IBM Training

Checkpoint solutions

1. List at least one of the data sources that Spark supports?
   - Structured data files
   - Tables in Hive
   - External databases
   - Existing RDDs
2. Which data source allows both Spark and Big SQL to work together?
   - The Hive warehouse
3. What is one reason why you would want to use Spark with your Big SQL tables?
   - Spark can provide additional analytical capabilities through the use of its libraries, including machine learning, streaming, graph computations, etc.

Checkpoint solutions

IBM Training

Demonstration 1

Using Spark operations on tables managed by Big SQL

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

- Work with data in Big SQL tables using the Spark shell.
- Create a Spark Schema RDD (resilient distributed dataset) from data in Big SQL tables.
- Query and extract data from Big SQL tables using Spark SQL.
- Expose data in Big SQL columns as a Spark MLlib data type (Vector) and invoke a simple MLlib statistical function over this data.

Demonstration 1: Using Spark operations on tables managed by Big SQL

Demonstration 1: Using Spark operations on tables managed by Big SQL

Purpose:
In this demonstration you will explore how Spark programmers can work with data managed by Big SQL. As you may know, Big SQL is a popular component of several IBM BigInsights offerings. It enables SQL professionals to query data stored in Hadoop using ISO SQL syntax; it also provides database monitoring, query federation, and other features. Apache Spark, part of IBM's Open Platform for Apache Hadoop and BigInsights, is a fast, general-purpose engine for processing Big Data, including data managed by Hadoop. Particularly appealing to many Spark programmers are built-in and third-party libraries for machine learning, streaming, SQL, and more.

Given the popularity of both Big SQL and Spark, it's reasonable to expect organizations to want to deploy and use both technologies. This demonstration introduces you to one way in which organizations can integrate these technologies, namely, by creating, populating, and manipulating Big SQL tables stored in HDFS directories or the Hive warehouse and then accessing data from these tables through Spark SQL and Spark MLlib.

Estimated time: 45 minutes

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

Services Password: **ibm2blue**

Task 1. Creating and populating your Big SQL sample tables.

Create two tables in the Hive warehouse using your default schema (which will be "bigsql" if you connected into your database as that user). The first table is part of the PRODUCT dimension and includes information about product lines in different languages. The second table is the sales FACT table, which tracks transactions (orders) of various products. The statements below create both tables in a TEXTFILE format. Within each row, fields are separated by tabs ("\t"). Furthermore, each line is terminated by new line character ("\n").

Both of these tables should have been created in one of the earlier demonstrations. Go back and create them now if you haven't done so already. You will be working with these two tables in the default *bigsql* schema.

```
sls_product_line_lookup
sls_sales_fact
```

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

Query the tables to verify that the expected number of rows was loaded into each table.

1. Execute each query below and compare the results with the number of rows specified in the comment line preceding each query.

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

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

Next you will create 1 externally managed Big SQL table - i.e., a table created over a user directory that resides outside of the Hive warehouse. This user directory will contain all the table's data in files. Creating such a table effectively layers a SQL schema over existing DFS data (or data that you may later upload into the target DFS directory).

2. Open a new terminal.
3. From the command line, issue this command to switch to the root user ID temporarily:

```
su root
```

4. Create a directory structure in your distributed file system for the source data file for the product dimension table. Ensure public read/write access to this directory structure. (If desired, alter the DFS information as appropriate for your environment.)

```
hdfs dfs -mkdir /user/bigsql_spark_lab
hdfs dfs -mkdir /user/bigsql_spark_lab/sls_product_dim
hdfs dfs -chmod -R 777 /user/bigsql_spark_lab
```

5. Upload the source data file (the Big SQL sample data file named GOSALESDW.SLS_PRODUCT_DIM.txt) into the target DFS directory. Change the local and DFS directories information below to match your environment.

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

6. List the contents of the DFS directory and verify that your sample data file is present.

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

7. Return to your Big SQL query execution environment (JSqsh or the Big SQL console).
8. Create an external Big SQL table for the sales product dimension (extern.sls_product_dim).
   Note that the LOCATION clause references the DFS directory into which you copied the sample data.

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

9. Verify that you can query the table.

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

Task 2. Setting up your Spark environment.

Now that you've created and populated the sample Big SQL tables required for this demonstration, it's time to experiment with accessing their data. In this module, you'll adjust the level of detail returned by the Spark shell so that only essential messages are returned. The Spark shell can be verbose, so suppressing informational messages will help you focus on the tasks at hand. Next, you'll launch the Spark shell.

1. Open up a new terminal, or use an existing one.
2. Switch to the **root** user, password **dalvm3**, if required.
3. Switch to the **spark** user.
4. As the *spark* user, change directories to Spark home:

```
cd /home/spark
```

5. Create a new file named **log4j.properties** using your favorite editor. Instructions in this demonstration are based on the vi editor.

```
vi log4j.properties
```

6. In vi, enter insert mode. Type

```
i
```

7. Cut and paste the following content into the file:

```
# Set everything to be logged to the console
log4j.rootCategory=ERROR, console
log4j.appender.console=org.apache.log4j.ConsoleAppender
log4j.appender.console.target=System.err
log4j.appender.console.layout=org.apache.log4j.PatternLa
yout
log4j.appender.console.layout.ConversionPattern=%d{yy/MM
/dd HH:mm:ss} %p %c{1}: %m%n

# Settings to quiet third party logs that are too
verbose
log4j.logger.org.eclipse.jetty=WARN
log4j.logger.org.eclipse.jetty.util.component.AbstractLi
feCycle=ERROR
log4j.logger.org.apache.spark.repl.SparkIMain$exprTyper=
INFO
log4j.logger.org.apache.spark.repl.SparkILoop$SparkILoop
Interpreter=INFO
```

8. To exit the file, press **Esc** and then enter.

```
:wq
```


9. Verify that your new log4j.properties file was successfully created and contains the right content.

```
more log4j.properties
```

10. Launch the Spark shell, which will enable you to issue Scala statements and expressions:

```
spark-shell
```

Task 3. Querying and manipulating Big SQL data through Spark.

In this task, you'll issue Scala commands and expressions from the Spark shell to retrieve Big SQL data. Specifically, you'll use Spark SQL to query data in Big SQL tables. You'll model your result sets from your queries as SchemaRDDs, a specific type of resilient distributed dataset (RDD). SchemaRDDs consist of Row objects and a schema describing each column in the row. For details on RDDs and SchemaRDDs, visit the Spark web site.

1. From the Spark shell, establish a Hive context named sqlContext:

```
val sqlContext = new
org.apache.spark.sql.hive.HiveContext(sc)
```

```
scala> val sqlContext = new org.apache.spark.sql.hive.HiveContext(sc)
sqlContext: org.apache.spark.sql.hive.HiveContext = org.apache.spark.sql.hive.Hi
veContext@c349c1d
```

The Hive context enables you to find tables in the Hive meta store (HCatalog) and issue HiveQL queries against these tables.

2. Experiment with querying a Big SQL table stored in the Hive warehouse.

```
sqlContext.sql("select * from bigsql.sls_product_lookup
limit 5").collect().foreach(println)
```

Let's examine this statement briefly. Using the Hive context established previously, we issue a simple query (using HiveQL syntax) to retrieve 5 rows from the *sls_product_lookup* table in the *bigsql* Hive schema. We call other Spark functions to collect the result and print each record.

```
scala> sqlContext.sql("select * from bigsql.sls_product_lookup limit 5").collect
().foreach(println)
[1110,CS,Vak na vodu Kuchtík,Lehký, skladný vak na tekutiny. Široké hrdlo usnadn
í plnění. Objem 10 litrů.]
[1110,DA,Sahara Vandtaske,Sammentrykkelig letvægtstaske til væsker. Bred åbning
til nem påfyldning. Indeholder 10 liter.]
[1110,DE,TrailChef Wasserbeutel,Leichter, zusammenfaltbarer Beutel zum einfachen
 Transport von Flüssigkeiten. Breite Öffnung zum einfachen Einfüllen. Fassungsve
rmögen 10 Liter.]
[1110,EL,Δοχείο υγρών Μαρμίτα Σεφ,Ελαφρύ, πτυσσόμενο δοχείο για εύκολη μεταφορά
υγρών.  Ευρύ στόμιο για εύκολη πλήρωση.  Χωρητικότητα 10 λίτρα.]
[1110,EN,TrailChef Water Bag,Lightweight, collapsible bag to carry liquids easil
y.  Wide mouth for easy filling.  Holds 10 liters.]
```

3. Issue a similar query against an externally managed Big SQL table (i.e., a Big SQL table created over an HDFS directory.

```
sqlContext.sql("select * from extern.sls_product_dim
limit 5").collect().foreach(println)
```

```
scala> sqlContext.sql("select * from extern.sls_product_dim limit 5").collect().
foreach(println)
[30001,991,951,951,1110,1,1,908,808,701,701,P01CE1CG1.jpg,1995-02-15 00:00:00.0,
null]
[30002,991,951,951,2110,2,2,906,807,701,701,P02CE1CG1.jpg,1995-02-15 00:00:00.0,
null]
[30003,991,951,951,3110,3,3,924,825,701,701,P03CE1CG1.jpg,1995-02-15 00:00:00.0,
null]
[30004,991,951,951,4110,4,4,923,804,701,701,P04CE1CG1.jpg,1995-02-15 00:00:00.0,
null]
[30005,991,951,951,5110,5,5,923,823,701,701,P05CE1CG1.jpg,1995-02-15 00:00:00.0,
null]
```

4. To make the result set readily available for further processing in Spark, create a SchemaRDD based on a query. In this case, we'll query the externally managed table.

```
val prodDim = sqlContext.sql("select * from
extern.sls_product_dim")
```

```
scala> val prodDim = sqlContext.sql("select * from extern.sls_product_dim")
prodDim: org.apache.spark.sql.SchemaRDD =
SchemaRDD[0] at RDD at SchemaRDD.scala:108
== Query Plan ==
== Physical Plan ==
HiveTableScan [product_key#0,product_line_code#1,product_type_key#2,product_type
_code#3,product_number#4,base_product_key#5,base_product_number#6,product_color_
code#7,product_size_code#8,product_brand_key#9,product_brand_code#10,product_ima
ge#11,introduction_date#12,discontinued_date#13], (MetastoreRelation extern, sls
_product_dim, None), None
```

5. Extract the value of the fifth column (the product number column) from the fourth row in the RDD. Since array elements begin at 0, issue the following statement:

```
prodDim.collect()(3).getInt(4)
```

```
scala> prodDim.collect()(3).getInt(4)
res3: Int = 4110
```

Let's step through the logic of this statement briefly. The *collect()* function returns an array containing all elements in the referenced RDD. Since array indexing begins at 0, specifying (3) indicates that we want to work only with the fourth row. The getInt(4) function retrieves the fifth column for this row. Given the Big SQL table definition, this is the value for the PRODUCT_NUMBER column (an integer).

6. Experiment with using the Spark map() function to create a new RDD named prodNum based on prodDim. In this case, the transformation performed by the map() function will be simple – it will simply extract the product numbers from prodDim. (Recall that product numbers reside in the fifth field of prodDim and that array indexing begins at zero.)

```
val prodNum = prodDim.map (row => row.getInt(4))
```

```
scala> val prodNum = prodDim.map (row => row.getInt(4))
prodNum: org.apache.spark.rdd.RDD[Int] = MappedRDD[5] at map at <console>:16
```

7. Use the count() function to verify that prodNum contains 274 records.

```
prodNum.count()
```

```
scala> prodNum.count()
res4: Long = 274
```

8. Optionally, validate these results through Big SQL.

   a. Open a second terminal window and launch JSqsh.

   b. Connect to your Big SQL database.

   c. Count the number of product number records in the sls_product_dim table.

```
select count(product_number) from
extern.sls_product_dim;
```

   d. Verify that 274 rows are present. Note that this matches the count of the prodNum RDD you defined in Scala.

Task 4. Accessing Big SQL data with Spark MLlib.

Now that you understand how to use Spark SQL to work with data managed by Big SQL tables, let's explore how to use other Spark technologies to manipulate this data. In this task, you'll transform Big SQL data into a data type commonly used with Spark's machine learning library (MLlib). Then you'll invoke a simple MLlib function on that data.

1. If necessary, return to the Spark shell that you launched in the previous task.
2. Import Spark classes that you'll be using shortly.

```
import org.apache.spark.mllib.linalg.Vectors

import
org.apache.spark.mllib.stat.{MultivariateStatisticalSumm
ary, Statistics}
```

```
scala> import org.apache.spark.mllib.linalg.Vectors
import org.apache.spark.mllib.linalg.Vectors

scala> import org.apache.spark.mllib.stat.{MultivariateStatisticalSummary, Stati
stics}
import org.apache.spark.mllib.stat.{MultivariateStatisticalSummary, Statistics}
```

3. Create a SchemaRDD named saleFacts based on data in the bigsql.sls_sales_fact table.

```
val saleFacts = sqlContext.sql("select * from
bigsql.sls_sales_fact")
```

```
scala> val saleFacts = sqlContext.sql("select * from bigsql.sls_sales_fact")
saleFacts: org.apache.spark.sql.SchemaRDD =
SchemaRDD[80] at RDD at SchemaRDD.scala:108
== Query Plan ==
== Physical Plan ==
HiveTableScan [order_day_key#271,organization_key#272,employee_key#273,retailer_
key#274,retailer_site_key#275,product_key#276,promotion_key#277,order_method_key
#278,sales_order_key#279,ship_day_key#280,close_day_key#281,quantity#282,unit_co
st#283,unit_price#284,unit_sale_price#285,gross_margin#286,sale_total#287,gross_
profit#288], (MetastoreRelation bigsql, sls_sales_fact, None), None
```

4. Count the records in saleFacts, verifying that 446023 are present.

```
saleFacts.count()
```

5. Create a Vector containing data about sales totals and gross profits, and map this into a new RDD named subset.

```
val subset = saleFacts.map {row =>
Vectors.dense(row.getDouble(16),row.getDouble(17))}
```

6. Run basic statistical functions over this data.

```
val stats = Statistics.colStats(subset)
```

This operation may take several minutes to complete, depending on your machine resources. If the operation appears to be hung, press Enter on your keyboard to see if it's completed.

7. Print various statistical results.

```
println(stats.mean)

println(stats.variance)

println(stats.max)
```

```
scala> println(stats.mean)
[10507.92396098396,4315.550979837396]

scala> println(stats.variance)
[3.29316420180777E8,5.02623271828062E7]

scala> println(stats.max)
[363575.08,163736.73]
```

8. Optionally, validate one of these statistical results through Big SQL.

   a. If necessary, open a terminal window and launch JSqsh.

   b. Connect to your Big SQL database.

   c. Issue these queries to determine the maximum sales total and maximum gross profit values in the bigsql.sls_sales_fact table:

```
select max(sale_total) from  bigsql.sls_sales_fact;

select max(gross_profit) from bigsql.sls_sales_fact;
```

```
[bdvs1052.svl.ibm.com][bigsql] 1> select max(sale_total) from  bigsql.sls_sales_fact;
+--------------+
|            1 |
+--------------+
| 363575.08000 |
+--------------+
1 row in results(first row: 0.44s; total: 0.44s)
[bdvs1052.svl.ibm.com][bigsql] 1> select max(gross_profit) from bigsql.sls_sales_fact;
+--------------+
|            1 |
+--------------+
| 163736.73000 |
+--------------+
1 row in results(first row: 0.47s; total: 0.47s)
```

Note that these values equal the results computed by the Spark MLlib function that you invoked for the maximum values in this data set.

In this lab, you explored one way of using Spark to work with data in Big SQL tables stored in the Hive warehouse or DFS directories. From the Spark shell, you established a Hive context, queried Big SQL tables using Hive's query syntax, performed basic Spark transactions and actions on the data, and even applied a simple statistical function from Spark MLlib on the data.

Results:
In this demonstration you explored how Spark programmers can work with data managed by Big SQL. You were introduced you to one way in which organizations can integrate these technologies, namely, by creating, populating, and manipulating Big SQL tables stored in HDFS directories or the Hive warehouse and then accessing data from these tables through Spark SQL and Spark MLlib.