Free Questions for CCA175 by dumpshq

Shared by Huff on 06-06-2022

For More Free Questions and Preparation Resources

Check the Links on Last Page

Question 1

Question Type: MultipleChoice

Problem Scenario 92 : You have been given a spark scala application, which is bundled in jar named hadoopexam.jar.

Your application class name is com.hadoopexam.MyTask

You want that while submitting your application should launch a driver on one of the cluster node.

Please complete the following command to submit the application.

spark-submit XXX -master yarn \

YYY SSPARK HOME/lib/hadoopexam.jar 10

Options:

A- Solution

XXX: -class com.hadoopexam.MyTask

B- Solution

XXX: -class com.hadoopexam.MyTask

YYY : --deploy-mode cluster

Answer:

B

Question 2

Question Type: MultipleChoice

Problem Scenario 96 : Your spark application required extra Java options as below. -XX:+PrintGCDetails-XX:+PrintGCTimeStamps

Please replace the XXX values correctly

./bin/spark-submit --name "My app" --master local[4] --conf spark.eventLog.enabled=talse --conf XXX hadoopexam.jar

Options:

A- Solution

XXX: Mspark.executoi\extraJavaOptions=-XX:+PrintGCDetails -XX:+PrintGCTimeStamps'

Notes: ./bin/spark-submit \

--class <maln-class>

--master <master-url> \

--deploy-mode <deploy-mode> \

-conf <key>=<value> \

... # other options

\
[application-arguments]
Here, conf is used to pass the Spark related contigs which are required for the application to run like any specific property(executor memory) or if you want to override the default property which is set in Spark-default.conf.

B- Solution
XXX: Mspark.executoi\extraJavaOptions=-XX:+PrintGCDetails -XX:+PrintGCTimeStamps'
Notes: ./bin/spark-submit \
\
[application-arguments]
Here, conf is used to pass the Spark related contigs which are required for the application to run like any specific property(executor memory) or if you want to override the default property which is set in Spark-default.conf.

Answer:

A

Question 3

Question Type: **MultipleChoice**

Problem Scenario 95 : You have to run your Spark application on yarn with each executor Maximum heap size to be 512MB and Number of processorcores to allocate on each executor will be 1 and Your main application required three values as input arguments V1 V2 V3.

Please replace XXX, YYY, ZZZ

./bin/spark-submit -class com.hadoopexam.MyTask --master yarn-cluster--num-executors 3 --driver-memory 512m XXX YYY lib/hadoopexam.jarZZZ

Options:

A- Solution

XXX: -executor-memory 512m YYY: -executor-cores 1

ZZZ : V1 V2 V3

Notes : spark-submit on yarn options Option Description

archives Comma-separated list of archives to be extracted into the working directory of each executor. The path must be globally visible inside your cluster; see Advanced Dependency Management.

executor-cores Number of processor cores to allocate on each executor. Alternatively, you can use the spark.executor.cores property, executor-memory Maximum heap size to allocate to each executor. Alternatively, you can use the spark.executor.memory-property. num-executors Total number of YARN containers to allocate for this application. Alternatively, you can use the spark.executor.instances property. queue YARN queue to submit to. For more information, see Assigning Applications and Queries to Resource Pools. Default: default.

B- Solution

XXX: -executor-memory 510m YYY: -executor-cores 1

ZZZ : V2 V6 V1

Notes : spark-submit on yarn options Option Description

archives Comma-separated list of archives to be extracted into the working directory of each executor. The path must be globally visible inside your cluster; see Advanced Dependency Management.

executor-cores Number of processor cores to allocate on each executor. Alternatively, you can use the spark.executor.cores property, executor-memory Maximum heap size to allocate to each executor. Alternatively, you can use the spark.executor.memory-property. num-executors Total number of YARN containers to allocate for this application.

Answer:

A

Question 4

Question Type: **MultipleChoice**

Problem Scenario 89 : You have been given below patient data in csv format,

patientID,name,dateOfBirth,lastVisitDate

1001,Ah Teck,1991-12-31,2012-01-20

1002,Kumar,2011-10-29,2012-09-20

1003,Ali,2011-01-30,2012-10-21

Accomplish following activities.

1. Find all the patients whose lastVisitDate between current time and '2012-09-15'

2. Find all the patients who born in 2011

3. Find all the patients age

4. List patients whose last visited more than 60 days ago

5. Select patients 18 years old or younger

Options:

A- Solution :

Step 1:

```
hdfs dfs -mkdir sparksql3
hdfs dfs -put patients.csv sparksql3/
```

Step 2 : Now in spark shell

```
// SQLContext entry point for working with structured data
val sqlContext = neworg.apache.spark.sql.SQLContext(sc)
// this is used to implicitly convert an RDD to a DataFrame.
import sqlContext.implicits._
// Import Spark SQL data types and Row.
import org.apache.spark.sql._
// load the data into a new RDD
val patients = sc.textFilef'sparksqIS/patients.csv')
// Return the first element in this RDD
patients.first()
//define the schema using a case class
```

```
case class Patient(patientid: Integer, name: String, dateOfBirth:String , lastVisitDate: String)
// create an RDD of Product objects
val patRDD = patients.map(_.split(M,M)).map(p => Patient(p(0).tolnt,p(1),p(2),p(3)))
patRDD.first()
patRDD.count(}
// change RDD of Product objects to a DataFrame val patDF = patRDD.toDF()
// register the DataFrame as a temp table patDF.registerTempTable('patients'}
// Select data from table
val results = sqlContext.sql(.....SELECT* FROM patients '.....)
// display dataframe in a tabular format
results.show()
//Find all the patients whose lastVisitDate between current time and '2012-09-15'
val results = sqlContext.sql(.....SELECT * FROM patients WHERE TO_DATE(CAST(UNIX_TIMESTAMP(lastVisitDate, 'yyyy-MM-dd')
AS TIMESTAMP))BETWEEN '2012-09-15' AND current_timestamp() ORDER BY lastVisitDate.....)
results.showQ
/.Find all the patients who born in 2011
results. showQ;
-- Select patients 18 years old or younger
SELECT' FROM patients WHERE TO_DATE(CAST(UNIXJTlMESTAMP(dateOfBirth, 'yyyy-MM-dd') AS TIMESTAMP}) >
DATE_SUB(current_date(),INTERVAL 18 YEAR);
val results = sqlContext.sql(.....SELECT' FROM patients WHERE TO_DATE(CAST(UNIX_TIMESTAMP(dateOfBirth, 'yyyy-MM--dd') AS
TIMESTAMP)) > DATE_SUB(current_date(), T8*365).....);
results. showQ;
val results = sqlContext.sql(.....SELECT DATE_SUB(current_date(), 18*365) FROM patients.....);
results.show();
```

B- Solution :

Step 1:

```
hdfs dfs -mkdir sparksql3
hdfs dfs -put patients.csv sparksql3/
```

Step 2 : Now in spark shell

```
// SQLContext entry point for working with structured data
val sqlContext = neworg.apache.spark.sql.SQLContext(sc)
// this is used to implicitly convert an RDD to a DataFrame.
import sqlContext.implicits._
// Import Spark SQL data types and Row.
import org.apache.spark.sql._
// load the data into a new RDD
val patients = sc.textFilef'sparksqIS/patients.csv')
// Return the first element in this RDD
patients.first()
//define the schema using a case class
case class Patient(patientid: Integer, name: String, dateOfBirth:String , lastVisitDate: String)
// create an RDD of Product objects
val patRDD = patients.map(_.split(M,M)).map(p => Patient(p(0).tolnt,p(1),p(2),p(3)))
patRDD.first()
patRDD.count{}
// change RDD of Product objects to a DataFrame val patDF = patRDD.toDF()
// register the DataFrame as a temp table patDF.registerTempTable('patients'}
// Select data from table
val results = sqlContext.sql(.....SELECT* FROM patients '.....)
// display dataframe in a tabular format
```

```
results.show()
//Find all the patients whose lastVisitDate between current time and '2012-09-15'
val results = sqlContext.sql(.....SELECT * FROM patients WHERE TO_DATE(CAST(UNIX_TIMESTAMP(lastVisitDate, 'yyyy-MM-dd') AS TIMESTAMP))BETWEEN '2012-09-15' AND current_timestamp() ORDER BY lastVisitDate.....)
results.showQ
/.Find all the patients who born in 2011
val results = sqlContext.sql(.....SELECT * FROM patients WHERE YEAR(TO_DATE(CAST(UNIXJTlMESTAMP(dateOfBirth, 'yyyy-MM-dd') AS TIMESTAMP))) = 2011 .....)
results. show()
//Find all the patients age
val results = sqlContext.sql(.....SELECT name, dateOfBirth, datediff(current_date(), TO_DATE(CAST(UNIX_TIMESTAMP(dateOfBirth, 'yyyy-MM-dd')AS TlMESTAMP}}}/365 AS age
FROM patients
Mini >
results.show()
//List patients whose last visited more than 60 days ago
-- List patients whose last visited more than 60 days ago
val results = sqlContext.sql(.....SELECT name, lastVisitDate FROM patients WHERE datediff(current_date(), TO_DATE(CAST(UNIX_TIMESTAMP[lastVisitDate, 'yyyy-MM-dd') AS T1MESTAMP))) > 60.....);
results. showQ;
-- Select patients 18 years old or younger
SELECT' FROM patients WHERE TO_DATE(CAST(UNIXJTlMESTAMP(dateOfBirth, 'yyyy-MM-dd') AS TIMESTAMP}) > DATE_SUB(current_date(),INTERVAL 18 YEAR);
val results = sqlContext.sql(.....SELECT' FROM patients WHERE TO_DATE(CAST(UNIX_TIMESTAMP(dateOfBirth, 'yyyy-MM--dd') AS TIMESTAMP)) > DATE_SUB(current_date(), T8*365).....);
results. showQ;
```

val results = sqlContext.sql(.....SELECT DATE_SUB(current_date(), 18*365) FROM patients.....);
results.show();

Answer:

B

Question 5

Question Type: **MultipleChoice**

Problem Scenario 88 : You have been given below three files

product.csv (Create this file in hdfs)

productID,productCode,name,quantity,price,supplierid

1001,PEN,Pen Red,5000,1.23,501

1002,PEN,Pen Blue,8000,1.25,501

1003,PEN,Pen Black,2000,1.25,501

1004,PEC,Pencil 2B,10000,0.48,502

1005,PEC,Pencil 2H,8000,0.49,502

1006,PEC,Pencil HB,0,9999.99,502

2001,PEC,Pencil 3B,500,0.52,501

2002,PEC,Pencil 4B,200,0.62,501

2003,PEC,Pencil 5B,100,0.73,501

2004,PEC,Pencil 6B,500,0.47,502

supplier.csv

supplierid,name,phone

501,ABC Traders,88881111

502,XYZ Company,88882222

503,QQ Corp,88883333

products_suppliers.csv

productID,supplierID

2001,501

2002,501

2003,501

2004,502

2001,503

Now accomplish all the queries given in solution.

1. It is possible that, same product can be supplied by multiple supplier. Now find each product, its price according to each supplier.

2. Find all the supllier name, who are supplying 'Pencil 3B'

3. Find all the products , which are supplied by ABC Traders.

Options:

A- Solution :
Step 1 : It is possible that, same product can be supplied by multiple supplier. Now find each product, its price according to each supplier.
val results = sqlContext.sql(.....SELECT products.name AS Product Name', price, suppliers.name AS Supplier Name'
FROM products_suppliers
JOIN products ON products_suppliers.productID = products.productID JOIN suppliers ON products_suppliers.supplierID = suppliers.supplierID
null t
results.show()
Step 2 : Find all the supllier name, who are supplying 'Pencil 3B'
val results = sqlContext.sql(.....SELECT p.name AS 'Product Name', s.name AS 'Supplier Name'
FROM products_suppliers AS ps
Step 3 : Find all the products , which are supplied by ABC Traders.

val results = sqlContext.sql(.....SELECT p.name AS 'Product Name', s.name AS 'Supplier Name'
FROM products AS p, products_suppliers AS ps, suppliers AS s WHERE p.productID = ps.productID AND ps.supplierlD = s.supplierlD
AND s.name = 'ABC Traders'.....)
results. show()

B- Solution :

Step 1 : It is possible that, same product can be supplied by multiple supplier. Now find each product, its price according to each supplier.

val results = sqlContext.sql(.....SELECT products.name AS Product Name', price, suppliers.name AS Supplier Name'
FROM products_suppliers
JOIN products ON products_suppliers.productlD = products.productID JOIN suppliers ON products_suppliers.supplierlD = suppliers.supplierlD
null t
results.show()

Step 2 : Find all the supllier name, who are supplying 'Pencil 3B'

val results = sqlContext.sql(.....SELECT p.name AS 'Product Name', s.name AS 'Supplier Name'
FROM products_suppliers AS ps
JOIN products AS p ON ps.productID = p.productID
JOIN suppliers AS s ON ps.supplierlD = s.supplierlD
WHERE p.name = 'Pencil 3B",M)
results.show()

Step 3 : Find all the products , which are supplied by ABC Traders.

val results = sqlContext.sql(.....SELECT p.name AS 'Product Name', s.name AS 'Supplier Name'
FROM products AS p, products_suppliers AS ps, suppliers AS s WHERE p.productID = ps.productID AND ps.supplierlD = s.supplierlD
AND s.name = 'ABC Traders'.....)
results. show()

Answer:

B

Question 6

Question Type: **MultipleChoice**

Problem Scenario 87 : You have been given below three files

product.csv (Create this file in hdfs)

productID,productCode,name,quantity,price,supplierid

1001,PEN,Pen Red,5000,1.23,501

1002,PEN,Pen Blue,8000,1.25,501

1003,PEN,Pen Black,2000,1.25,501

1004,PEC,Pencil 2B,10000,0.48,502

1005,PEC,Pencil 2H,8000,0.49,502

1006,PEC,Pencil HB,0,9999.99,502

2001,PEC,Pencil 3B,500,0.52,501

2002,PEC,Pencil 4B,200,0.62,501

2003,PEC,Pencil 5B,100,0.73,501

2004,PEC,Pencil 6B,500,0.47,502

supplier.csv

supplierid,name,phone

501,ABC Traders,88881111

502,XYZ Company,88882222

503,QQ Corp,88883333

products_suppliers.csv

productID,supplierID

2001,501

2002,501

2003,501

2004,502

2001,503

Now accomplish all the queries given in solution.

Select product, its price , its supplier name where product price is less than 0.6 using SparkSQL

Options:

A- Solution :

Step 1:

```
hdfs dfs -mkdir sparksql2
hdfs dfs -put product.csv sparksq!2/
hdfs dfs -put supplier.csv sparksql2/
hdfs dfs -put products_suppliers.csv sparksql2/
```

Step 2 : Now in spark shell

```
// this Is used to Implicitly convert an RDD to a DataFrame.
import sqlContext.implicits._
// Import Spark SQL data types and Row.
import org.apache.spark.sql._
// load the data into a new RDD
val products = sc.textFile('sparksql2/product.csv')
val supplier = sc.textFileC'sparksq^supplier.csv')
val prdsup = sc.textFile('sparksql2/products_suppliers.csv'}
// Return the first element in this RDD
products.fi rst()
supplier.first{).
prdsup.first()
```

```
//define the schema using a case class
case class Product(productid: Integer, code: String, name: String, quantity:Integer, price: Float, supplierid:Integer)
case class Suplier(supplierid: Integer, name: String, phone: String)
case class PRDSUP(productid: Integer.supplierid: Integer)
// create an RDD of Product objects
val prdRDD = products.map(_.split('\')).map(p => Product(p(0).tolnt,p(1),p(2),p(3).tolnt,p(4).toFloat,p(5).toint))
val supRDD = supplier.map(_.split(',')).map(p => Suplier(p(0).tolnt,p(1),p(2)))
val prdsupRDD = prdsup.map(_.split(',')).map(p => PRDSUP(p(0).tolnt,p(1}.tolnt}}
prdRDD.first()
prdRDD.count()
supRDD.first() supRDD.count()
prdsupRDD.first() prdsupRDD.count{}
// change RDD of Product objects to a DataFrame
val prdDF = prdRDD.toDF()
val supDF = supRDD.toDF()
val prdsupDF = prdsupRDD.toDF()
// register the DataFrame as a temp table prdDF.registerTempTablef'products')
supDF.registerTempTablef'suppliers')
prdsupDF.registerTempTablef'productssuppliers'}
//Select product, its price , its supplier name where product price is less than 0.6
val results = sqlContext.sql(.....SELECT products.name, price, suppliers.name as sup_name FROM products JOIN suppliers ON
products.supplierlD= suppliers.supplierlD WHERE price < 0.6.....]
results. show()
```

B- Solution :

Step 1:

```
hdfs dfs -mkdir sparksql2
hdfs dfs -put product.csv sparksq!2/
hdfs dfs -put supplier.csv sparksql2/
hdfs dfs -put products_suppliers.csv sparksql2/
Step 2 : Now in spark shell
// this Is used to Implicitly convert an RDD to a DataFrame.
import sqlContext.implicits._
// Import Spark SQL data types and Row.
import org.apache.spark.sql._
// load the data into a new RDD
val products = sc.textFile('sparksql2/product.csv')
val supplier = sc.textFileC'sparksq^supplier.csv')
val prdsup = sc.textFile('sparksql2/products_suppliers.csv'}
// Return the first element in this RDD
products.fi rst()
supplier.first{).
prdsup.first()
//define the schema using a case class
case class Product(productid: Integer, code: String, name: String, quantity:lnteger, price: Float, supplierid:lnteger)
case class Suplier(supplierid: Integer, name: String, phone: String)
case class PRDSUP(productid: Integer.supplierid: Integer)
// create an RDD of Product objects
val prdRDD = products.map(_.split('\')).map(p => Product(p(0).tolnt,p(1),p(2),p(3).tolnt,p(4).toFloat,p(5).toint))
val supRDD = supplier.map(_.split(',')).map(p => Suplier(p(0).tolnt,p(1),p(2)))
val prdsupRDD = prdsup.map(_.split(',')).map(p => PRDSUP(p(0).tolnt,p(1}.tolnt}}
val prdsupDF = prdsupRDD.toDF()
```

// register the DataFrame as a temp table prdDF.registerTempTablef'products')
supDF.registerTempTablef'suppliers')
prdsupDF.registerTempTablef'productssuppliers'}
//Select product, its price , its supplier name where product price is less than 0.6
val results = sqlContext.sql(.....SELECT products.name, price, suppliers.name as sup_name FROM products JOIN suppliers ON products.supplierlD= suppliers.supplierlD WHERE price < 0.6.....]
results. show()

Answer:

A

Question 7

Question Type: **MultipleChoice**

Problem Scenario 86 : In Continuation of previous question, please accomplish following activities.

1. Select Maximum, minimum, average , Standard Deviation, and total quantity.

2. Select minimum and maximum price for each product code.

3. Select Maximum, minimum, average , Standard Deviation, and total quantity for each product code, hwoever make sure Average and Standarddeviation will have maximum two decimal values.

4. Select all the product code and average price only where product count is more than or equal to 3.

5. Select maximum, minimum , average and total of all the products for each code. Also produce the same across all the products.

Options:

A- Solution :

Step 1 : Select Maximum, minimum, average , Standard Deviation, and total quantity.

val results = sqlContext.sql('.....SELECT MAX(price) AS MAX , MIN(price) AS MIN , AVG(price) AS Average, STD(price) AS STD, SUM(quantity) AStotal_products FROM products.....)

results. showQ

Step 2 : Select minimum and maximum price for each product code.

val results = sqlContext.sql(.....SELECT code, MAX(price) AS Highest Price', MIN(price) AS Lowest Price'

FROM products GROUP BY code.....)

results. showQ

Step 3 : Select Maximum, minimum, average , Standard Deviation, and total quantity for each product code, hwoever make sure Average andStandard deviation will have maximum two decimal values.

val results = sqlContext.sql(.....SELECT code, MAX(price), MIN(price),

CAST(AVG(price} AS DECIMAL(7,2)) AS Average', CAST(STD(price) AS DECIMAL(7,2)) AS 'Std Dev\ SUM(quantity) FROM products GROUP BY code.....)

results. showQ

Step 4 : Select all the product code and average price only where product count is more than or equal to 3.

val results = sqlContext.sql(.....SELECT code AS Product Code',

COUNTf) AS Count',

CAST(AVG(price) AS DECIMAL(7,2)) AS Average' FROM products GROUP BY code

HAVING Count >=3'M') results. showQ
Step 5 : Select maximum, minimum , average and total of all the products for each code. Also produce the same across all the products.
val results = sqlContext.sql('"SELECT
code,
MAX(price),
MIN(pnce),
CAST(AVG(price) AS DECIMAL(7,2)) AS Average',
SUM(quantity)-
FROM products
GROUP BY code
WITH ROLLUP"')
results. show()

B- Solution :
Step 1 : Select Maximum, minimum, average , Standard Deviation, and total quantity.
val results = sqlContext.sql('.....SELECT MAX(price) AS MAX , MIN(price) AS MIN , AVG(price) AS Average, STD(price) AS STD,
SUM(quantity) AStotal_products FROM products.....)
results. showQ
Step 2 : Select minimum and maximum price for each product code.
val results = sqlContext.sql(.....SELECT code, MAX(price) AS Highest Price', MIN(price) AS Lowest Price'
FROM products GROUP BY code.....)
results. showQ
Step 3 : Select Maximum, minimum, average , Standard Deviation, and total quantity for each product code, hwoever make sure
Average andStandard deviation will have maximum two decimal values.
val results = sqlContext.sql(.....SELECT code, MAX(price), MIN(price),
GROUP BY code.....)

results. showQ
Step 4 : Select all the product code and average price only where product count is more than or equal to 3.
val results = sqlContext.sql(.....SELECT code AS Product Code',
COUNTf) AS Count',
CAST(AVG(price) AS DECIMAL(7,2)) AS Average' FROM products GROUP BY code
HAVING Count >=3'M') results. showQ
Step 5 : Select maximum, minimum , average and total of all the products for each code. Also produce the same across all the products.
val results = sqlContext.sql('"SELECT
code,
MAX(price),
WITH ROLLUP"')
results. show()

Answer:

A

Question 8

Question Type: **MultipleChoice**

Problem Scenario 85 : In Continuation of previous question, please accomplish following activities.

1. Select all the columns from product table with output header as below. productID AS ID

code AS Code name AS Description price AS 'Unit Price'

2. Select code and name both separated by ' -' and header name should be Product Description'.

3. Select all distinct prices.

4. Select distinct price and name combination.

5. Select all price data sorted by both code and productID combination.

6. count number of products.

7. Count number of products for each code.

Options:

A- Solution :
Step 1 : Select all the columns from product table with output header as below. productID AS ID code AS Code name AS Description price AS 'Unit Price'
val results = sqlContext.sql(.....SELECT productID AS ID, code AS Code, name AS Description, price AS Unit Price' FROM products ORDER BY ID'"
results.show()
Step 2 : Select code and name both separated by ' -' and header name should be 'Product Description.
val results = sqlContext.sql(.....SELECT CONCAT(code,' -', name) AS Product Description, price FROM products"')
results.showQ
Step 3 : Select all distinct prices.
val results = sqlContext.sql(.....SELECT DISTINCT price AS Distinct Price' FROM products.....)

results.show()
Step 4 : Select distinct price and name combination.
val results = sqlContext.sql(.....SELECT DISTINCT price, name FROM products''')
results. showQ
Step 5 : Select all price data sorted by both code and productID combination.
val results = sqlContext.sql('.....SELECT' FROM products ORDER BY code, productID'.....)
results.show()
Step 6 : count number of products.
val results = sqlContext.sql(.....SELECT COUNT(') AS 'Count' FROM products.....)
results.show()
Step 7 : Count number of products for each code.
val results = sqlContext.sql(.....SELECT code, COUNT('} FROM products GROUP BY code.....)
results. showQ
val results = sqlContext.sql(.....SELECT code, COUNT('} AS count FROM products GROUP BY code ORDER BY count DESC.....)
results. showQ

B- Solution :
Step 1 : Select all the columns from product table with output header as below. productID AS ID code AS Code name AS Description price AS 'Unit Price'
val results = sqlContext.sql(.....SELECT productID AS ID, code AS Code, name AS Description, price AS Unit Price' FROM products ORDER BY ID'''
results.show()
Step 2 : Select code and name both separated by ' -' and header name should be 'Product Description.
val results = sqlContext.sql(.....SELECT CONCAT(code,' -', name) AS Product Description, price FROM products''')
results.showQ
Step 3 : Select all distinct prices.

val results = sqlContext.sql(.....SELECT DISTINCT price AS Distinct Price' FROM products.....)
results.show()
Step 4 : count number of products.
val results = sqlContext.sql(.....SELECT COUNT(') AS 'Count' FROM products.....)
results.show()
Step 5 : Count number of products for each code.
val results = sqlContext.sql(.....SELECT code, COUNT('} FROM products GROUP BY code.....)
results. showQ
val results = sqlContext.sql(.....SELECT code, COUNT('} AS count FROM products GROUP BY code ORDER BY count DESC.....)
results. showQ

Answer:

A

Question 9

Question Type: **MultipleChoice**

Problem Scenario 84 : In Continuation of previous question, please accomplish following activities.

1. Select all the products which has product code as null

2. Select all the products, whose name starts with Pen and results should be order by Price descending order.

3. Select all the products, whose name starts with Pen and results should be order by Price descending order and quantity ascending order.

4. Select top 2 products by price

Options:

A- Solution :

Step 1 : Select all the products which has product code as null

val results = sqlContext.sql(.....SELECT' FROM products WHERE code IS NULL.....)

results. showQ

val results = sqlContext.sql(.....SELECT * FROM products WHERE code = NULL ',,M)

results.showQ

Step 2 : Select all the products , whose name starts with Pen and results should be order by Price descending order. val results = sqlContext.sql(.....SELECT * FROM products WHERE name LIKE 'Pen %' ORDER BY price DESC.....)

results. showQ

Step 3 : Select all the products , whose name starts with Pen and results should be order by Price descending order and quantity ascending order. val results = sqlContext.sql('.....SELECT * FROM products WHERE name LIKE 'Pen %' ORDER BY price DESC, quantity.....)

results. showQ

Step 4 : Select top 2 products by price

val results = sqlContext.sql(.....SELECT' FROM products ORDER BY price desc LIMIT2.....}

results. show()

B- Solution :

Step 1 : Select all the products which has product code as null
val results = sqlContext.sql(.....SELECT' FROM products WHERE code IS NULL.....)
results. showQ
val results = sqlContext.sql(.....SELECT * FROM products WHERE code = NULL ',,M)
results.showQ
Step 2 : Select all the products , whose name starts with Pen and results should be order by Price descending order. val results = sqlContext.sql(.....SELECT * FROM products WHERE name LIKE 'Pen %' ORDER BY price DESC.....)
results. showQ
Step 3 : Select top 2 products by price
val results = sqlContext.sql(.....SELECT' FROM products ORDER BY price desc LIMIT2.....}
results. show()

Answer:

A