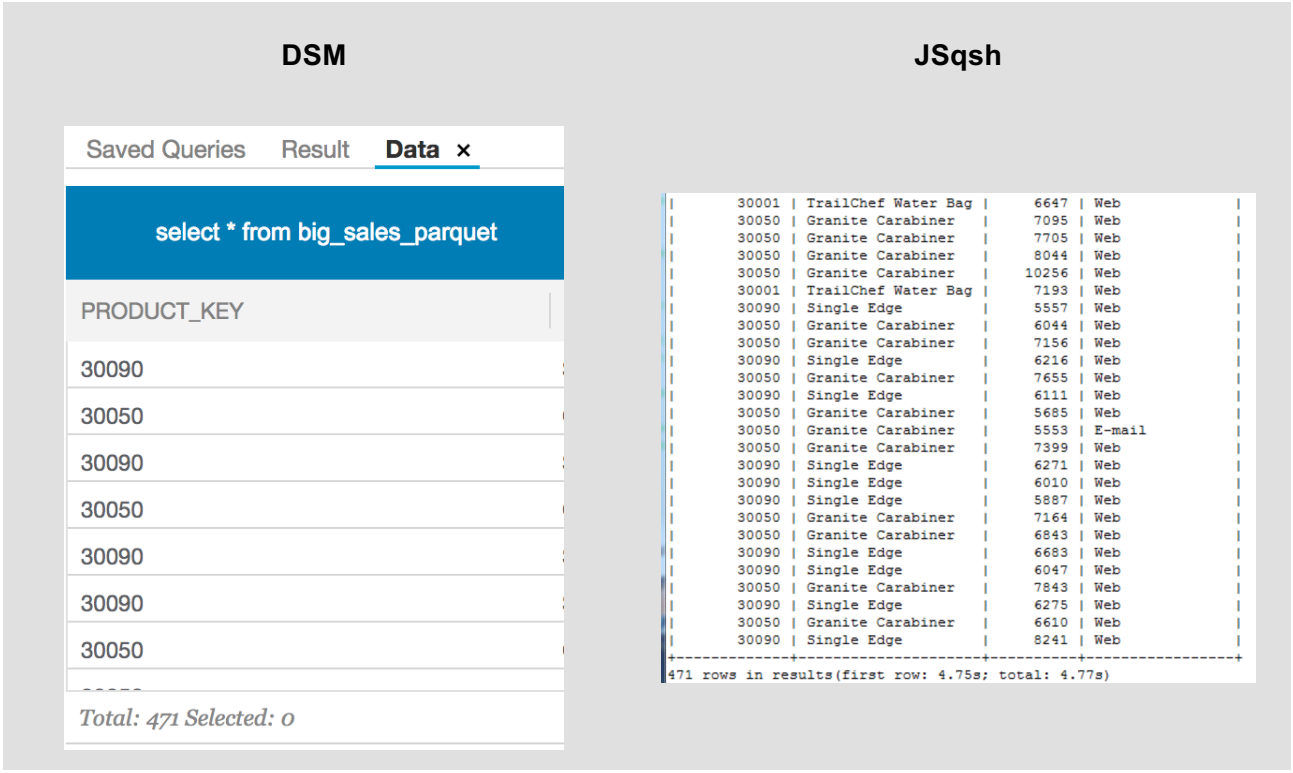
__4. Inspect the results, a subset of which are shown below. The query should return 471 rows.

DSM has a limit results (100). To change it, press:



And change:





6.6. [Optional] Working with external tables

The previous exercises in this lab 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 exercise, 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. Switch user to **hdfs**

```
su hdfs
```

__3. Check the directory permissions for your DFS.

```
hdfs dfs -ls /
```

```
[hdfs@bigi01 root]$ hdfs dfs -ls /
Found 8 items
drwxrwxrwx   - yarn   hadoop          0 2017-05-30 10:57 /app-logs
drwxr-xr-x   - hdfs   hdfs            0 2017-05-29 16:22 /apps
drwxrwxr-x   - hdfs   hdfs            0 2017-05-30 10:42 /biginsights
drwxr-xr-x   - hdfs   hdfs            0 2017-05-29 16:13 /iop
drwxr-xr-x   - mapred hdfs            0 2017-05-29 16:13 /mapred
drwxrwxrwx   - mapred hadoop          0 2017-05-29 16:13 /mr-history
drwxrwxrwx   - hdfs   hdfs            0 2017-05-31 09:55 /tmp
drwxr-xr-x   - hdfs   hdfs            0 2017-05-30 11:33 /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 lab using your standard lab user account.

As user hdfs, issue this command:

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

Next, confirm the effect of your change:

```
hdfs dfs -ls /
```

```
[hdfs@bigi01 root]$ hdfs dfs -ls /
Found 8 items
drwxrwxrwx   - yarn   hadoop          0 2017-05-30 10:57 /app-logs
drwxr-xr-x   - hdfs   hdfs            0 2017-05-29 16:22 /apps
drwxrwxr-x   - hdfs   hdfs            0 2017-05-30 10:42 /biginsights
drwxr-xr-x   - hdfs   hdfs            0 2017-05-29 16:13 /iop
drwxr-xr-x   - mapred hdfs            0 2017-05-29 16:13 /mapred
drwxrwxrwx   - mapred hadoop          0 2017-05-29 16:13 /mr-history
drwxrwxrwx   - hdfs   hdfs            0 2017-05-31 09:55 /tmp
drwxrwxrwx   - hdfs   hdfs            0 2017-05-30 11:33 /user
```

Exit the hdfs user account:

```
exit
```

Finally, exit the root user account and return to your standard user account (bigsql):

```
exit

su bigsql
```

__4. 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
```

__5. Upload the source data files into their respective DFS directories. Change the local and DFS directories information below to match your environment.

```
hdfs dfs -copyFromLocal
/usr/ibmpacks/bigsql/4.2.0.0/bigsql/samples/data/GOSALESDW.SLS_PRODUCT_DIM.tx
t /user/bigsql_lab/sls_product_dim/SLS_PRODUCT_DIM.txt
```

__6. 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
```

```
[bigsql@bigi01 root]$ hdfs dfs -ls /user/bigsql_lab/sls_product_dim
Found 1 items
-rw-r--r--   2 bigsql hdfs      24089 2017-05-31 15:23 /user/bigsql_lab/sls_product_dim/SLS_PRODUCT_DIM.txt
```

__7. From your query execution environment (such as JSqsh), 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.

```
-- 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';
```

__8. Query the table.

```
select product_key, introduction_date from sls_product_dim_external
where discontinued_date is not null
```

```
fetch first 20 rows only;
```

__9. Inspect the results.

DSM

select product_key, introduction_date from sls_product_dim_external wh…

PRODUCT_KEY	INTRODUCTION_DATE
30134	2003-01-01 00:00:00.0
30139	2003-01-01 00:00:00.0
30143	2003-01-01 00:00:00.0
30146	2003-01-01 00:00:00.0
30147	2003-01-01 00:00:00.0
30154	2003-01-01 00:00:00.0
30156	2003-01-01 00:00:00.0

Total: 20 Selected: 0 1 2

JSqsh

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

6.7. [Optional] Creating and querying the full sample database

Big SQL ships with sample SQL scripts for creating, populating, and querying more than 60 tables. These tables are part of the GOSALESDW schema -- a schema that differs from the one used in this lab. (You created tables in the default schema – i.e., your user ID's schema. Since the JSqsh connection you created earlier used the `bigsql` ID, your schema was BIGSQL.)

If desired, use standard Linux operating system facilities to inspect the SQL scripts and sample data for the GOSALESDW schema in the samples directory. By default, this location is `/usr/ibmpacks/bigsql/4.2.0.0/bigsql/samples`. Within this directory, you'll find subdirectories containing (1) the full sample data for the GOSALESDW tables and (2) a collection of SQL scripts for creating, loading, and querying these tables. Feel free to use the supplied scripts to create all GOSALESDW tables, load data into these tables, and query these tables. Note that you may need to modify portions of these scripts to match your environment.

Lab 7 [Optional] Developing and executing SQL user-defined functions

Big SQL enables users to create their own SQL functions that can be invoked in queries. User-defined functions (UDFs) promote code re-use and reduce query complexity. They can be written to return a single (scalar) value or a result set (table). Programmers can write UDFs in SQL or any supported programming languages (such as Java and C). For simplicity, this lab focuses on SQL UDFs.

After you complete this lab, you will understand how to:

- Create scalar and table UDFs written in SQL.
- Incorporate procedural logic in your UDFs.
- Invoke UDFs in Big SQL queries.
- Drop UDFs.

[Allow **1 - 1.5** hours to complete this lab.]

Please note that this lab discusses only some of the capabilities of Big SQL scalar and table UDFs. For an exhaustive list of all the capabilities, please see the BigInsights 4.2 Knowledge Center (https://www.ibm.com/support/knowledgecenter/SSPT3X_4.2.0/com.ibm.swg.im.infosphere.biginsights.welcome.doc/doc/welcome.html).

Prior to starting this lab, you must be familiar with how to use the Big SQL command line (JSqsh), and you must have created the sample GOSALESDW tables. If necessary, work through earlier lab exercises on these topics.

7.1. Understanding UDFs

Big SQL provides many built-in functions to perform common computations. An example is `dayname()`, which takes a date/timestamp and returns the corresponding day name, such as Friday.

Often, organizations need to perform some customized or complex operation on their data that's beyond the scope of any built-in-function. Big SQL allows users to embed their customized business logic inside a user-defined function (UDF) and write queries that call these UDFs.

As mentioned earlier, Big SQL supports two types of UDFs:

1. **Scalar UDF**: These functions take one or more values as input and return a single value as output. For example, a scalar UDF can take three values (price of an item, percent discount on that item, and percent sales tax) to compute the final price of that item.

2. **Table UDF**: These functions take one or more values as input and return a whole table as output. For example, a table UDF can take single value (department-id) as input and return a table of employees who work in that department. This result set could have multiple columns, such as employee-id, employee-first-name, employee-last-name.

Once created, UDFs can be incorporated into queries in a variety of ways, as you'll soon see.

In this lab, you will first set up your environment for UDF development and then explore how to create and invoke UDFs through various exercises.

Ready to get started?

7.2. Preparing JSqsh to create and execute UDFs

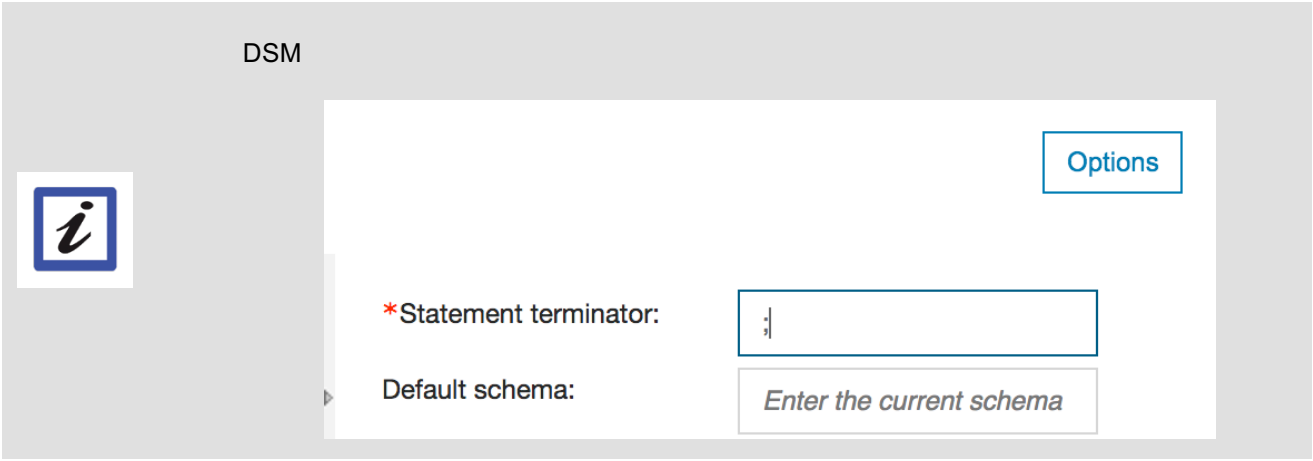
In this section, you will set up your JSqsh environment for UDF development.

__1. If necessary, launch JSqsh using the connection to your `bigsql` database. (This was covered in an earlier lab.)

__2. Reset the default SQL terminator character to “@”:

```
\set terminator = @;
```

Because some of the UDFs you will be developing involve multiple SQL statements, you must reset the JSqsh default termination character so that the semi-colon following each SQL statement in your UDF is not interpreted as the end of the CREATE FUNCTION statement for your UDF.



__3. Validate that the terminator was effectively reset:

```
\set @
```

__4. Inspect the output from the command (a subset of which is shown below), and verify that the terminator property is set to @.

```
| terminator          | @                                                    |
| timeout             | 0                                                    |
| timer               | true                                                 |
| user                | bigsql                                               |
| version             | 4.8                                                  |
| width               | 118                                                  |
| window_size         | 600x400                                              |
+---------------------+------------------------------------------------------+
```

You're now ready to create your first Big SQL UDF.

7.3. Creating and executing a scalar UDF

In this section, you will create a scalar SQL UDF to compute final price of a particular item that was sold. Your UDF will require several input parameters:

- unit sale price: Price of one item
- quantity: Number of units of this item being sold in this transaction
- % discount: Discount on the item (computed before tax)
- % sales-tax: Sales tax (computed after discount)

As you might expect, your UDF will return a single value – the final price of the item.

After creating and registering the UDF, you will invoke it using some test values to ensure that it behaves correctly. Afterwards, you will invoke it in a query, passing in values from columns in a table as input to your function.

__1. Create a UDF named `new_final_price` in the `gosalesdw` schema:

```
CREATE OR REPLACE FUNCTION gosalesdw.new_final_price
(
  quantity INTEGER,
  unit_sale_price DOUBLE,
  discount_in_percent DOUBLE,
  sales_tax_in_percent DOUBLE
)
RETURNS DOUBLE
LANGUAGE SQL
RETURN (quantity * unit_sale_price) * DOUBLE(1 - discount_in_percent / 100.0) *
DOUBLE(1 + sales_tax_in_percent / 100.0) @
```

__2. Review the logic of this function briefly. This first line creates the function, which is defined to take four input parameters. The RETURNS clause indicates that a single (scalar) value of type DOUBLE will be returned. The function's language is as SQL. Finally, the last two lines include the function's logic, which simply performs the necessary arithmetic operations to calculate the final sales price of an item.

__3. After creating the function, test it using some sample values. A simple way to do this is with the VALUES clause shown here:

```
VALUES gosalesdw.new_final_price (1, 10, 20, 8.75) @
```

__4. Verify that result returned by your test case is

```
8.70000
```

__5. Next, use the UDF in a query to compute the final price for items listed in sales transactions in the SLS_SALES_FACT table. Note that this query uses values from two columns in the table as input for the quantity and unit price and two user-supplied values as input for the discount rate and sales tax rate.

```
SELECT sales_order_key, quantity, unit_sale_price, gosalesdw.new_final_price(quantity,
unit_sale_price, 20, 8.75) as final_price
FROM sls_sales_fact
ORDER BY sales_order_key
FETCH FIRST 10 ROWS ONLY@
```

__6. Inspect the results.

```
+-----------------+----------+-----------------+-------------+
| SALES_ORDER_KEY | QUANTITY | UNIT_SALE_PRICE | FINAL_PRICE |
+-----------------+----------+-----------------+-------------+
|          100001 |      256 |        33.69000 |  7503.43680 |
|          100002 |       92 |       102.30000 |  8188.09200 |
|          100003 |      162 |       111.31000 | 15688.03140 |
|          100004 |      172 |        38.90000 |  5820.99600 |
|          100005 |       74 |       334.43000 | 21530.60340 |
|          100006 |       90 |        75.84000 |  5938.27200 |
|          100007 |      422 |         6.00000 |  2202.84000 |
|          100008 |     3252 |         6.51000 | 18418.35240 |
|          100009 |     1107 |         5.76000 |  5547.39840 |
|          100010 |       88 |       124.72000 |  9548.56320 |
+-----------------+----------+-----------------+-------------+
10 rows in results(first row: 0.589s; total: 0.592s)
```

__7. Now invoke your UDF in the WHERE clause of a query. (Scalar UDFs can be included anywhere in a SQL statement that a scalar value is expected.) This query is similar to your previous query expect that it includes a WHERE clause to restrict the result set to items with a file price of greater than 7000.

```
-- scalar UDF can be used wherever a scalar value is expected,
-- for example in WHERE clause
SELECT sales_order_key, quantity, unit_sale_price,
gosalesdw.new_final_price(quantity, unit_sale_price, 20, 8.75) as final_price
FROM sls_sales_fact
WHERE gosalesdw.new_final_price(quantity, unit_sale_price, 20, 8.75) > 7000
ORDER BY sales_order_key
FETCH FIRST 10 ROWS ONLY@
```

__8. Note that your results no longer include rows with items priced at 7000 or below.

```
+-----------------+----------+-----------------+-------------+
| SALES_ORDER_KEY | QUANTITY | UNIT_SALE_PRICE | FINAL_PRICE |
+-----------------+----------+-----------------+-------------+
|          100001 |      256 |        33.69000 |  7503.43680 |
|          100002 |       92 |       102.30000 |  8188.09200 |
|          100003 |      162 |       111.31000 | 15688.03140 |
|          100005 |       74 |       334.43000 | 21530.60340 |
|          100008 |     3252 |         6.51000 | 18418.35240 |
|          100010 |       88 |       124.72000 |  9548.56320 |
|          100012 |      354 |        83.78000 | 25802.56440 |
|          100013 |      261 |       344.22000 | 78162.03540 |
|          100014 |      139 |       541.65000 | 65501.73450 |
|          100015 |      279 |       120.64000 | 29282.94720 |
+-----------------+----------+-----------------+-------------+
10 rows in results(first row: 0.507s; total: 0.510s)
```

7.4. Optional: Invoking UDFs without providing fully-qualified name

In the previous lab, you used the fully-qualified UDF name (GOSALESDW.NEW_FINAL_PRICE) in your VALUES or SELECT statements. (GOSALESDW is the schema name and NEW_FINAL_PRICE is the function name.)

A UDF with the same name and input parameters can be specified in more than one schema, so providing Big SQL with the fully qualified function name identifies the function you want to execute. With Big SQL, you can also specify a list of schemas in a special register called “CURRENT PATH” (also called “CURRENT FUNCTION PATH”). When Big SQL encounters an unqualified UDF (in which no schema name specified), it will look for the UDF in the schemas specified in CURRENT PATH.

In this lab, you’ll learn how to set the CURRENT PATH and invoke your function without specifying a schema name.

__1. To begin, determine the values of your current function path by issuing either of these two statements:

```
VALUES CURRENT PATH@

VALUES CURRENT FUNCTION PATH@
```

__2. Verify that the results are similar to this:

```
"SYSIBM","SYSFUN","SYSPROC","SYSIBMADM","BIGSQL"
```

__3. Add the GOSALESDW schema to the current path:

```
SET CURRENT FUNCTION PATH = CURRENT FUNCTION PATH, "GOSALESDW"@
```

__4. Inspect your function path setting again:

```
VALUES CURRENT FUNCTION PATH@
```

__5. Verify that the GOSALESDW schema is now in the path:

```
"SYSIBM","SYSFUN","SYSPROC","SYSIBMADM","BIGSQL","GOSALESDW"
```

__6. Re-run the query you executed earlier, but this time remove the GOSALESDW schema from the function name with you invoke it:

```
SELECT sales_order_key, quantity, unit_sale_price,
new_final_price(quantity, unit_sale_price, 20, 8.75) as final_price
FROM sls_sales_fact
ORDER BY sales_order_key
FETCH FIRST 10 ROWS ONLY@
```

Note that Big SQL will automatically locate your UDF and successfully execute your query.

__7. Inspect the results.

```
+-----------------+----------+-----------------+-------------+
| SALES_ORDER_KEY | QUANTITY | UNIT_SALE_PRICE | FINAL_PRICE |
+-----------------+----------+-----------------+-------------+
|          100001 |      256 |        33.69000 |  7503.43680 |
|          100002 |       92 |       102.30000 |  8188.09200 |
|          100003 |      162 |       111.31000 | 15688.03140 |
|          100004 |      172 |        38.90000 |  5820.99600 |
|          100005 |       74 |       334.43000 | 21530.60340 |
|          100006 |       90 |        75.84000 |  5938.27200 |
|          100007 |      422 |         6.00000 |  2202.84000 |
|          100008 |     3252 |         6.51000 | 18418.35240 |
|          100009 |     1107 |         5.76000 |  5547.39840 |
|          100010 |       88 |       124.72000 |  9548.56320 |
+-----------------+----------+-----------------+-------------+
10 rows in results(first row: 7.146s; total: 7.181s)
```

7.5. Incorporating IF/ELSE statements

Quite often, you may find it useful to incorporate conditional logic in your UDFs. In this section, you will learn how to include IF/ELSE statements to calculate the final price of an item based on a varying discount rate. To keep your work simple, you will create a modified version of the previous UDF that includes the following logic:

- If the unit price is 0 to 10, use a discount rate of X%
- If the unit price is 10 to 100, use a discount rate of Y%
- If the unit price is greater than 100, use a discount rate of Z%

The three different discount rates (X, Y, and Z) are based on input parameters.

__1. Create a UDF named new_final_price_v2 in the gosalesdw schema:

```
CREATE OR REPLACE FUNCTION gosalesdw.new_final_price_v2
(
```

```
  quantity INTEGER,
  unit_sale_price DOUBLE,
  discount_in_percent_if_price_0_t0_10 DOUBLE,
  discount_in_percent_if_price_10_to_100 DOUBLE,
  discount_in_percent_if_price_greater_than_100 DOUBLE,
  sales_tax_in_percent DOUBLE
)
RETURNS DOUBLE
LANGUAGE SQL
BEGIN ATOMIC
  DECLARE final_price DOUBLE;
  SET final_price = -1;

  IF unit_sale_price <= 10
  THEN
    SET final_price = (quantity * unit_sale_price) * DOUBLE(1 -
discount_in_percent_if_price_0_t0_10 / 100.0) * DOUBLE(1 + sales_tax_in_percent /
100.0) ;
  ELSEIF unit_sale_price <= 100
  THEN
    SET final_price = (quantity * unit_sale_price) * DOUBLE(1 -
discount_in_percent_if_price_10_to_100 / 100.0) * DOUBLE(1 + sales_tax_in_percent /
100.0) ;
  ELSE
    SET final_price = (quantity * unit_sale_price) * DOUBLE(1 -
discount_in_percent_if_price_greater_than_100 / 100.0) * DOUBLE(1 +
sales_tax_in_percent / 100.0) ;
  END IF;

  RETURN final_price;
END @
```

__2. Review the logic of this function briefly. As shown on lines 3 – 8, the function requires 6 input parameters. The first two represent the quantity ordered and the base unit price of each. The next three parameters specify different discount rates. The final input parameter represents the sales tax. The body of this function uses various conditional logic clauses (IF, THEN, ELSEIF, and ELSE) to calculate the final price of an item based on the appropriate discount rate and sales tax. Note that the unit price of the item determines the discount rate applied.

__3. Test you function's logic using sample data values:

```
VALUES gosalesdw.new_final_price_v2 (1, 100, 10, 20, 30, 8.75)@
```

__4. Verify that the result is

```
87.00000
```

If desired, review the function's logic to confirm that this is the correct value based on the input parameters. Note that 1 item was ordered at a price of $100, qualifying it for a 20% discount (to $80). Sales tax of 8.75% on $80 is $7, which results in a final item price of $87.

__5. Now invoke your UDF in a query to report the final sales prices for various items recorded in your SLS_SALES_FACT table:

```
SELECT sales_order_key, quantity, unit_sale_price,
gosalesdw.new_final_price_v2(quantity, unit_sale_price, 10,20,30, 8.75) as final_price
FROM sls_sales_fact
ORDER BY sales_order_key
FETCH FIRST 10 ROWS ONLY @
```

__6. Inspect the results.

```
+-----------------+----------+-----------------+-------------+
| SALES_ORDER_KEY | QUANTITY | UNIT_SALE_PRICE | FINAL_PRICE |
+-----------------+----------+-----------------+-------------+
|          100001 |      256 |        33.69000 |  7503.43680 |
|          100002 |       92 |       102.30000 |  7164.58050 |
|          100003 |      162 |       111.31000 | 13727.02747 |
|          100004 |      172 |        38.90000 |  5820.99600 |
|          100005 |       74 |       334.43000 | 18839.27797 |
|          100006 |       90 |        75.84000 |  5938.27200 |
|          100007 |      422 |         6.00000 |  2478.19500 |
|          100008 |     3252 |         6.51000 | 20720.64645 |
|          100009 |     1107 |         5.76000 |  6240.82320 |
|          100010 |       88 |       124.72000 |  8354.99280 |
+-----------------+----------+-----------------+-------------+
10 rows in results(first row: 1.207s; total: 1.209s)
```

7.6. Incorporating WHILE loops

Big SQL enables you to include loops in your scalar UDFs. In this section, you'll use a WHILE loop to create a mathematical function for factorials. As a reminder, the factorial of a non-negative integer N is the product of all positive integers less than or equal to N. In other words,

```
factorial(N) = N * (N-1) * (N-2) …...* 1
```

As an example,

factorial(5) = 5 * 4 * 3 * 2 * 1 = 120

__1. Create a scalar UDF named `gosalesdw.factorial` that uses a WHILE loop to perform the necessary multiplication operations.

```
-- WHILE-DO loop in scalar UDF
-- This example is independent of gosalesdw tables
-- Given a number n (n >= 1), returns its factorial
-- as long as it is in INTEGER range.
-------------------------------
-- Create scalar UDF with WHILE-DO loop
CREATE OR REPLACE FUNCTION gosalesdw.factorial(n INTEGER)
RETURNS INTEGER
LANGUAGE SQL
BEGIN ATOMIC
  DECLARE n2 INTEGER;
```

```
  DECLARE res INTEGER;
  SET res = n;
  SET n2 = n;

  loop1:
  WHILE (n2 >= 2)
  DO
    SET n2 = n2 - 1;
    SET res = res * n2;
  END WHILE  loop1;

  RETURN res;
END @
```

__2. Review the logic of this function. Note that two variables are declared and set to the value of the input parameter. The first variable (`res`) holds the result of the computation. Its value changes as the body of the WHILE loop is executed. The second variable (`n2`) controls the loop's execution and serves as part of the calculation of the factorial.

__3. Test your function supplying different input parameters:

```
-- The output of factorial(5) should be 120

VALUES gosalesdw.factorial(5)@

-- The output of factorial(7) should be 5040

VALUES gosalesdw.factorial(7)@
```

__4. Optionally, drop your function.

```
drop function gosalesdw.factorial@
```

Note that if you try to invoke your function again, you will receive an error message similar to this:

```
No authorized routine named "FACTORIAL" of type "FUNCTION" having compatible arguments
was found.. SQLCODE=-440, SQLSTATE=42884, DRIVER=3.68.61

[State: 56098][Code: -727]: An error occurred during implicit system action type "2".
Information returned for the error includes SQLCODE "-440", SQLSTATE "42884" and
message tokens "FACTORIAL|FUNCTION".. SQLCODE=-727, SQLSTATE=56098, DRIVER=3.68.61
```

7.7. Incorporating FOR loops

As you might expect, Big SQL also supports FOR-DO loops in SQL-bodied UDFs. In this exercise, you'll create a function to calculate the sum of the top 5 sales for a given day.

__1. Create a scalar UDF named `gosalesdw.sum_sale_total_top_5`. Note that this UDF references the SLS_SALES_FACT table that you created in an earlier lab.

```
-- FOR-DO loop and a SELECT statement inside scalar UDF
-- Given order_day_key, returns sum of sale_total
```

```
-- for first 5 sales with given order_day_key. Order by sale_total
--------------------------------
-- Create UDF with FOR-DO loop and a SELECT statement inside
CREATE OR REPLACE FUNCTION gosalesdw.sum_sale_total_top_5(input_order_day_key INTEGER)
RETURNS DOUBLE
LANGUAGE SQL
READS SQL DATA
BEGIN ATOMIC
  DECLARE result DOUBLE;
  DECLARE counter INTEGER;
  SET result = 0;
  SET counter = 5;

  FOR v1 AS
    SELECT sale_total
    FROM sls_sales_fact
    WHERE order_day_key = input_order_day_key
    ORDER BY  sale_total DESC

  DO
    IF counter > 0
    THEN
      SET result = result + sale_total;
      SET counter = counter - 1;
    END IF;

  END FOR;

  RETURN result;
END @
```

__1. Review the logic of this function. Note that the FOR loop begins by retrieving SALE_TOTAL values from the SLS_SALES_FACT table based on the order key day provided as input. These results are ordered, and the DO block uses a counter to control the number of times it will add a SALE_TOTAL value to the result. In this example, that will occur 5 times.

__2. Finally, use this UDF to compute the sum of the top 5 sales on a specific order day key (20040112).

```
-- The output of this function call should be 925973.09000
VALUES (gosalesdw.sum_sale_total_top_5(20040112)) @
```

7.8. Creating a table UDF

Now that you've created several scalar UDFs, it's time to explore how you can create a simple UDF that will return a result set. Such UDFs are called table UDFs because they can return multiple columns and multiple rows.

In this lab, you will create a table UDF that returns information about the items sold on a given day input by the user. The result set will include information about the sales order, the quantity of items, the pre-

discounted sales price, and the final sales price (including tax and a discount). In doing so, your table UDF will call a scalar UDF you created previously: `new_final_price_v2`.

__1. Create a table UDF named `gosalesdw.sales_summary`. Note that this UDF references the SLS_SALES_FACT table that you created in an earlier lab.

```
-- Table UDF
-- given an order_day_key, returns some desired fields and
-- new_final_price for that order_day_key
--------------------------------

-- Create a simple table UDF
CREATE OR REPLACE FUNCTION gosalesdw.sales_summary(input_order_day_key INTEGER)
RETURNS TABLE(sales_order_key INTEGER, quantity INTEGER, sale_total DOUBLE,
new_final_price DOUBLE)
LANGUAGE SQL
READS SQL DATA
RETURN
  SELECT sales_order_key, quantity, sale_total, gosalesdw.new_final_price_v2(quantity,
unit_sale_price, 10,20,30, 8.75)
  FROM sls_sales_fact
  WHERE order_day_key = input_order_day_key
@
```

__2. Inspect the logic in this function. Note that it includes a READS SQL DATA clause (because the function SELECTs data from a table) and that the RETURNS clause specifies a TABLE with columns and data types. Towards the end of the function is the query that drives the result set that is returned. As mentioned earlier, this query invokes a scalar UDF that you created earlier.

__3. Invoke your table UDF in the FROM clause of a query, supplying an input parameter of `20040112` to your function for the order day key.

```
-- use it in the FROM clause
SELECT t1.*
FROM TABLE (gosalesdw.sales_summary(20040112)) AS t1
ORDER BY sales_order_key
FETCH FIRST 10 ROWS ONLY
@
```

__4. Inspect your output.

```
+-----------------+----------+-------------+-----------------+
| SALES_ORDER_KEY | QUANTITY |  SALE_TOTAL | NEW_FINAL_PRICE |
+-----------------+----------+-------------+-----------------+
|          100001 |      256 |  8624.64000 |      7503.43680 |
|          100002 |       92 |  9411.60000 |      7164.58050 |
|          100003 |      162 | 18032.22000 |     13727.02747 |
|          100004 |      172 |  6690.80000 |      5820.99600 |
|          100005 |       74 | 24747.82000 |     18839.27797 |
|          100006 |       90 |  6825.60000 |      5938.27200 |
|          100007 |      422 |  2532.00000 |      2478.19500 |
|          100008 |     3252 | 21170.52000 |     20720.64645 |
|          100009 |     1107 |  6376.32000 |      6240.82320 |
|          100010 |       88 | 10975.36000 |      8354.99280 |
+-----------------+----------+-------------+-----------------+
10 rows in results(first row: 0.447s; total: 0.450s)
```

As you might imagine, the bodies of table UDFs aren't limited to queries. Indeed, you can write table UDFs that contain IF/ELSE, WHILE/DO, FOR-DO, and many more constructs. Consult the BigInsights Knowledge Center for details.

7.9. Optional: Overloading UDFs and dropping UDFs

As you saw in an earlier exercise, you can drop UDFs with the DROP FUNCTION statement. In addition, you can create multiple UDFs with the same name (even in the same schema) if their input parameters differ enough so that Big SQL can identify which should be called during a query. Such UDFs are said to be "overloaded". When working with overloaded UDFs, you must use the DROP SPECIFIC FUNCTION statement to properly identify which UDF bearing the same name should be dropped.

In this lab, you'll explore the concepts of overloading functions and dropping a specific function. To keep things simple and focused on the topics at hand, the UDFs will be trivial – they will simply increment a supplied INTEGER or DOUBLE value by 1.

__1. Create a scalar UDF that increments an INTEGER value.

```
-- Create a scalar UDF
CREATE FUNCTION increment_by_one(p1 INT)
RETURNS INT
LANGUAGE SQL
SPECIFIC increment_by_one_int
RETURN p1 + 1 @
```

Note that the SPECIFIC clause provides a unique name for the function that we can later reference it when we need to drop this function.

__2. Create a scalar UDF that increments a DOUBLE value.

```
-- Create another scalar UDF with same name (but different specific name)
CREATE FUNCTION increment_by_one(p1 DOUBLE)
RETURNS DOUBLE
LANGUAGE SQL
```

```
SPECIFIC increment_by_one_double
RETURN p1 + 1 @
```

__3. Attempt to drop the `increment_by_one` function without referencing the specific name you included in each function.

```
-- If we try to drop the function using DROP FUNCTION statement,
-- Big SQL will throw Error : SQLCODE=-476, SQLSTATE=42725, because
-- Big SQL needs to know which function should be dropped
DROP FUNCTION increment_by_one@
```

Note that this statement will fail because Big SQL isn't certain which of the two `increment_by_one` functions you intended to drop.

__4. Drop the function that requires an INTEGER as its input parameter. Reference the function's specific name in a DROP SPECIFIC FUNCTION statement.

```
-- User must drop using specific name
DROP SPECIFIC FUNCTION increment_by_one_int@
```

__5. Now drop the remaining `increment_by_one` function. Since we only have 1 function by this name in this schema, we can issue a simple DROP FUNCTION statement:

```
-- Now we have only one function with this name, so we can use
-- simple DROP FUNCTION statement.
DROP FUNCTION increment_by_one@
```

What if you didn't include a SPECIFIC clause (i.e., a specific name) in your UDF definition? Big SQL will explicitly provide one, and you can query the system catalog tables to identify it. Let's explore that scenario.

__6. Create a simple scalar UDF again.

```
-- Create a UDF
CREATE FUNCTION increment_by_one(p1 INT)
RETURNS INT
LANGUAGE SQL
RETURN p1 + 1 @
```

__7. Create another scalar UDF with the same name (but different input parameter)

```
-- Create another scalar UDF with same name (but different input parm)
CREATE FUNCTION increment_by_one(p1 DOUBLE)
RETURNS DOUBLE
LANGUAGE SQL
RETURN p1 + 1 @
```

__8. Query the Big SQL catalog for specific names for these functions:

```
-- Query catalog for specific name:
SELECT ROUTINENAME, SPECIFICNAME, PARM_COUNT, RETURN_TYPENAME
FROM SYSCAT.ROUTINES
WHERE  ROUTINENAME = 'INCREMENT_BY_ONE' @
```

__9. Inspect the output, noting the different names assigned to your functions. (Your output may vary from that shown below.)

```
+------------------+--------------------+------------+-----------------+
| ROUTINENAME      | SPECIFICNAME       | PARM_COUNT | RETURN_TYPENAME |
+------------------+--------------------+------------+-----------------+
| INCREMENT_BY_ONE | SQL170531210041488 |          1 | INTEGER         |
| INCREMENT_BY_ONE | SQL170531210104189 |          1 | DOUBLE          |
+------------------+--------------------+------------+-----------------+
2 rows in results(first row: 0.043s; total: 0.046s)
```

__10. If desired, drop each of these UDFs. Remember that you will need to reference the specific name of the first UDF that you drop when you execute the DROP SPECIFIC FUNCTION statement.