CS777 Module 2

Topics:

- Spark - RDDs, DataFrames, Spark SQL
- PySpark + NumPy + SciPy, Code Optimization, Cluster Configurations
- Linear Algebra Computation in Large Scale.
- Distributed File Storage Systems

2.1. Learning Objectives

After we have learned about the major problems of Big Data storage and processing in the first module, and we learned about basic concepts of Spark distributed data processing and Resilient Distributed Datasets (RDDs), in this module, we will learn about Spark Dataframes and the combination of python libraries with Spark to run bulk data processing operations. Spark provides two main distributed data structures using python API: 1) RDDs and 2.) Dataframes.

In the following sections, we introduce concepts of Spark Dataframes and Spark Dataframe API operations and we learn how we can use Spark RDD and Spark Dataframes for numerical operations like those operations required in large-scale machine learning.

The main learning objectives of this module are:

- Understand the concepts of using Apache Spark Dataframe.
- Understand the difference between Spark RDDs and Dataframe.
- Be able to work with different Spark Dataframe API operations.
- Be able to understand the concept of Bulk Operations and Vectorization.
- Run large-scale data processing pipeline on Spark.

2.2. Spark Dataframes

A DataFrame is a distributed collection of data organized into named columns. A Dataframe in Spark is a data structure that includes data schema similar to a table in a relational database or a Dataframe in R or Python libraries like Pandas.[1] However, a Dataframe in PySpark differs from a Pandas Dataframe. A Spark Dataframe it is a distributed data structure while in Pandas it is a local data structure on a single machine and not scalable to use for large datasets. In your implementation you should not use Pandas Dataframe and Spark Dataframe in the same code.

Spark SQL is Spark's module for working with structured data. You can create and query structured data inside Spark programs, using either SQL or DataFrame API. You can find the main Spark PySpark SQL and Dataframe documentation on the main documentation site of Spark.[2] A sparkcontext or sparksession is the main entry point for working with DataFrame and SQL functionality. Listing 2.1 is an instantiation example of sparkSession and sqlContext for a stand-alone installation of Spark on local machine ("local[*]" is the name of the standalone local machine).

Listing 2.1: Create SparkSession or sqlContext

```
>>>> spark = SparkSession.builder.master("local[*]").getOrCreate()

>>> sc = SparkContext.getOrCreate()
>>> sqlContext = SQLContext(sc)
```

1. Python Pandas libraries

2. Spark

2.3. Create a Dataframe

A Spark dataFrame can be constructed from different sources such as:

- Structured data files, (CSV, JSON, ...)
- External databases (MySql, PostgreSQL, S3, JDBS, ...)
- Existing RDDs in memory

Listing 2.2 is an example of creating a dataframe from a python list data in the main memory. The operation `createDataFrame(createDataFrame(data, schema=None, samplingRatio=None)` on `sqlContext` creates a Spark dataframe from a python list. On Spark dataframes similar to Spark RDDs we have `collect()` operation which can pull the distributed data (dataframe or RDD) to the driver (Driver is the process that submits the Spark scripts to the master node).

Note: The 'u' sing behind the strings in Listing 2.2 means that the string is encoded in Unicode format.

Listing 2.2: Create a Spark Dataframe

```
>>> a = [('Chris', 'Berliner', 5)]
>>> sqlContext.createDataFrame(a, ['drinker', 'beer', 'score']).collect()
[Row(drinker=u'Chris', beer=u'Berliner', score=5)]
```

Listing 2.3 is an example of creating a Dataframe from an RDD. Here for illustration first, we create an RDD from a python list and then create a Spark Dataframe from the existing RDD.

Listing 2.3: Create a Spark Dataframe

```
>>> a = [('Chris', 'Berliner', 5)]
>>> rdd = sc.parallelize(a)
>>> sqlContext.createDataFrame(rdd).collect()
[Row(_1=u'Chris', _2=u'Berliner', _3=5)]
>>> df = sqlContext.createDataFrame(rdd, ['drinker', 'beer', 'score'])
>>> df.collect()
[Row(drinker=u'Chris', beer=u'Berliner', score=5)]
```

As you can observe in Listing 2.2 and 2.3, Spark system can automatically detect the data schema and create a Dataframe from an RDD or a list. Spark gives also automated names to the Dataframe columns besides the automated schema. Listing 2.4 is an example of how you can add your schema specification and columns names to create a Spark Dataframe.

Listing 2.4: Create a Spark Dataframe with schema and column names

```
>>> from pyspark.sql.types import *
>>> schema = StructType([
...     StructField("drinker", StringType(), True),
...     StructField("beer", StringType(), True),
...     StructField("score", IntegerType(), True)])
>>> df3 = sqlContext.createDataFrame(rdd, schema)
>>> df3.collect()
[Row(drinker=u'Chris', beer=u'Berliner', score=5)]
```

Listing 2.5 is an example of importing data from a database and creating Spark Dataframe. In this example, we create a Dataframe by proving a JDBC (Java Database Connectivity) driver url.

Listing 2.5: Create a Spark Dataframe from a database table

```
>>> df = sqlContext.load( source="jdbc", url="jdbc:postgresql://<HOST>/<DATABASE>?
     user=<USERNAME>&password=<PASSWORD>",
dbtable="<SCHEMA>.<TABLENAME>")
```

2.3.1. Create a Dataframe from a JSON File.

You can use the operation `jsonFile(PATH)` to create a Dataframe from a JSON file. Columns of the Dataframe will have the key names of the Json object and Spark detects automatically the data types. You can change the data types after creating the Dataframe if needed. Listing 2.6 shows an example of this operation and creates a Dataframe from a file path (In this example Path is a local file but it can be any Path including distributed file.)

Listing 2.6: Create a Spark Dataframe from a JSON file

```
>>> sqlContext.jsonFile('file:///home/rates.json').dtypes
[ ('drinker', 'string'), ('beer', 'string'), ('score', 'bigint'),]
```

Listing 2.7 is an example of loading data from a CSV (Comma Separated Volume) file. We can define different signs as the separator sign, if the data set has a different separator sign than comma, like in this example. Again, Spark detects automatically the data schema and column names can be imported from the header line of the CSV file, if such header line exists in the data file.

In Listing 2.7 two further operations are also used, the first one is the show() operation that you can use to print out a Projection of a dataframe to see what are the column names and data schema. If you need only to look at the schema of your dataframe you can use `printSchema()` operation.

Listing 2.7: Create a Spark Database

```
>>  customers = sqlContext.read.format('csv')\
>>                        .options(header='true', inferSchema='true',  sep="|")\
>>                        .load("file:///YOUR-FOLDER/customer.tbl")

# shows the content of the DataFrame
# Show the table and 20 rows of it.
>> customers.show()

# Print out schema of Dataframe
>> customers.printSchema()
```

2.4. Working with Spark Dataframes

Spark Dataframe provides a list of transformation and action operations similar to spark RDDs operations. These operations are similar to RDDs with some differences because a dataframe includes data schema and Spark process it in a rational form and database table style. In the following sections, we describe the most relevant operations and refer the readers to the main documentation page Spark dataframe.[3]

2.4.1. Selection Operation

Spark dataframe selection operation is very similar to SQL selection operation. `select(*cols)` Projects a set of expressions and returns a new DataFrame.

Listing 2.8: Create a Spark Dataframe

```
>>> df.select('*').collect()
[Row(drinker=u'Chris',  beer=u'Berliner', score=5),  Row(drinker=u'Kia',
     beer=u'Erdinger Kristal', score=2)]

>>> df.select('beer', 'score').collect()
[Row(beer=u'Berliner', score=5),  Row(beer=u'Erdinger Kristal', score=2)]
```

2.4.2. Selection with Expression

Similar to the select operation, Spark provides another selection operation that can work with specific SQL expression. It projects a set of SQL expressions and returns a new Dataframe. The `selectExpr(*expr)` selects one or more columns and runs the specified computations on them. Listing 2.9 is an example of `selectExpr()` operation which selects a column score and runs a mathematic computation on it. The resulting data frame will have columns named with the same SQL expression as show in the example.

Listing 2.9: Create a Spark Dataframe

```
>>> df.selectExpr("(score / 10) * 100", "abs(score)").collect()
[Row(((score / 10) * 100)=50, abs(age)=5), Row(((score / 10) * 100)=20, abs(age)=2)]
```

2.4.3. Drop Operation

The `drop()` operation removes the specified columns and returns a new Spark Dataframe with it. Listing 2.10 is an example of the drop operation. In this example, assume that the starting Dataframe "`df`" includes a column with the name '`drinker`', the resulting Dataframe will include all other columns but not the '`drinker`' column.

Listing 2.10: Create a Spark Dataframe

```
>>> df.drop('drinker').collect()
[Row(beer=u'Berliner', score=5), Row(beer=u'Erdinger Kristal', score=2)]
```

2.4.4. Filtering Rows

Filter operation can filter rows using the condition. Conditions is a Column of type ”BooleanType” or a ”String of SQL expression”. Listing 2.11 is an example of filter operation. In this example, we assume that the Dataframe ’`df`’ includes a column with the name '`score`’, the filter operation filters out all of those rows that satisfies the condition.

Listing 2.11: Create a Spark Dataframe

```
>>> df.filter("df.score > 3").collect()
[Row(drinker=u'Chris',  beer=u'Berliner', score=5)]

>>> df.filter("df.score = 2").collect()
[Row(drinker=u'Kia',  beer=u'Erdinger Kristal', score=2)]
```

2.4.5. Group By Operation

The spark operation `groupBy(*cols)` groups the specified columns in the Dataframe. The result of "`groupBy()`" operation is a grouped object (The resulting grouped object cannot printed out). Listing 2.12 is an example of the "`groupBy()`" operation. In this example, we use "`groupBy()`" to group columns based on the values of column ’`beer`’ and on top of it we use an aggregation operation to aggregate all the ’`score`’ and calculate the mean of them.

Listing 2.12: Create a Spark Dataframe

```
>>> df.drop('drinker').groupBy('beer').agg({'score': 'mean'}).collect()
[Row(beer=u'Berliner', score=5), Row(beer=u'Erdinger Kristal', score=2)]
```

2.4.6. Join Operation

Join operation can join two Dataframes using keys. The operation `.join(other, on=None, how=None)` has 3 main arguments, first the other Dataframe that is join is done with, the key specification ’on=’, and ’how’ the join operation has to be done. Listing 2.13 is an example of ’right’ join between two Spark Dataframes.

Listing 2.13: Spark Dataframe Join Operation

```
# Two simple python lists of tuples
>>> likes = [('Chris', 'Bud'), ('Kia', 'Berliner'), ('Matt', 'ARJK')]
>>> frequents = [('Chris', 'Bohene'), ('Kia', 'Little'), ('Oscar', 'Griff')]
# Two lists for columns names of the Dataframes
>>> likesName=['Drinker', 'Beer']
>>> frequentsName=['Drinker', 'Bar']
# Create the Dataframes with the above Lists and names
>>> likesDF = sqlContext.createDataFrame(likes, likesName)
>>> frequentsDF = sqlContext.createDataFrame(frequents, frequentsName)
# Show the Dataframes
>>> likesDF.show()
>>> frequentsDF.show()
+-------+--------+
|Drinker|    Beer|
+-------+--------+
|  Chris|     Bud|
|    Kia|Berliner|
|   Matt|    ARJK|
+-------+--------+

+-------+------+
|Drinker|   Bar|
+-------+------+
|  Chris|Bohene|
|    Kia|Little|
|  Oscar| Griff|
+-------+------+
# A right join on Key Drinker
>>> likesDF.join(frequentsDF, likesDF.Drinker == frequentsDF.Drinker, 'right').show()
+-------+--------+-------+------+
|Drinker|    Beer|Drinker|   Bar|
+-------+--------+-------+------+
|  Chris|     Bud|  Chris|Bohene|
|    Kia|Berliner|    Kia|Little|
|   null|    null|  Oscar| Griff|
+-------+--------+-------+------+
```

Join can be done using the following different join methods: ‘`full`‘, ‘`fullouter`‘, ‘`full outer`‘, ‘`left`‘, ‘`leftouter`‘, ‘`left outer`‘, ‘`right`‘, ‘`rightouter`‘, ‘`right outer`‘, ‘`semi`‘, ‘`leftsemi`‘, ‘`left semi`‘, ‘`anti`‘, ‘`leftanti`‘ and ‘`left anti`‘.

Note: For simplicity ’on=’ and ’how=’ can be dropped in implementation, just like the implementation shown in Listing 2.13.

Listing 2.14 and 2.15 are examples of using full join and anti-join operations.

Listing 2.14: Spark Dataframe Full Join Operation

```
>>> likesDF.join(frequentsDF, likesDF.Drinker == frequentsDF.Drinker, 'full').show()
+-------+--------+-------+------+
|Drinker|    Beer|Drinker|   Bar|
+-------+--------+-------+------+
|  Chris|     Bud|  Chris|Bohene|
|    Kia|Berliner|    Kia|Little|
|   null|    null|  Oscar| Griff|
+-------+--------+-------+------+
```

Listing 2.15: Spark Dataframe Anti Join Operation

```
>>> llikesDF.join(frequentsDF, likesDF.Drinker == frequentsDF.Drinker, 'anti').show()
+-------+----+
|Drinker|Beer|
+-------+----+
|   Matt|ARJK|
+-------+----+
```

2.4.7. Count operation

Listing 2.16: Create a Spark Dataframe

```
# Count number of rows
>>> likes DF.count()
3
```

2.4.8. Distinct Operation

Distinct operation can remove duplicate entries from Dataframe. Listing 2.17 is an example of using distinct operation.

Listing 2.17: Distinct Operation on Spark Dataframe

```
# Counts the number of records
>>> df = spark.createDataFrame([('a', 1), ('b', 1), ('b', 1), ('a', 2)], ('id', 'c'))
>>> df.show()
+---+---+
| id|  c|
+---+---+
|  a|  1|
|  b|  1|
|  b|  1|
|  a|  2|
+---+---+

>>> df.distinct().show()
```

```
+---+---+
| id|  c|
+---+---+
|  a|  1|
|  b|  1|
|  a|  2|
+---+---+
```

2.4.9. Converting a Spark Dataframe to a Spark RDD

You can convert a Dataframe to an RDD using the ".rdd" as a property value of a Dataframe. Listing 2.18 provides an example usages of converting a Dataframe to an RDD. As you can read `.rdd` is a property of a Spark Dataframe that you can access it and it will provide a Dataframe that includes values of type `Row()`. If you need to convert it to a normal RDD you use a map to a simple normal RDD.

Listing 2.18: Convert a Spark Dataframe to an RDD

```
# Create an example Dataframe
>>> df = spark.createDataFrame([('a', 1), ('b', 1), ('b', 1), ('a', 2)], ('id', 'c'))
>>> df.show()
+---+---+
| id|  c|
+---+---+
|  a|  1|
|  b|  1|
|  b|  1|
|  a|  2|
+---+---+

# Get access to the RDD.
# Returns the content as an pyspark.RDD
>>> rdd = df.rdd
>>> print(rdd.collect())
[Row(id='a', c=1), Row(id='b', c=1), Row(id='b', c=1), Row(id='a', c=2)]

# If you need a normal RDD you can map it and use list to get ride of Rows
>>> print(df.rdd.map(list).collect())
[['a', 1], ['b', 1], ['b', 1], ['a', 2]]
# Similar to above you can use tuple to convert it to a normal rdd.
>>> print(df.rdd.map(tuple).collect())
[('a', 1), ('b', 1), ('b', 1), ('a', 2)]
```

2.4.10. Spark Dataframe Column Types

One advantage of using Spark Dataframe is that a Dataframe include the data schema. This makes the computation on Dataframes to run with high performance compared to Spark RDDs because it includes the data schema and can allocate the required memory in a more efficient way than in RDDs.

Spark Dataframe includes a set of object types including numerical types. Table 2.1 provides an overview of the existing numeric data types in Spark dataframe. You can cast types using cast operation on types, for example using `variableName.cast(LongType())`.

Table 2.1: Numeric Types of Spark Dataframe

Data Type	Description
`ByteType`	Represents 1-byte signed integer numbers. (-128 to 127)
`ShortType`	Represents 2-byte signed integer numbers. (-32768 to 32767)
`IntegerType`	Represents 4-byte signed integer numbers. (-2147483648 to 2147483647)
`LongType`	Represents 8-byte signed integer numbers. (-9223372036854775808 to 9223372036854775807)
`FloatType`	Represents 4-byte single-precision floating point numbers.
`DoubleType`	Represents 8-byte double-precision floating point numbers.
`StringType`	Represents character string values.
`BinaryType`	Represents byte sequence values.
`BooleanType`	Represents boolean values.
`TimestampType`	Represents values comprising values of fields year, month, day, hour, minute, and second.
`DateType`	Represents values comprising values of fields year, month, day.

2.4.11. Add a New Column to Dataframe

The Spark Dataframe operation `withColumn()` can add a new column to an existing Dataframe. Listing 2.19 shows how we can use `withColumn` operation to add a new column. Listing 2.19 is an example of this process, we add a new column to the existing Dataframe and initialize the new column with value of an integer 1. The function `pyspark.sql.functions.lit(col)` creates a column of literal value. Spark assigns automatically the type of integer to this value.

Listing 2.19: Add a new column to Dataframe using withColumn

```
>>> df = spark.createDataFrame([['a'], ['b'], ['b'], ['c']], (['word']))
>>> df.show()
+----+
|word|
+----+
|   a|
|   b|
|   b|
|   c|
+----+

>>> from pyspark.sql.functions import lit
>>> new_df=df.withColumn("COUNT", lit(1))
>>> new_df.show()
+----+-----+
|word|COUNT|
+----+-----+
|   a|    1|
|   b|    1|
|   b|    1|
|   c|    1|
+----+-----+
```

Listing 2.20 shows how we can use aggregation operation to aggregate on the count the numbers and build up a word count example using the code from Listing 2.19. The resulting dataframe will have a new column as the result of aggregation operation with the name of the applied function (`sum(COUNT)` in this example).

Listing 2.20: Create a Spark Dataframe

```
>>> from pyspark.sql.functions import lit
>>> from pyspark.sql import functions as func

>>> df = spark.createDataFrame([['a'], ['b'], ['b'], ['c']], (['word']))
>>> new_df = df.withColumn("COUNT", lit(1))
>>> new_df.groupBy("word").agg(func.sum("COUNT")).show()
+----+----------+
|word|sum(COUNT)|
+----+----------+
|   c|         1|
|   b|         2|
|   a|         1|
+----+----------+
```

One of the very useful functionality of Spark is to allow users to implement the data processing computations using user-defined functions. Listing 2.21 is an example of using user-defined functions to work on Dataframe. The user-defined functions is a powerful functionality that we use to define any computations in form of functions to apply them on rows of a dataframe.

Listing 2.21: Run User Defined on Spark Dataframe

```
# UDF - User Defined Functions
>>> from pyspark.sql.types import StringType
>>> from pyspark.sql.functions import udf

>>> l = [('Alice', 25), ('Robert', 12), ('Chris', 45)]
>>> df = sqlContext.createDataFrame(l, ['Name', 'Age'])
>>> df.show()
+------+---+
|  Name|Age|
+------+---+
| Alice| 25|
|Robert| 12|
| Chris| 45|
+------+---+

>>> maturity_udf = udf(lambda age: "Adult" if age >=18 else "Child", StringType())
>>> newdf=df.withColumn("Maturity", maturity_udf(df.Age))
>>> newdf.show()
+------+---+--------+
|  Name|Age|Maturity|
+------+---+--------+
| Alice| 25|   Adult|
|Robert| 12|   Child|
| Chris| 45|   Adult|
+------+---+--------+
```

2.4.12. Top-K Queries

Spark dataframe does not provide a specific operation to run top-k queries similar to `top()` in RDDs. Instead, in dataframe we have to order the dataframe by specific key (one of the columns) and then access the top. This top-k operation dataframe is an efficient operation because it just need to organize the points to the data values in a sorted form and not moving the data around. Listing 2.22 is an example of a top-k operation on dataframe using `orderBy()`. The (limit()) operations then picks up the top of the list based on the requested number.

Listing 2.22: Create a Spark Dataframe

```
>>> from pyspark.sql import functions as func
>>> from pyspark.sql.functions import lit
>>> from pyspark.sql.types import StringType
>>> from pyspark.sql.functions import udf

>>> l = [('Alice', 25), ('Robert', 12), ('Chris', 45)]
>>> df = sqlContext.createDataFrame(l, ['Name', 'Age'])
>>> df.orderBy("Age", ascending=False).limit(1).show()
+-----+---+
| Name|Age|
+-----+---+
|Chris| 45|
+-----+---+
```

3. Spark SQL and Dataframe and Spark SQL and Dataframe Programing Guide

2.5. Spark Dataframes and SQLs

You can use the query language SQL when working with Spark Dataframe. As shown in Listing 2.23, you need to register your Spark Dataframe using sqlContext, then you can define your SQL query and execute it. When your SQL query is very complex on multiple Dataframes, this kind of table registry and querying in Spark does not provide a high performance data processing. It is preferred to work the Spark Dataframe API directly than using SQL API.

Listing 2.23: Using SQL with Spark Dataframes

```
>>> sqlContext.registerDataFrameAsTable(df, "rates")
>>> df2 = sqlContext.sql("SELECT drinker AS d, beer as b, score as s from rates")
>>> df2.collect()
[Row(d=u'Chris', b=u'Berliner', s=5), Row(d=u'Kia', b=u'Erdinger Kristal', s=2) ]
```

2.6. Code Optimization

Data processing of an enormous volume of data requires cluster management and a well-functioning computational implementation code optimized to use all the processing power of the cluster machine and use all available CPU cores and available RAM memory for high-performance processing. Multi-threading and multi-processing is the key concept for using all cluster computing resources for data processing. We want to run multiple processes on different machines and let each of these separate processes run multiple threads in parallel on multiple CPU cores. In this way, it is possible to use CPU hyper-threading on virtual cores of each machine and have them work together to perform a very large data processing computation in one data processing pipeline.

To achieve scalability by scaling up on each machine and scaling out on multiple machines we use bulk operation and vectorization techniques to achieve multithreading on each running process.

2.6.1. Bulk Operations and Vectorization Techniques

In large-scale machine learning data processing, we use bulk operations which multiple independent threads can execute and allow data processing systems to scale up on a single machine without need of iterative interpretation of computations.

In PySpark, we combine python library Numpy with Spark RDDs to achieve multi-threading on each process and avoid python low performance for and while loops.

2.6.2. RDDs + NumPy Arrays

In machine learning data processing, we mostly pre-process and clean up the data first and then create a very large feature matrix. In PySpark, when we want to use PySpark RDDs to store the large matrix, we prefer to combine Numpy with Spark RDDs and store as values in each row of RDD a single Numpy vector. We store each row of the feature matrix in an RDD row as a single Numpy array. We can use a unique array of representations like using Spark Sparse or Dense arrays from Spark Mllib library. Spark Dense arrays representation can be helpful to reduce the required cluster memory because mostly we are working with feature matrix with many zero values.

Listing 2.24 is an evaluation experiment to compare performance differences between using python for-loops and Numpy library. When we run this script on an example computer, the usage of Numpy was over 200 times faster than using python for-loops. In Big Data processing this will happen a lot on each process running on different server machines. If each of these tasks can be 200 times faster when using Numpy library this will sum up to an enormous performance difference.

Listing 2.24: Using numpy with Spark RDDs - Performance Comparision

```
from __future__ import print_function
import sys
import numpy as np
import time
from datetime import datetime

if __name__ == "__main__":

  x = np.ones(20000000)*2
  y = np.ones(20000000)*2

  start = time.time()
  result1 =0
  result1=np.sum(x*y)

  done = time.time()
  print (result1)
  elapsed = done - start
  print('Numpy Elapsed Time: ', elapsed)



# Now, let us do the same thing and compare the time.
  startNew = time.time()
  result2 =0
  for i in range (20000000):
    result2 = result2 + x[i]*y[i]

  doneNew = time.time()
  print (result2)
  elapsedNew = doneNew - startNew
  print('ForLoop Elapsed Time: ',  elapsedNew)

# On my computer it will result in following
80000000.0
Numpy Elapsed Time:  0.03132128715515137

80000000.0
ForLoop Elapsed Time:  7.372159004211426
```

Listing 2.25 provides another example of how vectorized bulk computation can be done by combining Spark RDDs and Python Numpy.

Listing 2.25: Using numpy with Spark RDDs - Performance Comparision

```
# Generate some example data.
# This kind of data can be very large in practice.
# We generate a small data to illustrate
<<< mat = np.random.rand(8,1).reshape(4, -1)
<<< rdd = sc.parallelize(mat)
<<< print(rdd.collect())
[array([0.90628538, 0.5352517 ]), array([0.09070436, 0.31141309]), array
    ([0.84965395, 0.02553221]), array([0.62701883, 0.06943633])]

# Run vectorized bulk operation like following.
# Instead of np.add you can use + sign, like
# lambda x,y : x +y  # this would be numpy add
<<< rdd.reduce(lambda x, y: np.add(x, y))
array([2.47366252, 0.94163334])
```

2.6.3. Dataframes + DenseVector

In Spark dataframes, we need to have data schema for every column. Spark includes a linear algebra library `pyspark.ml.linalg` that provides basic data structures and mathematic functions. Linear algebra library (Linalg) includes Vector.dense and Vector.sparse data structures which work similar to Numpy library in PySpark. Listing 2.26 is illustrating an example on how these sparse vectors can be included in dataframe.

Listing 2.26: Using numpy with Spark RDDs - Performance Comparision

```
<<< from pyspark.ml.linalg import Vectors

# Example vector size of 2. This can be very large Vector
<<< size=2

<<< data = [(0, Vectors.dense(np.random.rand(size)),),
<<<         (1, Vectors.dense(np.random.rand(size)),),
<<<         (1, Vectors.dense(np.random.rand(size)),),
<<<         (0, Vectors.dense(np.random.rand(size)),)]

<<< df = spark.createDataFrame(data, ["label", "features"])
<<< df.show()

+-----+--------------------+
|label|            features|
+-----+--------------------+
|    0|[0.93408840423865...|
|    1|[0.09589846183053...|
|    1|[0.60574712432561...|
|    0|[0.77373569129505...|
+-----+--------------------+
```

2.7. Cluster Configuration

As described in the Spark Installations Standalone vs. Cluster Mode section of Module 1, spark system includes many configuration parameters for setting up the resources for executors and the driver process. Changing the configurations will change the processing time that your task need to finish a specific task on a Spark cluster. One of the best practices is to start up large executors and driver that include maximum memory and CPU to each of these processes so they do not terminate on out of memory exceptions easily. Running lots of small executors will hit the overall performance of the systems because of the costs of related to instantiations, termination and coordination of small collaborating processes.

You can find more details about the exact Spark Cluster configuration parameters on Spark website.[4]

4. Spark Configuration

2.8. Practice Examples

In this section, you can find two example data wrangling examples that you can use to train and improve your Spark coding abilities.

2.8.1. Practice Example: Large-Scale Customer Data Wrangling

The data set in this example is about a business that has data about customers, orders, lineitems and parts (products). We generated the data using the TPCH data generator from a database benchmark system named TPC-H.[5]

Customer Table. Customer data is stored in CVS file separated by pipe sign ”l” and is storing the following values.

CUSTKEYlNAMElADDRESSlNATIONKEYlPHONElACCBATLlMKTSEGMENTlCOMMENTl

Orders Table. Order data is stored in CVS file separated by pipe sign ”l” and is storing the following values.

lORDERKEYlCUSTKEYlORDERSTATUSlTOTALPRICElORDERDATEl ORDER—PRIORITYlCLERKlSHIP—PRIORITYlCOMMENTl

Lineitem Table. Lineitem data is stored in another CVS file separated with the following fields.

lORDERKEYlPARTKEYlSUPPKEYlLINENUMBERlQUANTITYlEXTENDEDPRICE lDISCOUNTjlTAXlRETURNFLAGlLINESTATUSlSHIPDATElCOMMITDATEl RECEIPTDATElSHIPINSTRUCTlSHIPMODElCOMMENTl

Figure 2.1[6] illustrates the TPC-H data schema. You can see the connecting unique key links that build the table relationships.

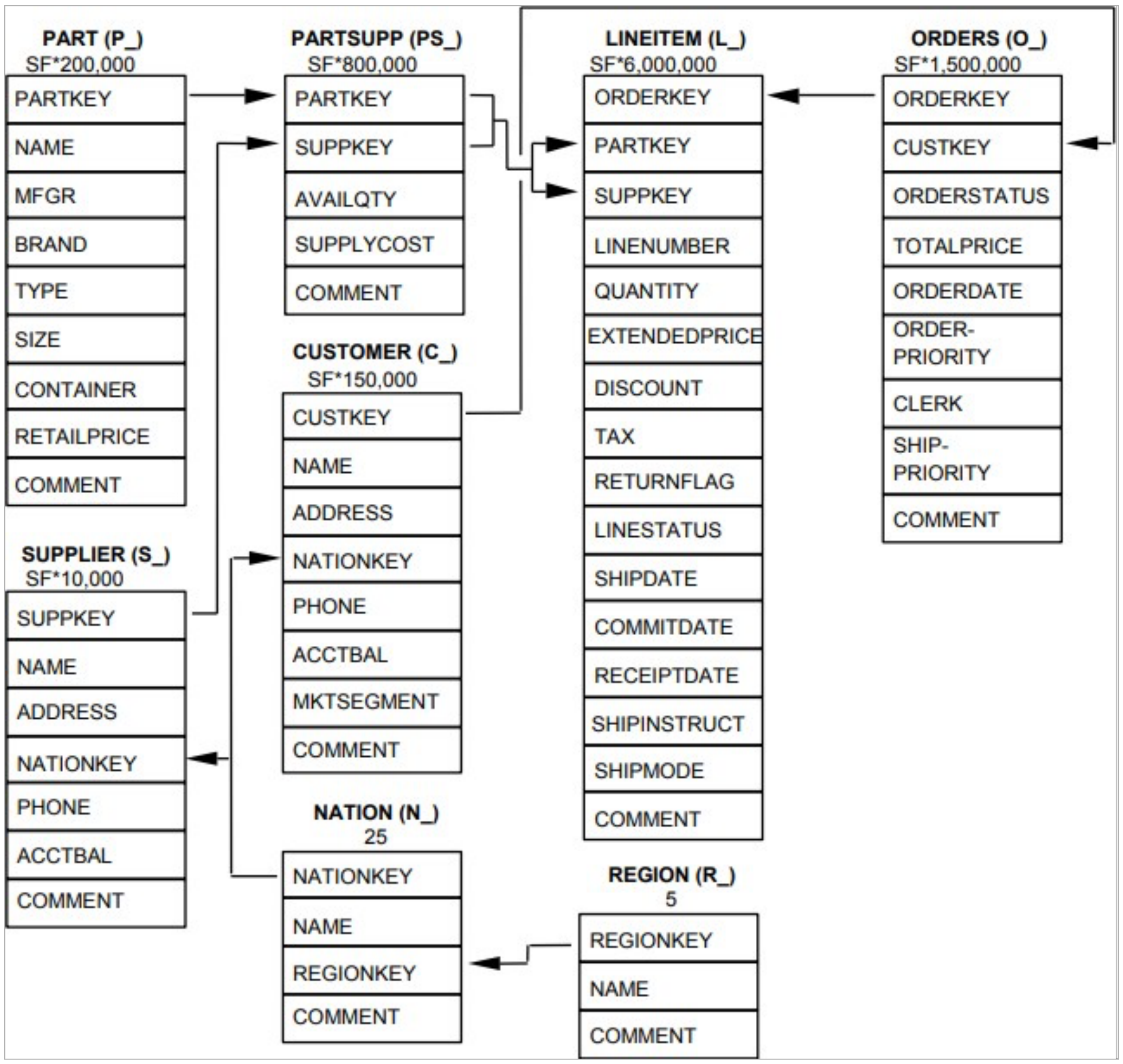

Figure 2.1: The TPC-H Database Schema

You can download 3 data files of the 0.1 scale TPCH generated dataset, files are Customer data file, Order data file, Lineitem data file.

Your task in this practice is to implement a set if PySpark Scripts using Dataframes or RDDs to answer the following questions:

- **Question 1:** What are the top-10 sold products?
- **Question 2:** What are the top-10 customers based on the number of products ordered?
- **Question 3:** What are the top-10 customers that have ordered products from the same supplier?
- **Question 4:** Who are the customers that have not ordered products from their own country and have ordered only foreign products?
- **Question 5:** Which top 3 countries produced most of the products that are ordered?
- **Question 6:** Who are the top-10 similar customers based on their orders? (Use Jaccard similarity to calculate the similarity) Consider only customers that have ordered at least 10 products. First collect all the products that each customer ordered.

- **Question 7:** What are the top-10 products pairs that the customer ordered mostly together?

Solution: You can find the solutions of this practice example on the Big Data Analytics Github Repository

2.8.2. Practice Example: Flight Data Wrangling

Implement the flight data wrangling example described in the Module 1 Practice Example, using Spark Dataframe API only.

---

5. TPC-H database benchmark

6. Image from TPCH benchmark original documentation

2.9. Summary

In this module we have learned:

- How to run bulk operations in PySpark by using python Numpy library with PySpark RDDs
- What is a Spark Dataframe and how it can be used to run Big Data processing pipelines.
- Different Data Types in Spark Dataframes
- Spark data processing using Spark dataframes

In the next module we we will learn:

- What the data models are in data science.
- What are the different data modeling techniques including cost models and Bayesian methods.
- Optimization Methods including gradient descent and newtons method.

2.10. Further Reading References

- Bill Chambers and Matei Zaharia (2018). Spark: The Definitive Guide: Big Data Processing Made Simple, published by O'Reilly Media Inc.
- Tomasz Drabas, Denny Lee (2017). Learning PySpark. Build data-intensive applications locally and deploy at scale using the combined powers of Python and Spark 2.0, published by O'Reilly Media Inc.

Further Systems

- Apache Flink is a similar system to Apache Spark and includes further data processing features.
- Apache SystemML is an optimized Big Data system to execute Large Scale Linear Algebra Operations and Machine Learing Algorthms on top of Apache Spark and Hadoop Ecosystem.

Boston University Metropolitan College