**Apache Spark**

**Apache Spark Introduction:**

[Spark](https://spark.apache.org/) is an Apache project advertised as “lightning fast cluster computing”. It has a thriving open-source community and is the most active Apache project at the moment.

Spark provides a faster and more general data processing platform. Spark lets you run programs up to 100x faster in memory, or 10x faster on disk, than Hadoop. Last year, Spark took over Hadoop by completing the 100 TB Daytona GraySort contest 3x faster on one tenth the number of machines and it also became the [fastest open source engine for sorting a petabyte](http://databricks.com/blog/2014/10/10/spark-petabyte-sort.html).

Spark also makes it possible to write code more quickly as you have over 80 high-level operators at your disposal. To demonstrate this, let’s have a look at the “Hello World!” of BigData: the Word Count example. Written in Java for MapReduce it has around 50 lines of code, whereas in Spark (and Scala) you can do it as simply as this:

sparkContext.textFile("hdfs://...")

.flatMap(line => line.split(" "))

.map(word => (word, 1)).reduceByKey(\_ + \_)

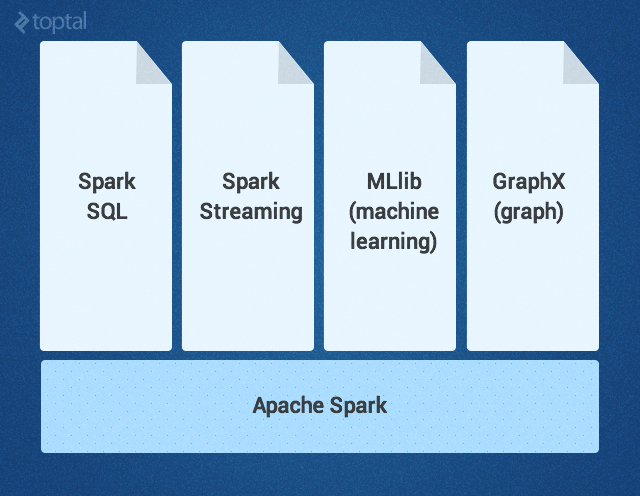
.saveAsTextFile("hdfs://...")

Another important aspect when learning how to use Apache Spark is the interactive shell (REPL) which it provides out-of-the box. Using REPL, one can test the outcome of each line of code without first needing to code and execute the entire job. The path to working code is thus much shorter and ad-hoc data analysis is made possible.

Additional key features of Spark include:

* Currently provides APIs in Scala, Java, and Python, with support for other languages (such as R) on the way
* Integrates well with the Hadoop ecosystem and data sources (HDFS, Amazon S3, Hive, HBase, Cassandra, etc.)
* Can run on clusters managed by Hadoop YARN or Apache Mesos, and can also run standalone

The Spark core is complemented by a set of powerful, higher-level libraries which can be seamlessly used in the same application. These libraries currently include SparkSQL, Spark Streaming, MLlib (for machine learning), and GraphX, each of which is further detailed in this article. Additional Spark libraries and extensions are currently under development as well.



## Spark Core

[Spark Core](http://mvnrepository.com/artifact/org.apache.spark/spark-core_2.10/1.0.0) is the base engine for large-scale parallel and distributed data processing. It is responsible for:

* memory management and fault recovery
* scheduling, distributing and monitoring jobs on a cluster
* interacting with storage systems

Spark introduces the concept of an [RDD (Resilient Distributed Dataset)](http://spark.apache.org/docs/1.2.1/programming-guide.html#resilient-distributed-datasets-rdds), an immutable fault-tolerant, distributed collection of objects that can be operated on in parallel. An RDD can contain any type of object and is created by loading an external dataset or distributing a collection from the driver program.

RDDs support two types of operations:

* [Transformations](http://spark.apache.org/docs/1.2.1/programming-guide.html#transformations) are operations (such as map, filter, join, union, and so on) that are performed on an RDD and which yield a new RDD containing the result.
* [Actions](http://spark.apache.org/docs/1.2.1/programming-guide.html#actions) are operations (such as reduce, count, first, and so on) that return a value after running a computation on an RDD.

Transformations in Spark are “lazy”, meaning that they do not compute their results right away. Instead, they just “remember” the operation to be performed and the dataset (e.g., file) to which the operation is to be performed. The transformations are only actually computed when an action is called and the result is returned to the driver program. This design enables Spark to run more efficiently. For example, if a big file was transformed in various ways and passed to first action, Spark would only process and return the result for the first line, rather than do the work for the entire file.

By default, each transformed RDD may be recomputed each time you run an action on it. However, you may also persist an RDD in memory using the persist or cache method, in which case Spark will keep the elements around on the cluster for much faster access the next time you query it.

## SparkSQL

[SparkSQL](https://spark.apache.org/sql/) is a Spark component that supports querying data either via SQL or via the [Hive Query Language](https://cwiki.apache.org/confluence/display/Hive/LanguageManual). It originated as the Apache Hive port to run on top of Spark (in place of MapReduce) and is now integrated with the Spark stack. In addition to providing support for various data sources, it makes it possible to weave SQL queries with code transformations which results in a very powerful tool. Below is an example of a Hive compatible query:

// sc is an existing SparkContext.

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

sqlContext.sql("CREATE TABLE IF NOT EXISTS src (key INT, value STRING)")

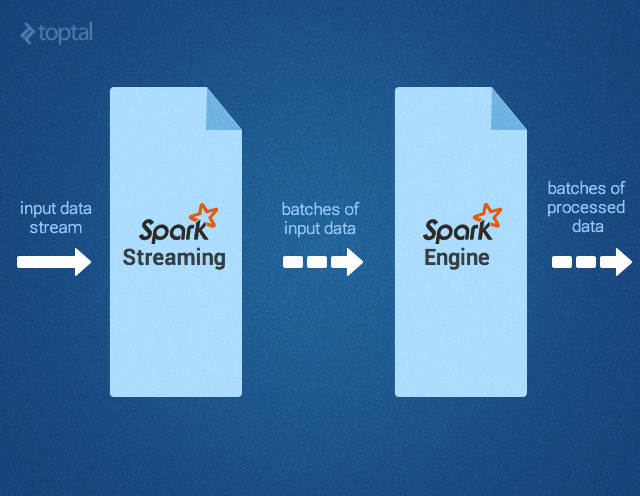
sqlContext.sql("LOAD DATA LOCAL INPATH 'examples/src/main/resources/kv1.txt' INTO TABLE src")

// Queries are expressed in HiveQL

sqlContext.sql("FROM src SELECT key, value").collect().foreach(println)

## Spark Streaming

[Spark Streaming](https://spark.apache.org/streaming/) supports real time processing of streaming data, such as production web server log files (e.g. Apache Flume and HDFS/S3), social media like Twitter, and various messaging queues like Kafka. Under the hood, Spark Streaming receives the input data streams and divides the data into batches. Next, they get processed by the Spark engine and generate final stream of results in batches, as depicted below.



The Spark Streaming API closely matches that of the Spark Core, making it easy for programmers to work in the worlds of both batch and streaming data.

## MLlib

[MLlib](https://spark.apache.org/mllib/) is a machine learning library that provides various algorithms designed to scale out on a cluster for classification, regression, clustering, collaborative filtering, and so on (check out Toptal’s article on [machine learning](http://www.toptal.com/machine-learning/machine-learning-theory-an-introductory-primer) for more information on that topic). Some of these algorithms also work with streaming data, such as linear regression using ordinary least squares or k-means clustering (and more on the way). Apache[Mahout](http://mahout.apache.org/) (a machine learning library for Hadoop) has already turned away from MapReduce and joined forces on Spark MLlib.

Top of Form

**Like what you're reading?**

[GraphX](https://spark.apache.org/graphx/)

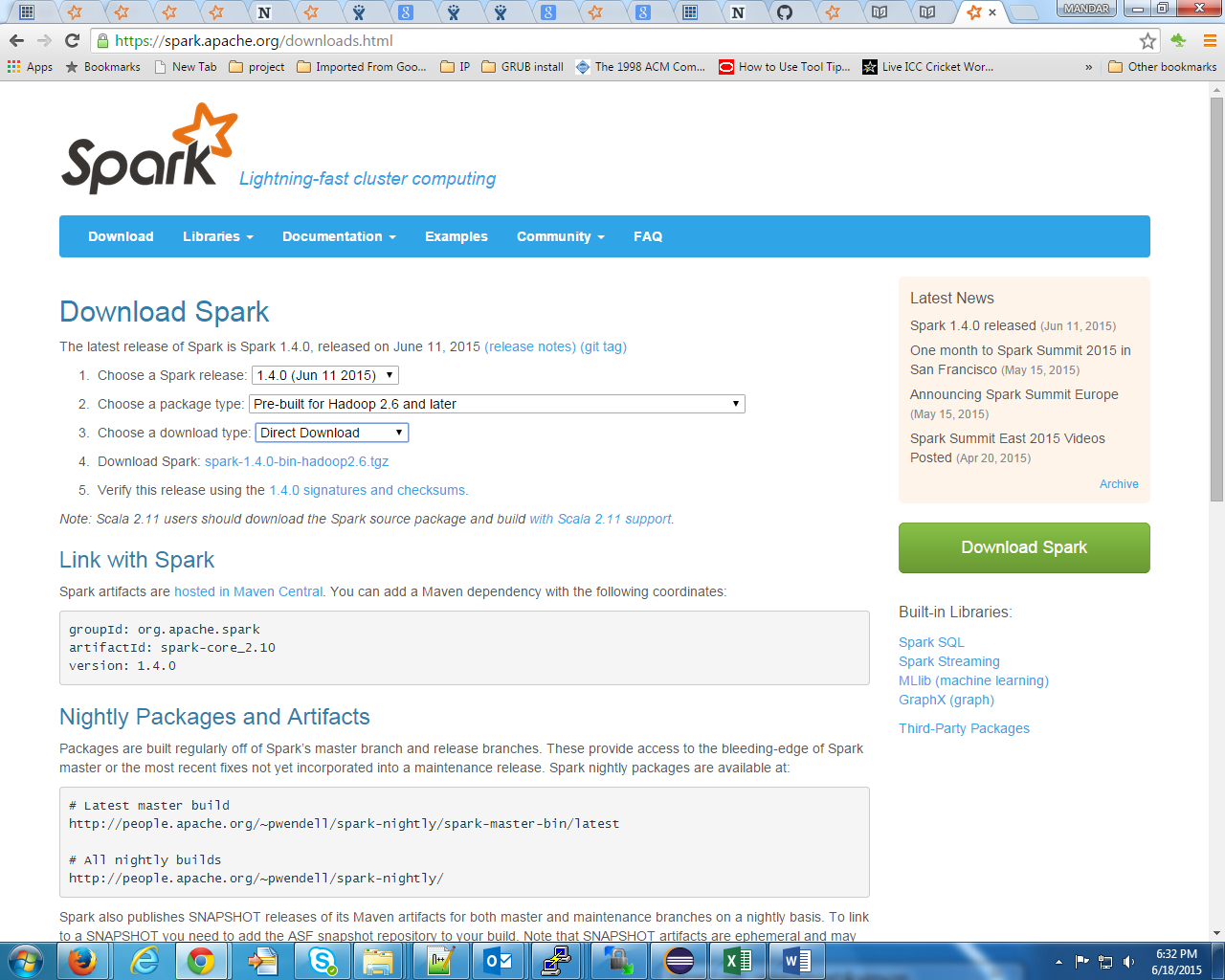
[GraphX](https://spark.apache.org/graphx/) is a library for manipulating graphs and performing graph-parallel operations. It provides a uniform tool for ETL, exploratory analysis and iterative graph computations. Apart from built-in operations for graph manipulation, it provides a library of common graph algorithms such as PageRank.

**Installation Process:**

1) Download the apache spark from spark site.

<https://spark.apache.org/downloads.html>

2) Select following option for download as shown in the figure.

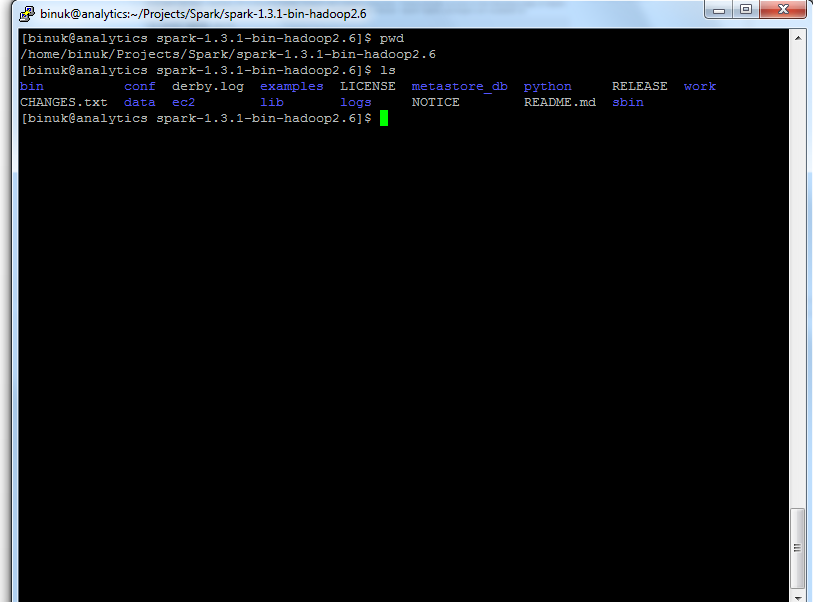


3) Click on the download spark: spark-1.4.0-bin-hadoop2.6.tgz link for downloading.

4) After Downloading extract the spark-1.4.0-bin-hadoop2.6.tgz .

**How to Run the Spark:**

Open the extracted spark folder as you can see the folder structure as follows.



2) You can run the spark from **Scala shell** or **python shell**

**Note: Before Running the Spark JAVA\_HOME should be set the correct java version above 6.**

To Run the Spark from Scala shell run the following command:

**./bin/spark-shell**

To Run the Spark from python shell run the following command:

**./bin/pyspark**

**Setting the SPARK\_CLASSPATH for loading the external jar:**

Loading the external jar which is required while running the spark you can set in the SPARK\_CLASSPATH before running the spark shell.

e.g. : Set the SPARKCLASSPATH for oracle and sql database connectivity jar

export SPARK\_CLASSPATH=/home/binuk/Projects/Spark/ojdbc14.jar ->Oracle

export SPARK\_CLASSPATH=/home/binuk/Projects/Spark/mysql-connector-java-5.1.18-bin.jar ->Mysql

**Loading the json data from file into spark:**

1. Goto the spark download folder as shown in the **How to Run the Spark section**
2. Run the python spark shell using following command:

./bin/pyspark

1. >>> shell will come on the terminal
2. Run the following command:

>>> df = sqlContext.jsonFile("/home/binuk/Projects/Spark/people.json")

>>> df.show()

age name

10 ABC

>>> df.printSchema()

root

|-- age: string (nullable = true)

|-- name: string (nullable = true)

>>> df.select("name").show()

name

ABC

>>> df.select("age").show()

age

10

Where ("/home/binuk/Projects/Spark/people.json”) contains the data

Like:

{"age":"10","name":"ABC"}

**Loading the oracle data from database into spark:**

1. Set the oracle jdbc connection jar in the SPARK\_CLASSPATH as shown in the **Setting the SPARK\_CLASSPATH for loading the external jar** Section or **set in the ~/.bash\_profile file**.

**Edit the bash\_profile using following command:**

Vi ~/.bash\_profile

Add the following line at the end of file where modify the jar location according to the your location

export SPARK\_CLASSPATH=/home/binuk/Projects/Spark/mysql-connector-java-5.1.32.jar:/home/binuk/Projects/Spark/ojdbc14.jar

1. Goto the spark download folder as shown in the **How to Run the Spark section**
2. Run the python spark shell using following command:

./bin/pyspark

1. >>> shell will come on the terminal
2. Run the following command:

>>>from pyspark.sql import SQLContext

>>>sqlContext = SQLContext(sc)

>>> sqlContext.tables()

DataFrame[tableName: string, isTemporary: boolean]

>>>ddoracle1 = sqlContext.load(source="jdbc", driver="oracle.jdbc.driver.OracleDriver",url="jdbc:oracle:thin:prasads/prasads@10.113.59.5:1521:ora11g", dbtable="IDEA\_AGEING\_BADDEBT\_201415")

OR

>>>ddoracle = sqlContext.load(source="jdbc", driver="oracle.jdbc.driver.OracleDriver",url="jdbc:oracle:thin:prasads/prasads@10.113.59.5:1521:ora11g", dbtable="employee")

>>> ddoracle ->check the data frame content

>>>ddoracle.show()

>>>ddoracle.show(150) ->Show the 150 row

>>>ddoracle.printSchema()

>>>ddoracle.select("BADDEBTGROUP").show() ->select only 20 record

>>>ddoracle.select("BADDEBTGROUP").collect() ->select and display all record

>>>ddoracle.select("BADDEBTGROUP","MSISDN").show() ->Select multiple column

>>>ddoracle.select("BADDEBTGROUP","MSISDN").collect() ->Returns all the records as a list of Row.

>>>ddoracle.select("BADDEBTGROUP", ddoracle.CO\_CODE + 1.0).show() ->not working(Decimal Type comparison issue[not able to conver the java decimal to scala bigdecimal existing issue in spark])

>>>ddoracle.groupBy("BADDEBTGROUP").count().show()

>>ddoracle.groupBy("BADDEBTGROUP").collect()

>>> ddoracle.orderBy("EMPNAME")

DataFrame[EMPNAME: string, EMPAGE: decimal(10,0), EMPSALARY: string, EMPCOUNTRY: string]

>>> ddoracle.orderBy("EMPNAME").collect()

15/06/17 16:29:49 WARN SizeEstimator: Failed to check whether UseCompressedOops is set; assuming yes

[Row(EMPNAME=u'Anil', EMPAGE=Decimal('30'), EMPSALARY=u'3424', EMPCOUNTRY=u'China'), Row(EMPNAME=u'Kiran', EMPAGE=Decimal('24'), EMPSALARY=u'12356', EMPCOUNTRY=u'China'), Row(EMPNAME=u'Kishor', EMPAGE=Decimal('31'), EMPSALARY=u'34258', EMPCOUNTRY=u'USA'), Row(EMPNAME=u'Mahesh', EMPAGE=Decimal('23'), EMPSALARY=u'67890', EMPCOUNTRY=u'India'), Row(EMPNAME=u'Mayur', EMPAGE=Decimal('35'), EMPSALARY=u'45676', EMPCOUNTRY=u'China'), Row(EMPNAME=u'Umesh', EMPAGE=Decimal('45'), EMPSALARY=u'767890', EMPCOUNTRY=u'India'), Row(EMPNAME=u'Yogesh', EMPAGE=Decimal('24'), EMPSALARY=u'87864', EMPCOUNTRY=u'India'), Row(EMPNAME=u'kedar', EMPAGE=Decimal('24'), EMPSALARY=u'697890', EMPCOUNTRY=u'USA')]

>>> ddoracle.sort("EMPSALARY").collect()

[Row(EMPNAME=u'Kiran', EMPAGE=Decimal('24'), EMPSALARY=u'12356', EMPCOUNTRY=u'China'), Row(EMPNAME=u'Anil', EMPAGE=Decimal('30'), EMPSALARY=u'3424', EMPCOUNTRY=u'China'), Row(EMPNAME=u'Kishor', EMPAGE=Decimal('31'), EMPSALARY=u'34258', EMPCOUNTRY=u'USA'), Row(EMPNAME=u'Mayur', EMPAGE=Decimal('35'), EMPSALARY=u'45676', EMPCOUNTRY=u'China'), Row(EMPNAME=u'Mahesh', EMPAGE=Decimal('23'), EMPSALARY=u'67890', EMPCOUNTRY=u'India'), Row(EMPNAME=u'kedar', EMPAGE=Decimal('24'), EMPSALARY=u'697890', EMPCOUNTRY=u'USA'), Row(EMPNAME=u'Umesh', EMPAGE=Decimal('45'), EMPSALARY=u'767890', EMPCOUNTRY=u'India'), Row(EMPNAME=u'Yogesh', EMPAGE=Decimal('24'), EMPSALARY=u'87864', EMPCOUNTRY=u'India')]

>>>ddoracle.filter(ddoracle.ISNP > 211253).show()

>>>ddoracle.filter(ddoracle.LAND\_LINE\_NO\_2 > 211253).show()

>>> ddoracle.filter(ddoracle.LAND\_LINE\_NO\_2 > 211253).collect()

>>> ddoracle.groupBy("CATEGORY").count().show()

>>> ddoracle.filter(ddoracle.BADDEBTGROUP=="MAHRoam One").show()

>>> ddoracle.count() ->Get the count in the data frame

>>>ddoracle.sort(ddoracle.CCZIP).collect() ->Sort the data frame

>>> ddoracle.toJSON()

MapPartitionsRDD[41] at mapPartitions at DataFrame.scala:862

>>> ddoracle.withColumn("id\_new",ddoracle.ID+2).show() ->Returns a new DataFrame by adding a column.

>>> ddoracle.withColumnRenamed("ID","NEW\_ID") ->REturns a new DataFrame by renaming an existing column.

**Aggregate:**

agg(\*exprs)

Compute aggregates and returns the result as a DataFrame.

The available aggregate functions are avg, max, min, sum, count.

If exprs is a single dict mapping from string to string, then the key is the column to perform aggregation on, and the value is the aggregate function.

Alternatively, exprs can also be a list of aggregate Column expressions.

>>> gdf=ddoracle.groupBy(ddoracle.BILLCITY) ->First groupBy on billcity

>>> gdf.agg({"\*":"count"}).collect() ->Get the count of the billcity

>>> ddoracle.groupBy(ddoracle.ISNP).agg({"\*":"count"}).collect()

[Row(ISNP=Decimal('1'), COUNT(1)=31815), Row(ISNP=Decimal('2'), COUNT(1)=62144)]

>> ddoracle.groupBy(ddoracle.CCCITY).count().collect()

>>> ddoracle.groupBy(ddoracle.CCCITY).agg({"\*":"count"}).collect()

>>> ddoracle.flatMap(lambda p: p.CCCITY).collect()

>>> ddoracle.map(lambda p: p.CCCITY).collect()

**Average:**

avg(\*args)

Computes average values for each numeric columns for each group.

>>> ddoracle.groupBy().avg('ISNP').collect() ->Failed because java.lang.ClassCastException: java.math.BigDecimal cannot be cast to org.apache.spark.sql.types.Decimal

>>> gdf=ddoracle.groupBy(ddoracle.BILLCITY).count().show() ->Get the group By count

**MAX**

max(\*args)

Computes the max value for each numeric columns for each group.

>>>ddoracle.groupBy.max(ddoracle.ISNP).count()

Complete API For DataFrame will be available in following site:

Python: <https://spark.apache.org/docs/latest/api/python/pyspark.sql.html#pyspark.sql.DataFrame>

Java:

<https://spark.apache.org/docs/latest/api/java/index.html?org/apache/spark/sql/DataFrame.html>

**Perform the SQL Query operation on Data frame using spark SQL DataFrame API:**

1. Set the oracle jdbc connection jar in the SPARK\_CLASSPATH as shown in the **Setting the SPARK\_CLASSPATH for loading the external jar** Section
2. Goto the spark download folder as shown in the **How to Run the Spark section**
3. Run the python spark shell using following command:

./bin/pyspark

1. >>> shell will come
2. Run the following command:

>>>from pyspark.sql import SQLContext

>>>sqlContext = SQLContext(sc)

>>>ddoracle = sqlContext.load(source="jdbc", driver="oracle.jdbc.driver.OracleDriver",url="jdbc:oracle:thin:prasads/prasads@10.113.59.5:1521:ora11g", dbtable="IDEA\_AGEING\_BADDEBT\_201415")

OR

ddoracle = sqlContext.load(source="jdbc", driver="oracle.jdbc.driver.OracleDriver",url="jdbc:oracle:thin:prasads/prasads@10.113.59.5:1521:ora11g", dbtable="employee")

Register Table for performing the normal SQL operation on data frame we need to **register data frame as temp table** or use ***sqlContext*.registerDataFrameAsTable(ddoracle, "tempTable" ) method for registering the table**

>>> ddoracle.registerTempTable("IDEA\_AGEING\_BADDEBT\_201415")

**Select Query:**

>>> oracleQuery=sqlContext.sql("SELECT \* FROM IDEA\_AGEING\_BADDEBT\_201415") ->Select the all column from the db

>>> oracleQuery.show() ->Show the first 20 column from the db

>>> oracleQuery=sqlContext.sql("SELECT COUNT(\*) FROM IDEA\_AGEING\_BADDEBT\_201415") ->Select the count of the record in the db

>>> oracleQuery=sqlContext.sql("SELECT BADDEBTGROUP,CATEGORY FROM IDEA\_AGEING\_BADDEBT\_201415")

**Display Query:**

>>> oracleQuery.show() ->Show first 20 record

>>>oracleQuery.collect() ->Show all the record from the db

**Group By:**

>>> oracleQuery=sqlContext.sql("SELECT BADDEBTGROUP FROM IDEA\_AGEING\_BADDEBT\_201415 GROUP BY BADDEBTGROUP")

>>> oracleQuery.show()

**Filter:**

>>> oracleQuery=sqlContext.sql("SELECT \* FROM IDEA\_AGEING\_BADDEBT\_201415 WHERE LAND\_LINE\_NO\_2 > 211253")

>>> oracleQuery.show()

>>> oracleQuery.count ->get the count of the data frame

<bound method DataFrame.count of DataFrame[c0: bigint]>

>>> oracleQuery.count() ->count() Returns the number of rows in this DataFrame.

1L

>>> oracleQuery.show()

c0

93959

>>> oracleQuery.collect() ->Get the all record count

[Row(c0=93959)]

>>> oracleQuery.explain() ->Display the processing DAG of the Query

Aggregate false, [], [Coalesce(SUM(PartialCount#83L),0) AS c0#69L]

Exchange SinglePartition

Aggregate true, [], [COUNT(1) AS PartialCount#83L]

PhysicalRDD [], JDBCRDD[110] at collect at <stdin>:1

>>> oracleQuery.isLocal ->flag where we can perform take() or collect() operation local or not

<bound method DataFrame.isLocal of DataFrame[c0: bigint]>

>>> oracleQuery.explain(True)

== Parsed Logical Plan ==

'Project [COUNT(1) AS c0#69L]

'UnresolvedRelation [IDEA\_AGEING\_BADDEBT\_201415], None

== Analyzed Logical Plan ==

Aggregate [], [COUNT(1) AS c0#69L]

Subquery IDEA\_AGEING\_BADDEBT\_201415

Relation[BDCLASS#0,BD#1,RISK#2,BADDEBTCLASS#3,BADDEBTGROUP#4,CIRCLE#5,MSISDN#6,CO\_CODE#7,CUST NUM#8,CUSTCODE#9,BT\_CUST\_ID#10,IMSI#11,CCNAME#12,AGENT\_CODE#13,SUB\_AGENT\_CODE#14,CORP\_CODE# USTOMER\_TYPE#16,CUSTOMER\_SUBS\_TYPE#17,MY\_IDEA\_CODE#18,SEGMENT#19,CSSEX#20,EMAILID#21,BIRTHD 22,BEGDATE#23,ENDDATE#24,CSTYPE#25,CO\_STATUS#26,CRM\_CONTRACT\_REASON#27,CRM\_INITIAL\_ACT\_REAS 8,TMCODE#29,RATEPLAN#30,DAYS\_00\_30#31,DAYS\_30\_60#32,DAYS\_60\_90#33,DAYS\_90\_120#34,DAYS\_120\_1 5,DAYS\_150\_180#36,DAYS\_180\_195#37,DAYS\_195\_210#38,DAYS\_210\_365#39,DAYS\_365\_MORE#40,BALANCE# REDIT\_LIMIT#42,BILL\_LINE1#43,BILL\_LINE2#44,BILL\_LINE3#45,BILL\_LINE4#46,BILL\_LINE5#47,LAND\_L NO#48,LAND\_LINE\_NO\_2#49,BILLCITY#50,BILLCYCLE#51,CCZIP#52,CATEGORY#53,UNBILLED\_AMOUNT#54,NE CURITY#55,SECURITY\_ADJUSTED#56,PAYMNTRESP#57,PAYMENT\_RESP\_CUSTCODE#58,CRM\_STATUS#59,CRM\_STA DT#60,CSACTIVATED#61,CSDEACTIVATED#62,CCCITY#63,CONNECTION\_TYPE#64,BILL\_MEDIA#65,ISNP#66,ID JDBCRelation(jdbc:oracle:thin:prasads/prasads@10.113.59.5:1521:ora11g,IDEA\_AGEING\_BADDEBT\_ 15,[Lorg.apache.spark.Partition;@2a4437,{url=jdbc:oracle:thin:prasads/prasads@10.113.59.5:1 ora11g, dbtable=IDEA\_AGEING\_BADDEBT\_201415, driver=oracle.jdbc.driver.OracleDriver, protoco in})

== Optimized Logical Plan ==

Aggregate [], [COUNT(1) AS c0#69L]

Project []

Relation[BDCLASS#0,BD#1,RISK#2,BADDEBTCLASS#3,BADDEBTGROUP#4,CIRCLE#5,MSISDN#6,CO\_CODE#7, NUM#8,CUSTCODE#9,BT\_CUST\_ID#10,IMSI#11,CCNAME#12,AGENT\_CODE#13,SUB\_AGENT\_CODE#14,CORP\_CODE# USTOMER\_TYPE#16,CUSTOMER\_SUBS\_TYPE#17,MY\_IDEA\_CODE#18,SEGMENT#19,CSSEX#20,EMAILID#21,BIRTHD 22,BEGDATE#23,ENDDATE#24,CSTYPE#25,CO\_STATUS#26,CRM\_CONTRACT\_REASON#27,CRM\_INITIAL\_ACT\_REAS 8,TMCODE#29,RATEPLAN#30,DAYS\_00\_30#31,DAYS\_30\_60#32,DAYS\_60\_90#33,DAYS\_90\_120#34,DAYS\_120\_1 5,DAYS\_150\_180#36,DAYS\_180\_195#37,DAYS\_195\_210#38,DAYS\_210\_365#39,DAYS\_365\_MORE#40,BALANCE# REDIT\_LIMIT#42,BILL\_LINE1#43,BILL\_LINE2#44,BILL\_LINE3#45,BILL\_LINE4#46,BILL\_LINE5#47,LAND\_L NO#48,LAND\_LINE\_NO\_2#49,BILLCITY#50,BILLCYCLE#51,CCZIP#52,CATEGORY#53,UNBILLED\_AMOUNT#54,NE CURITY#55,SECURITY\_ADJUSTED#56,PAYMNTRESP#57,PAYMENT\_RESP\_CUSTCODE#58,CRM\_STATUS#59,CRM\_STA DT#60,CSACTIVATED#61,CSDEACTIVATED#62,CCCITY#63,CONNECTION\_TYPE#64,BILL\_MEDIA#65,ISNP#66,ID JDBCRelation(jdbc:oracle:thin:prasads/prasads@10.113.59.5:1521:ora11g,IDEA\_AGEING\_BADDEBT\_ 15,[Lorg.apache.spark.Partition;@2a4437,{url=jdbc:oracle:thin:prasads/prasads@10.113.59.5:1 ora11g, dbtable=IDEA\_AGEING\_BADDEBT\_201415, driver=oracle.jdbc.driver.OracleDriver, protoco in})

== Physical Plan ==

Aggregate false, [], [Coalesce(SUM(PartialCount#83L),0) AS c0#69L]

Exchange SinglePartition

Aggregate true, [], [COUNT(1) AS PartialCount#83L]

PhysicalRDD [], JDBCRDD[110] at collect at <stdin>:1

Code Generation: false

== RDD ==

>>> oracleQuery.isLocal()

False

>>> ddoracle.groupBy(ddoracle.EMPNAME).count().collect() ->Group By on empname

[Row(EMPNAME=u'Umesh', count=1), Row(EMPNAME=u'kedar', count=1), Row(EMPNAME=u'Mahesh', count=1), Row(EMPNAME=u'Mayur', count=1), Row(EMPNAME=u'Anil', count=1), Row(EMPNAME=u'Yogesh', count=1), Row(EMPNAME=u'Kishor', count=1), Row(EMPNAME=u'Kiran', count=1)]

>>> ddoracle1.groupBy(ddoracle1.EMPNAME).count().collect()

[Row(EMPNAME=u'Kunal', count=1), Row(EMPNAME=u'Mahesh', count=1), Row(EMPNAME=u'Magesh', count=1), Row(EMPNAME=u'Keshav', count=1), Row(EMPNAME=u'Kirti', count=4), Row(EMPNAME=u'Kedar', count=1), Row(EMPNAME=u'Kiran', count=1)]

>>> ddoracle.groupBy(ddoracle.EMPNAME).count().distinct().collect() ->Distinct element from groupBy

[Row(EMPNAME=u'Umesh', count=1), Row(EMPNAME=u'kedar', count=1), Row(EMPNAME=u'Mahesh', count=1), Row(EMPNAME=u'Mayur', count=1), Row(EMPNAME=u'Anil', count=1), Row(EMPNAME=u'Yogesh', count=1), Row(EMPNAME=u'Kishor', count=1), Row(EMPNAME=u'Kiran', count=1)]

>>> ddoracle1.groupBy(ddoracle1.EMPNAME).count().distinct().collect()

[Row(EMPNAME=u'Kunal', count=1), Row(EMPNAME=u'Mahesh', count=1), Row(EMPNAME=u'Magesh', count=1), Row(EMPNAME=u'Keshav', count=1), Row(EMPNAME=u'Kirti', count=4), Row(EMPNAME=u'Kedar', count=1), Row(EMPNAME=u'Kiran', count=1)]

>>> ddoracle.dropna().show() ->dropna (how='any', thresh=None, subset=None) Returns a new DataFrame omitting rows with null values.DataFrame.dropna() and DataFrameNaFunctions.drop() are aliases of each other.

EMPNAME EMPAGE EMPSALARY EMPCOUNTRY

Mahesh 23 67890 India

kedar 24 697890 USA

Umesh 45 767890 India

Mayur 35 45676 China

Kishor 31 34258 USA

Yogesh 24 87864 India

Anil 30 3424 China

Kiran 24 12356 China

>>> ddoracle.count()

9L

>>> ddoracle.dtypes

[('EMPNAME', 'string'), ('EMPAGE', 'decimal(10,0)'), ('EMPSALARY', 'string'), ('EMPCOUNTRY', 'string')]

>>> ddoracle.fillna({'EMPAGE':10}).collect()

[Row(EMPNAME=u'Mahesh', EMPAGE=Decimal('23'), EMPSALARY=u'67890', EMPCOUNTRY=u'India'), Row(EMPNAME=u'kedar', EMPAGE=Decimal('24'), EMPSALARY=u'697890', EMPCOUNTRY=u'USA'), Row(EMPNAME=u'Umesh', EMPAGE=Decimal('45'), EMPSALARY=u'767890', EMPCOUNTRY=u'India'), Row(EMPNAME=u'Mayur', EMPAGE=Decimal('35'), EMPSALARY=u'45676', EMPCOUNTRY=u'China'), Row(EMPNAME=u'Kishor', EMPAGE=Decimal('31'), EMPSALARY=u'34258', EMPCOUNTRY=u'USA'), Row(EMPNAME=u'Yogesh', EMPAGE=Decimal('24'), EMPSALARY=u'87864', EMPCOUNTRY=u'India'), Row(EMPNAME=u'Anil', EMPAGE=Decimal('30'), EMPSALARY=u'3424', EMPCOUNTRY=u'China'), Row(EMPNAME=u'Kiran', EMPAGE=Decimal('24'), EMPSALARY=u'12356', EMPCOUNTRY=u'China'), Row(EMPNAME=u'Anil', EMPAGE=Decimal('89'), EMPSALARY=None, EMPCOUNTRY=None)]

>>> ddoracle.fillna({'EMPSALARY':1230}).collect() ->Insert the null value with default value

[Row(EMPNAME=u'Mahesh', EMPAGE=Decimal('23'), EMPSALARY=u'67890', EMPCOUNTRY=u'India'), Row(EMPNAME=u'kedar', EMPAGE=Decimal('24'), EMPSALARY=u'697890', EMPCOUNTRY=u'USA'), Row(EMPNAME=u'Umesh', EMPAGE=Decimal('45'), EMPSALARY=u'767890', EMPCOUNTRY=u'India'), Row(EMPNAME=u'Mayur', EMPAGE=Decimal('35'), EMPSALARY=u'45676', EMPCOUNTRY=u'China'), Row(EMPNAME=u'Kishor', EMPAGE=Decimal('31'), EMPSALARY=u'34258', EMPCOUNTRY=u'USA'), Row(EMPNAME=u'Yogesh', EMPAGE=Decimal('24'), EMPSALARY=u'87864', EMPCOUNTRY=u'India'), Row(EMPNAME=u'Anil', EMPAGE=Decimal('30'), EMPSALARY=u'3424', EMPCOUNTRY=u'China'), Row(EMPNAME=u'Kiran', EMPAGE=Decimal('24'), EMPSALARY=u'12356', EMPCOUNTRY=u'China'), Row(EMPNAME=u'Anil', EMPAGE=Decimal('89'), EMPSALARY=u'1230.0', EMPCOUNTRY=None)]

>>> ddoracle.fillna({'EMPCOUNTRY':'Unknown'}).collect() ->Insert the Unknown value to row empcountry if for any value is NULL

[Row(EMPNAME=u'Mahesh', EMPAGE=Decimal('23'), EMPSALARY=u'67890', EMPCOUNTRY=u'India'), Row(EMPNAME=u'kedar', EMPAGE=Decimal('24'), EMPSALARY=u'697890', EMPCOUNTRY=u'USA'), Row(EMPNAME=u'Umesh', EMPAGE=Decimal('45'), EMPSALARY=u'767890', EMPCOUNTRY=u'India'), Row(EMPNAME=u'Mayur', EMPAGE=Decimal('35'), EMPSALARY=u'45676', EMPCOUNTRY=u'China'), Row(EMPNAME=u'Kishor', EMPAGE=Decimal('31'), EMPSALARY=u'34258', EMPCOUNTRY=u'USA'), Row(EMPNAME=u'Yogesh', EMPAGE=Decimal('24'), EMPSALARY=u'87864', EMPCOUNTRY=u'India'), Row(EMPNAME=u'Anil', EMPAGE=Decimal('30'), EMPSALARY=u'3424', EMPCOUNTRY=u'China'), Row(EMPNAME=u'Kiran', EMPAGE=Decimal('24'), EMPSALARY=u'12356', EMPCOUNTRY=u'China'), Row(EMPNAME=u'Anil', EMPAGE=Decimal('89'), EMPSALARY=None, EMPCOUNTRY=u'Unknown')]

>>> ddoracle.join(ddoracle1,ddoracle.EMPAGE == ddoracle1.EMPAGE).collect() ->Join the two data frame(default: 'inner’. One of inner, outer, left\_outer ->left join, right\_outer ->right join, semijoin.)

[Row(EMPNAME=u'Mahesh', EMPAGE=Decimal('23'), EMPSALARY=Decimal('67890'), EMPCOUNTRY=u'India', EMPNAME=u'Mahesh', EMPAGE=Decimal('23'), EMPSALARY=Decimal('67890'), EMPCOUNTRY=u'India'), Row(EMPNAME=u'Kedar', EMPAGE=Decimal('24'), EMPSALARY=Decimal('3423'), EMPCOUNTRY=u'USA', EMPNAME=u'Kedar', EMPAGE=Decimal('24'), EMPSALARY=Decimal('3423'), EMPCOUNTRY=u'USA'), Row(EMPNAME=u'Kiran', EMPAGE=Decimal('24'), EMPSALARY=Decimal('23453'), EMPCOUNTRY=u'India', EMPNAME=u'Kiran', EMPAGE=Decimal('24'), EMPSALARY=Decimal('23453'), EMPCOUNTRY=u'India'), Row(EMPNAME=u'Kedar', EMPAGE=Decimal('24'), EMPSALARY=Decimal('3423'), EMPCOUNTRY=u'USA', EMPNAME=u'Kedar', EMPAGE=Decimal('24'), EMPSALARY=Decimal('3423'), EMPCOUNTRY=u'USA'), Row(EMPNAME=u'Kiran', EMPAGE=Decimal('24'), EMPSALARY=Decimal('23453'), EMPCOUNTRY=u'India', EMPNAME=u'Kiran', EMPAGE=Decimal('24'), EMPSALARY=Decimal('23453'), EMPCOUNTRY=u'India'), Row(EMPNAME=u'Kedar', EMPAGE=Decimal('24'), EMPSALARY=Decimal('3423'), EMPCOUNTRY=u'USA', EMPNAME=u'Kedar', EMPAGE=Decimal('24'), EMPSALARY=Decimal('3423'), EMPCOUNTRY=u'USA'), Row(EMPNAME=u'Kiran', EMPAGE=Decimal('24'), EMPSALARY=Decimal('23453'), EMPCOUNTRY=u'India', EMPNAME=u'Kiran', EMPAGE=Decimal('24'), EMPSALARY=Decimal('23453'), EMPCOUNTRY=u'India')]

>>> ddoracle.join(ddoracle1,ddoracle.EMPAGE == ddoracle1.EMPAGE).count()

7L

>>> ddoracle.join(ddoracle1,ddoracle.EMPAGE == ddoracle1.EMPAGE,'outer').collect()

[Row(EMPNAME=u'Kirti', EMPAGE=Decimal('27'), EMPSALARY=Decimal('434334'), EMPCOUNTRY=u'USA', EMPNAME=u'Kirti', EMPAGE=Decimal('27'), EMPSALARY=Decimal('434334'), EMPCOUNTRY=u'USA'), Row(EMPNAME=u'Keshav', EMPAGE=Decimal('34'), EMPSALARY=Decimal('2324243'), EMPCOUNTRY=u'USA', EMPNAME=u'Keshav', EMPAGE=Decimal('34'), EMPSALARY=Decimal('2324243'), EMPCOUNTRY=u'USA'), Row(EMPNAME=None, EMPAGE=None, EMPSALARY=None, EMPCOUNTRY=None, EMPNAME=None, EMPAGE=None, EMPSALARY=None, EMPCOUNTRY=None), Row(EMPNAME=u'Mahesh', EMPAGE=Decimal('23'), EMPSALARY=Decimal('67890'), EMPCOUNTRY=u'India', EMPNAME=u'Mahesh', EMPAGE=Decimal('23'), EMPSALARY=Decimal('67890'), EMPCOUNTRY=u'India'), Row(EMPNAME=None, EMPAGE=None, EMPSALARY=None, EMPCOUNTRY=None, EMPNAME=None, EMPAGE=None, EMPSALARY=None, EMPCOUNTRY=None), Row(EMPNAME=u'Kedar', EMPAGE=Decimal('24'), EMPSALARY=Decimal('3423'), EMPCOUNTRY=u'USA', EMPNAME=u'Kedar', EMPAGE=Decimal('24'), EMPSALARY=Decimal('3423'), EMPCOUNTRY=u'USA'), Row(EMPNAME=u'Kiran', EMPAGE=Decimal('24'), EMPSALARY=Decimal('23453'), EMPCOUNTRY=u'India', EMPNAME=u'Kiran', EMPAGE=Decimal('24'), EMPSALARY=Decimal('23453'), EMPCOUNTRY=u'India'), Row(EMPNAME=u'Kedar', EMPAGE=Decimal('24'), EMPSALARY=Decimal('3423'), EMPCOUNTRY=u'USA', EMPNAME=u'Kedar', EMPAGE=Decimal('24'), EMPSALARY=Decimal('3423'), EMPCOUNTRY=u'USA'), Row(EMPNAME=u'Kiran', EMPAGE=Decimal('24'), EMPSALARY=Decimal('23453'), EMPCOUNTRY=u'India', EMPNAME=u'Kiran', EMPAGE=Decimal('24'), EMPSALARY=Decimal('23453'), EMPCOUNTRY=u'India'), Row(EMPNAME=u'Kedar', EMPAGE=Decimal('24'), EMPSALARY=Decimal('3423'), EMPCOUNTRY=u'USA', EMPNAME=u'Kedar', EMPAGE=Decimal('24'), EMPSALARY=Decimal('3423'), EMPCOUNTRY=u'USA'), Row(EMPNAME=u'Kiran', EMPAGE=Decimal('24'), EMPSALARY=Decimal('23453'), EMPCOUNTRY=u'India', EMPNAME=u'Kiran', EMPAGE=Decimal('24'), EMPSALARY=Decimal('23453'), EMPCOUNTRY=u'India'), Row(EMPNAME=None, EMPAGE=None, EMPSALARY=None, EMPCOUNTRY=None, EMPNAME=None, EMPAGE=None, EMPSALARY=None, EMPCOUNTRY=None), Row(EMPNAME=None, EMPAGE=None, EMPSALARY=None, EMPCOUNTRY=None, EMPNAME=None, EMPAGE=None, EMPSALARY=None, EMPCOUNTRY=None), Row(EMPNAME=u'Kunal', EMPAGE=Decimal('25'), EMPSALARY=Decimal('343423'), EMPCOUNTRY=u'China', EMPNAME=u'Kunal', EMPAGE=Decimal('25'), EMPSALARY=Decimal('343423'), EMPCOUNTRY=u'China'), Row(EMPNAME=u'Magesh', EMPAGE=Decimal('25'), EMPSALARY=Decimal('434334'), EMPCOUNTRY=u'India', EMPNAME=u'Magesh', EMPAGE=Decimal('25'), EMPSALARY=Decimal('434334'), EMPCOUNTRY=u'India'), Row(EMPNAME=u'Kirti', EMPAGE=Decimal('25'), EMPSALARY=Decimal('434334'), EMPCOUNTRY=u'India', EMPNAME=u'Kirti', EMPAGE=Decimal('25'), EMPSALARY=Decimal('434334'), EMPCOUNTRY=u'India'), Row(EMPNAME=u'Kirti', EMPAGE=Decimal('25'), EMPSALARY=Decimal('434334'), EMPCOUNTRY=u'India', EMPNAME=u'Kirti', EMPAGE=Decimal('25'), EMPSALARY=Decimal('434334'), EMPCOUNTRY=u'India'), Row(EMPNAME=u'Kirti', EMPAGE=Decimal('25'), EMPSALARY=Decimal('434334'), EMPCOUNTRY=u'China', EMPNAME=u'Kirti', EMPAGE=Decimal('25'), EMPSALARY=Decimal('434334'), EMPCOUNTRY=u'China'), Row(EMPNAME=None, EMPAGE=None, EMPSALARY=None, EMPCOUNTRY=None, EMPNAME=None, EMPAGE=None, EMPSALARY=None, EMPCOUNTRY=None)]

>>> ddoracle.join(ddoracle1,ddoracle.EMPAGE == ddoracle1.EMPAGE,'outer').count()

19L

>>> ddoracle.join(ddoracle1,ddoracle.EMPAGE == ddoracle1.EMPAGE,'left\_outer').count()

12L

>>> ddoracle.join(ddoracle1,ddoracle.EMPAGE == ddoracle1.EMPAGE,'right\_outer').count()

14L

>>> sqlContext.tableNames()

[]

>>> sqlContext.registerDataFrameAsTable(ddoracle,"tempTable") ->Register data frame as table

>>> df1=sqlContext.tables() ->tables(dbName=None) Returns a DataFrame containing names of tables in the given database

>>> df1

DataFrame[tableName: string, isTemporary: boolean]

>>> df1.printSchema()

root

|-- tableName: string (nullable = false)

|-- isTemporary: boolean (nullable = false)

>>> df1.count

<bound method DataFrame.count of DataFrame[tableName: string, isTemporary: boolean]>

>>> df1.count()

1L

>>> df1.collect()

[Row(tableName=u'tempTable', isTemporary=True)]

>>> ddoracle.registerTempTable("tempTable1")

>>> df1=sqlContext.tables()

>>> df1.count

<bound method DataFrame.count of DataFrame[tableName: string, isTemporary: boolean]>

>>> df1.count()

2L

>>> df1.collect()

[Row(tableName=u'tempTable', isTemporary=True), Row(tableName=u'tempTable1', isTemporary=True)]

>>> ddoracle.sort(ddoracle.EMPCOUNTRY).collect() ->(Returns a new DataFrame sorted by the specified column(s). Currently supported onyl string sorting for other type getting Error Caused by: java.lang.ClassCastException: java.math.BigDecimal cannot be cast to org.apache.spark.sql.types.Decimal)

[Row(EMPNAME=u'Anil', EMPAGE=Decimal('89'), EMPSALARY=None, EMPCOUNTRY=None), Row(EMPNAME=u'Mayur', EMPAGE=Decimal('35'), EMPSALARY=u'45676', EMPCOUNTRY=u'China'), Row(EMPNAME=u'Anil', EMPAGE=Decimal('30'), EMPSALARY=u'3424', EMPCOUNTRY=u'China'), Row(EMPNAME=u'Kiran', EMPAGE=Decimal('24'), EMPSALARY=u'12356', EMPCOUNTRY=u'China'), Row(EMPNAME=u'Mahesh', EMPAGE=Decimal('23'), EMPSALARY=u'67890', EMPCOUNTRY=u'India'), Row(EMPNAME=u'Umesh', EMPAGE=Decimal('45'), EMPSALARY=u'767890', EMPCOUNTRY=u'India'), Row(EMPNAME=u'Yogesh', EMPAGE=Decimal('24'), EMPSALARY=u'87864', EMPCOUNTRY=u'India'), Row(EMPNAME=u'kedar', EMPAGE=Decimal('24'), EMPSALARY=u'697890', EMPCOUNTRY=u'USA'), Row(EMPNAME=u'Kishor', EMPAGE=Decimal('31'), EMPSALARY=u'34258', EMPCOUNTRY=u'USA')]

>>> ddoracle.cube('EMPNAME',ddoracle.EMPAGE).count().show() ->Create a multi-dimensional cube for the current DataFrame using the specified columns, so we can run aggregation on them.

+-------+------+-----+

|EMPNAME|EMPAGE|count|

+-------+------+-----+

| Yogesh| null| 1|

| null| 23| 1|

| kedar| null| 1|

| null| 45| 1|

| Kishor| null| 1|

| null| 24| 3|

| null| 89| 1|

| null| 35| 1|

| Umesh| null| 1|

| Mayur| null| 1|

| Kiran| 24| 1|

| null| 30| 1|

| Anil| 89| 1|

| Yogesh| 24| 1|

| Anil| 30| 1|

| null| 31| 1|

| kedar| 24| 1|

| Mahesh| null| 1|

| Umesh| 45| 1|

| null| null| 9|

+-------+------+-----+

>>> ddoracle.cube('EMPNAME').count().show() ->Create a multi-dimensional cube for the current DataFrame using the specified columns, so we can run aggregation on them.

+-------+-----+

|EMPNAME|count|

+-------+-----+

| Umesh| 1|

| null| 9|

| Mahesh| 1|

| kedar| 1|

| Kishor| 1|

| Yogesh| 1|

| Kiran| 1|

| Mayur| 1|

| Anil| 2|

+-------+-----+

>>> ddoracle.drop('EMPSALARY').show() ->Drop particular column from the data frame(Returns a new DataFrame that drops the specified column.Parameters: col – a string name of the column to drop, or a Column to drop.)

+-------+------+----------+

|EMPNAME|EMPAGE|EMPCOUNTRY|

+-------+------+----------+

| Mahesh| 23| India|

| kedar| 24| USA|

| Umesh| 45| India|

| Mayur| 35| China|

| Kishor| 31| USA|

| Yogesh| 24| India|

| Anil| 30| China|

| Kiran| 24| China|

| Anil| 89| null|

+-------+------+----------+

>>> ddoracle.describe('EMPSALARY').show() ->(Computes statistics for numeric columns.

This include count, mean, stddev, min, and max. If no columns are given, this function computes statistics for all numerical columns.)

+-------+------------------+

|summary| EMPSALARY|

+-------+------------------+

| count| 8|

| mean| 214656.0|

| stddev|300799.21753555146|

| min| 12356|

| max| 87864|

+-------+------------------+

>>> ddoracle.describe().show()

+-------+----------------+

|summary| EMPAGE|

+-------+----------------+

| count| 9|

| mean| 36.1111|

| stddev|19.8687255955182|

| min| 23|

| max| 89|

+-------+----------------+

>>> ddoracle.describe('empsalary','empage').show()

+-------+------------------+-----------------+

|summary| empsalary| empage|

+-------+------------------+-----------------+

| count| 12| 12|

| mean| 1107574.0| 5.75|

| stddev|1857363.9454459825|4.085033659592048|

| min| 25634| 1|

| max| 9874| 13|

+-------+------------------+-----------------+

>>> ddoracle.dropDuplicates().show() ->Return a new DataFrame with duplicate rows removed, optionally only considering certain columns.

+-------+------+---------+----------+

|EMPNAME|EMPAGE|EMPSALARY|EMPCOUNTRY|

+-------+------+---------+----------+

| Kishor| 31| 34258| USA|

| Anil| 30| 3424| China|

| Umesh| 45| 767890| India|

| Anil| 89| null| null|

| Yogesh| 24| 87864| India|

| kedar| 24| 697890| USA|

| Mayur| 35| 45676| China|

| Mahesh| 23| 67890| India|

| Kiran| 24| 12356| China|

+-------+------+---------+----------+

>>> ddoracle.describe('EMPNAME').show() ->Describe on String data type

+-------+-------+

|summary|EMPNAME|

+-------+-------+

| count| 9|

| mean| null|

| stddev| null|

| min| Anil|

| max| kedar|

+-------+-------+

>>> ddoracle.dropDuplicates(['EMPCOUNTRY']).show()

+-------+------+---------+----------+

|EMPNAME|EMPAGE|EMPSALARY|EMPCOUNTRY|

+-------+------+---------+----------+

| Mahesh| 23| 67890| India|

| kedar| 24| 697890| USA|

| Mayur| 35| 45676| China|

| Anil| 89| null| null|

+-------+------+---------+----------+

**Aggregate:**

>>> ddoracle.groupBy("empcountry").agg({"empsalary":"min"}).collect()

[Row(empcountry=u'India', MIN(empsalary)=u'26872'), Row(empcountry=u'USA', MIN(empsalary)=u'4324342'), Row(empcountry=u'China', MIN(empsalary)=u'25634')]

ddoracle.groupBy("empcountry").agg({"empsalary":"avg"}).collect()

[Row(empcountry=u'India', AVG(empsalary)=2175607.0), Row(empcountry=u'USA', AVG(empsalary)=894356.0), Row(empcountry=u'China', AVG(empsalary)=893578.8)]

>>> ddoracle.groupBy("empcountry").agg({"empsalary":"max"}).collect()

[Row(empcountry=u'India', MAX(empsalary)=u'4324342'), Row(empcountry=u'USA', MAX(empsalary)=u'9874'), Row(empcountry=u'China', MAX(empsalary)=u'87642')]

>>> ddoracle1.groupBy("CRM\_STATUS\_DT").agg({"UNBILLED\_AMOUNT":"sum"}).collect() ->Group on datetime field

**Convert the data frame into the RDD:**

>>> rdddata=oracleQuery.rdd

>>> rdddata

PythonRDD[127] at RDD at PythonRDD.scala:43

>>> rdddata.count

<bound method PipelinedRDD.count of PythonRDD[127] at RDD at PythonRDD.scala:43>

>>> rdddata.count() ->Get the total element in the RDD

1

>>> rdd.getNumPartitions()

1

**Convert the rdd into the dataframe :**

>>> df2 = sqlContext.createDataFrame(rdddata)

>>> df2.collect()

[Row(EMPNAME=u'Mahesh', EMPAGE=Decimal('23'), EMPSALARY=u'67890', EMPCOUNTRY=u'India'), Row(EMPNAME=u'kedar', EMPAGE=Decimal('24'), EMPSALARY=u'697890', EMPCOUNTRY=u'USA'), Row(EMPNAME=u'Umesh', EMPAGE=Decimal('45'), EMPSALARY=u'767890', EMPCOUNTRY=u'India'), Row(EMPNAME=u'Mayur', EMPAGE=Decimal('35'), EMPSALARY=u'45676', EMPCOUNTRY=u'China'), Row(EMPNAME=u'Kishor', EMPAGE=Decimal('31'), EMPSALARY=u'34258', EMPCOUNTRY=u'USA'), Row(EMPNAME=u'Yogesh', EMPAGE=Decimal('24'), EMPSALARY=u'87864', EMPCOUNTRY=u'India'), Row(EMPNAME=u'Anil', EMPAGE=Decimal('30'), EMPSALARY=u'3424', EMPCOUNTRY=u'China'), Row(EMPNAME=u'Kiran', EMPAGE=Decimal('24'), EMPSALARY=u'12356', EMPCOUNTRY=u'China')]

>>> sqlContext.createDataFrame(rdd).collect()

[Row(EMPNAME=u'Mahesh', EMPAGE=Decimal('23'), EMPSALARY=u'67890', EMPCOUNTRY=u'India'), Row(EMPNAME=u'kedar', EMPAGE=Decimal('24'), EMPSALARY=u'697890', EMPCOUNTRY=u'USA'), Row(EMPNAME=u'Umesh', EMPAGE=Decimal('45'), EMPSALARY=u'767890', EMPCOUNTRY=u'India'), Row(EMPNAME=u'Mayur', EMPAGE=Decimal('35'), EMPSALARY=u'45676', EMPCOUNTRY=u'China'), Row(EMPNAME=u'Kishor', EMPAGE=Decimal('31'), EMPSALARY=u'34258', EMPCOUNTRY=u'USA'), Row(EMPNAME=u'Yogesh', EMPAGE=Decimal('24'), EMPSALARY=u'87864', EMPCOUNTRY=u'India'), Row(EMPNAME=u'Anil', EMPAGE=Decimal('30'), EMPSALARY=u'3424', EMPCOUNTRY=u'China'), Row(EMPNAME=u'Kiran', EMPAGE=Decimal('24'), EMPSALARY=u'12356', EMPCOUNTRY=u'China')]

**RDD Function:**

>>>from pyspark.sql import SQLContext

>>>sqlContext = SQLContext(sc)

>>>ddoracle = sqlContext.load(source="jdbc", driver="oracle.jdbc.driver.OracleDriver",url="jdbc:oracle:thin:prasads/prasads@10.113.59.5:1521:ora11g", dbtable="IDEA\_AGEING\_BADDEBT\_201415")

>>> rdd=ddoracle.rdd

>>> rdd

PythonRDD[127] at RDD at PythonRDD.scala:43

>>> rdd.count()

93959

>>> rdd.count

<bound method PipelinedRDD.count of PythonRDD[127] at RDD at PythonRDD.scala:43>

>>> rdd.cache()

PythonRDD[11] at RDD at PythonRDD.scala:43

>>> rdd.context

<pyspark.context.SparkContext object at 0xa25dc4c>

>>> rdd.countApprox(1000, 1.0)

93959

>>> rdd.countApproxDistinct()

90434L

>>> rdd.countByKey()

defaultdict(<type 'int'>, {Decimal('1'): 93959})

>>> rdd.isCheckpointed()

False

>>> rdd.isEmpty()

15/06/15 15:00:09 WARN SizeEstimator: Failed to check whether UseCompressedOops is set; assuming yes

False

>>> rdd.getNumPartitions()

1

>>> rdd.first()

Row(BDCLASS=Decimal('1'), BD=u'BadDebt', RISK=Decimal('1'), BADDEBTCLASS=u'90-', BADDEBTGROUP=u'365\_MORE', CIRCLE=u'MH', MSISDN=u'919527003791', CO\_CODE=Decimal('1000011538836'), CUSTNUM=Decimal('100011125394'), CUSTCODE=u'1.16518031', BT\_CUST\_ID=u'1.16518031', IMSI=Decimal('404220002684723'), CCNAME=None, AGENT\_CODE=u'500154', SUB\_AGENT\_CODE=u'500154\_1', CORP\_CODE=None, CUSTOMER\_TYPE=u'Individual', CUSTOMER\_SUBS\_TYPE=u'Retail', MY\_IDEA\_CODE=None, SEGMENT=u'Other', CSSEX=u'Male', EMAILID=u'AKSHAY98@GMAIL.COM', BIRTHDATE=datetime.date(1988, 8, 18), BEGDATE=datetime.date(2013, 11, 14), ENDDATE=datetime.date(2014, 3, 26), CSTYPE=u'd', CO\_STATUS=u'd', CRM\_CONTRACT\_REASON=u'Non- Payment', CRM\_INITIAL\_ACT\_REASON=u'New Connection', TMCODE=u'5482', RATEPLAN=u'T1227-3G NS 750 QOS (4GB UL)', DAYS\_00\_30=Decimal('0'), DAYS\_30\_60=Decimal('0'), DAYS\_60\_90=Decimal('0'), DAYS\_90\_120=Decimal('0'), DAYS\_120\_150=Decimal('0'), DAYS\_150\_180=Decimal('0'), DAYS\_180\_195=Decimal('0'), DAYS\_195\_210=Decimal('0'), DAYS\_210\_365=Decimal('190.28'), DAYS\_365\_MORE=Decimal('1246.09'), BALANCE=Decimal('1436.37'), CREDIT\_LIMIT=Decimal('1000'), BILL\_LINE1=None, BILL\_LINE2=u'Mr. AKSHAY SUDHIR SAKORE', BILL\_LINE3=u'A 6 BHAGYASHREE APPARTMENT', BILL\_LINE4=u'NR WATER TANK SANT TUKARAM NAGAR', BILL\_LINE5=u'DIGHI ROAD BHOSARI PUNE', LAND\_LINE\_NO=None, LAND\_LINE\_NO\_2=u'9923060439', BILLCITY=u'PUNE', BILLCYCLE=u'083', CCZIP=u'411039', CATEGORY=u'MAHRegular Price Group', UNBILLED\_AMOUNT=u'0', NEW\_SECURITY=Decimal('0'), SECURITY\_ADJUSTED=Decimal('0'), PAYMNTRESP=u'X', PAYMENT\_RESP\_CUSTCODE=u'100011125394', CRM\_STATUS=u'PD', CRM\_STATUS\_DT=datetime.date(2014, 3, 26), CSACTIVATED=datetime.date(2013, 11, 13), CSDEACTIVATED=u'02-JUL-2014', CCCITY=u'PUNE', CONNECTION\_TYPE=u'IOIP', BILL\_MEDIA=None, ISNP=Decimal('1'), ID=Decimal('115820'))

>>> rdd.name

<bound method PipelinedRDD.name of PythonRDD[4] at RDD at PythonRDD.scala:43>

>>> rdd.getStorageLevel()

StorageLevel(False, False, False, False, 1)

>>> print(rdd.getStorageLevel())

Serialized 1x Replicated

>>> rdd.collectAsMap()

>>> rdd.coalesce(1).glom().collect()

[[Row(EMPNAME=u'Mahesh', EMPAGE=Decimal('23'), EMPSALARY=u'67890', EMPCOUNTRY=u'India'), Row(EMPNAME=u'kedar', EMPAGE=Decimal('24'), EMPSALARY=u'697890', EMPCOUNTRY=u'USA'), Row(EMPNAME=u'Umesh', EMPAGE=Decimal('45'), EMPSALARY=u'767890', EMPCOUNTRY=u'India'), Row(EMPNAME=u'Mayur', EMPAGE=Decimal('35'), EMPSALARY=u'45676', EMPCOUNTRY=u'China'), Row(EMPNAME=u'Kishor', EMPAGE=Decimal('31'), EMPSALARY=u'34258', EMPCOUNTRY=u'USA'), Row(EMPNAME=u'Yogesh', EMPAGE=Decimal('24'), EMPSALARY=u'87864', EMPCOUNTRY=u'India'), Row(EMPNAME=u'Anil', EMPAGE=Decimal('30'), EMPSALARY=u'3424', EMPCOUNTRY=u'China'), Row(EMPNAME=u'Kiran', EMPAGE=Decimal('24'), EMPSALARY=u'12356', EMPCOUNTRY=u'China')]]

>>> rdd.distinct().collect()

>>> rdd.countByKey()

defaultdict(<type 'int'>, {u'Keshav': 1, u'Kunal': 1, u'Kirti': 4, u'Kiran': 1, u'Mahesh': 1, u'Magesh': 1, u'Kedar': 1})

>>> rdd.countByKey().items()

[(u'Keshav', 1), (u'Kunal', 1), (u'Kirti', 4), (u'Kiran', 1), (u'Mahesh', 1), (u'Magesh', 1), (u'Kedar', 1)]

>>> rdd.countByValue().items()

[(Row(EMPNAME=u'Magesh', EMPAGE=Decimal('25'), EMPSALARY=Decimal('434334'), EMPCOUNTRY=u'India'), 1), (Row(EMPNAME=u'Kiran', EMPAGE=Decimal('24'), EMPSALARY=Decimal('23453'), EMPCOUNTRY=u'India'), 1), (Row(EMPNAME=u'Kunal', EMPAGE=Decimal('25'), EMPSALARY=Decimal('343423'), EMPCOUNTRY=u'China'), 1), (Row(EMPNAME=u'Keshav', EMPAGE=Decimal('34'), EMPSALARY=Decimal('2324243'), EMPCOUNTRY=u'USA'), 1), (Row(EMPNAME=u'Kirti', EMPAGE=Decimal('27'), EMPSALARY=Decimal('434334'), EMPCOUNTRY=u'USA'), 1), (Row(EMPNAME=u'Kirti', EMPAGE=Decimal('25'), EMPSALARY=Decimal('434334'), EMPCOUNTRY=u'China'), 1), (Row(EMPNAME=u'Mahesh', EMPAGE=Decimal('23'), EMPSALARY=Decimal('67890'), EMPCOUNTRY=u'India'), 1), (Row(EMPNAME=u'Kirti', EMPAGE=Decimal('25'), EMPSALARY=Decimal('434334'), EMPCOUNTRY=u'India'), 2), (Row(EMPNAME=u'Kedar', EMPAGE=Decimal('24'), EMPSALARY=Decimal('3423'), EMPCOUNTRY=u'USA'), 1)]

>>> rdd.groupBy(lambda x:x.EMPCOUNTRY).collect()

[(u'India', <pyspark.resultiterable.ResultIterable object at 0xa3a99cc>), (u'China', <pyspark.resultiterable.ResultIterable object at 0xa3aa8cc>), (u'USA', <pyspark.resultiterable.ResultIterable object at 0xa3a9e2c>)]

>>> rdd.groupBy(lambda x:x.EMPCOUNTRY).reduceByKey(add).collect()

[(u'India', <pyspark.resultiterable.ResultIterable object at 0xa3b0c2c>), (u'China', <pyspark.resultiterable.ResultIterable object at 0xa31d92c>), (u'USA', <pyspark.resultiterable.ResultIterable object at 0xa3a9e0c>)]

>>> sorted(rdd.countByKey().items())

[(Decimal('1'), 93959)]

**Cache the data frame in spark:**

sqlContext.cacheTable("tableName")

or

dataFrame.cache()

E.g:

sqlContext.cacheTable("IDEA\_AGEING\_BADDEBT\_201415")

or

ddoracle.cache()

>>> ddoracle.saveAsTable("sampleTable",source="jdbc",mode="overwrite")

**Reference Site:**

Function available for data frame:

Scala API:

<https://spark.apache.org/docs/1.3.1/api/scala/index.html#org.apache.spark.sql.DataFrame>

Python API:

<https://spark.apache.org/docs/1.3.1/api/python/pyspark.sql.html>

Other:

<http://apache-spark-user-list.1001560.n3.nabble.com/saveAsTable-fails-to-save-RDD-in-Spark-SQL-1-3-0-td22108.html>

<https://www.mail-archive.com/user@spark.apache.org/msg28310.html>

<http://www.sparkexpert.com/2015/03/28/loading-database-data-into-spark-using-data-sources-api/>

<http://www.river-of-bytes.com/2014/12/filtering-and-projection-in-spark-sql.html>

<https://github.com/sujee81/SparkApps> ->Spark java code for loading SQL DataBase into the spark

<http://www.sparkexpert.com/2015/04/17/save-apache-spark-dataframe-to-database/>

<http://blog.cloudera.com/blog/2015/03/how-to-tune-your-apache-spark-jobs-part-2/> ->How to reduce the execution time configuration settings

<https://spark.apache.org/docs/latest/configuration.html> ->Spark Apache Configuration

<https://spark.apache.org/docs/latest/api/python/pyspark.sql.html#pyspark.sql.DataFrame> ->DataFrame APIs(Python)

<https://spark.apache.org/docs/latest/api/python/pyspark.html#pyspark.RDD>

->RDD API(Python)

<http://www.infoq.com/articles/apache-spark-introduction>

<https://spark.apache.org/> ->Spark official site

<http://www.informationweek.com/big-data/big-data-analytics/apache-spark-3-promising-use-cases/a/d-id/1319660>

<http://www.toptal.com/spark/introduction-to-apache-spark>

<http://databricks.gitbooks.io/databricks-spark-reference-applications/content/logs_analyzer/chapter1/spark.html> ->Nice Article

<http://databricks.gitbooks.io/databricks-spark-reference-applications/content/logs_analyzer/chapter1/sql.html> ->SQL Query Java API

<http://apache-spark-user-list.1001560.n3.nabble.com/> ->Spark user mailing list

<http://databricks.gitbooks.io/databricks-spark-reference-applications/content/logs_analyzer/chapter1/sql.html> ->SQL Query Java API

<http://spark.apache.org/docs/1.4.0/tuning.html#memory-tuning> ->Memory Tuning for spark 1.4.0

<http://spark.apache.org/docs/1.4.0/configuration.html#scheduling> ->Spark Performance tuning parameter configuration

<https://spark.apache.org/docs/0.9.1/api/core/index.html#org.apache.spark.storage.StorageLevel$> ->Storage level in spark

<http://sujee.net/2015/01/22/understanding-spark-caching/#.VYP9CPmqouQ> -> spark caching basic

<https://forums.databricks.com/questions/271/should-i-always-cache-my-rdds.html> ->When to cache dataframe and RDD

**How to start the Spark standalone cluster**

**Setup the Spark as standalone mode for deployment Mode (Master-slave):**

First go to the spark directory (e.g : cd /home/binuk/Projects/Spark/spark-1.3.1-bin-hadoop2.6)

1. Start the master node:

./sbin/start-master.sh

URL: MachineIP:8080/8081

e.g:

http://10.113.41.99:8081/

1. Start the Worker Node:

.bin/spark-class org.apache.spark.deploy.worker.Worker spark://localhost.localdomain:7077

Where localhost.localdomain:7077 : URL of the master where spark is running that you can see by Master Node URL(e.g: <http://10.113.41.99:8081/>)

**Spark Configuration parameter:**

You can modify the configuration parameter for the spark by modifying the

spark-1.3.1-bin-hadoop2.6/conf/spark-defaults.conf file.

Set the following parameter in the file for memory.

**spark.driver.memory** 1g //set the memory for driver(Default ->512m)

**spark.executor.memory** 900m

**Spark Memory Configuration :(conf/spark-defaults.conf)**

**spark.driver.memory** ->Driver memory

**spark.executor.memory** -> Executor Memory(worker)

**spark.storage.memoryFraction** (Default->0.6)

**spark.storage.safetyFraction** (Default->0.9)

**spark.executor.cores** 3

**spark.executor.instances** (configuration property control the number of executors requested)

**spark.shuffle.spill** false

**spark.dynamicAllocation.enabled** true/false(Dynamic allocation enables a Spark application to request executors when there is a backlog of pending tasks and free up executors when idle)

**spark.default.parallelism** none or integer value(level of parallelism ) [recommend 2-3 tasks per CPU core in your cluster]

**spark.serializer** property controls the serializer that’s used to convert between a deserialized Java object representation and a serialized binary representation[org.apache.spark.serializer.KryoSerializer ->KryoSerializer or else default java serializer]

**spark.kryoserializer.buffer** buffer value for kryoserializer (default->2)

Note: For more spark configuration setting see the refer link. <https://spark.apache.org/docs/latest/configuration.html>

**How to submit the job to spark:**

**Use the following command to submit the spark job:**

./bin/spark-submit --class com.sparkexpert.OracleMainJDBCConnect --master local[1] /home/binuk/Projects/Spark/SparkProgramJar/spark-load-from-db-1.0-SNAPSHOT.jar --driver-memory 1g --executor-memory 800m --executor-cores 8 --num-executors 17

Local[1] ->Number of thread use for the worker

**Performance Tuning for Oracle DB:**

|  |  |  |  |
| --- | --- | --- | --- |
| **Time Comparison measure for different load database method** | | | |
| Query Type | Oracle Query-93959 record(show) | DataFrame(Load Command)(show) | collect |
| Select particular column | 11 (s) | 0.1(s) | 5(s) |
| Select \* | 1m | 0.6(s) | 1.1m |
| Group By | 22(s) | 5(s) | 8(s) |
| Filter | 35(s) | 9(s) | 56(s) |
| count | 35(s) | 18(s) | 35(s) |
| sort | 56(s) |  | 56(s) |
| Join | 4(s) | 4(s) | 4(s) |
| Distinct | 3(s) | 3(s) | 3(s) |
| Aggregate | 3(s) ->Only count on string | 3(s) | 3(s) |
| Cube | 2(s) | 2(s) | 2(s) |

**Cache Mechanism in the Spark:**

|  |  |  |
| --- | --- | --- |
| **Cache in the spark** | | |
| **Parameter** | **Row Caching** | **Serialized Cache** |
| Memory taking for Data storing(Storage) | 57.5 MB | 57.5 MB |
| Processing data(Time)[Collect() on DataFrame) | 4.5 min | 6.0 min |

**Understanding Spark Caching:**

Spark excels at processing in-memory data.  We are going to look at various caching options and their effects, and (hopefully) provide some tips for optimizing Spark memory caching.

When caching in Spark, there are two options

1. Raw storage
2. Serialized

Here are some differences between the two options

|  |  |
| --- | --- |
| **Raw caching** | **Serialized Caching** |
| Pretty fast to process | Slower processing than raw caching |
| Can take up 2x-4x more spaceFor example, 100MB data cached could consume 350MB memory | Overhead is minimal |
| can put pressure in JVM and JVM garbage collection | less pressure |
| usage:rdd.persist( StorageLevel.MEMORY\_ONLY)  or  rdd.cache() | usage:rdd.persist( StorageLevel.MEMORY\_ONLY\_SER) |

All Storage level available in spark:

<https://spark.apache.org/docs/latest/api/scala/index.html#org.apache.spark.storage.StorageLevel$>

**JAVA DataFrame API for Spark:**

Sample Example:

package com.sparkexpert;

import java.io.Serializable;

import java.util.HashMap;

import java.util.List;

import java.util.Map;

import org.apache.spark.SparkConf;

import org.apache.spark.api.java.JavaSparkContext;

import org.apache.spark.sql.DataFrame;

import org.apache.spark.sql.Row;

import org.apache.spark.sql.SQLContext;

public class OracleMainJDBCConnect implements Serializable

{

private static final org.apache.log4j.Logger LOGGER = org.apache.log4j.Logger.getLogger( OracleMainJDBCConnect.class );

private static final String ORACLE\_DRIVER = "oracle.jdbc.driver.OracleDriver";

private static final String ORACLE\_USERNAME = "prasads";

private static final String ORACLE\_PWD = "prasads";

private static final String ORACLE\_DB\_MACHINE\_IP = "10.113.59.5";

private static final String ORACLE\_DB\_MACHINE\_PORT = "1521";

private static final String COLON = ":";

private static final String SEPARATOR = "/";

private static final String ATTHERATE = "@";

private static final String ORACLE\_VERSION = "ora11g";

private static final String ORACLE\_CONNECTION\_URL = "jdbc:oracle:thin:" + ORACLE\_USERNAME + SEPARATOR + ORACLE\_PWD + ATTHERATE + ORACLE\_DB\_MACHINE\_IP + COLON + ORACLE\_DB\_MACHINE\_PORT + COLON + ORACLE\_VERSION;

private static final JavaSparkContext sc = new JavaSparkContext( new SparkConf().setAppName( "SparkJdbcDs" ).setMaster( "local[\*]" ).set( "spark.executor.memory", "1g" ) );

private static final SQLContext sqlContext = new SQLContext( sc );

public static void main( String[] args )

{

// Data source options

Map<String, String> options = new HashMap<>();

options.put( "driver", ORACLE\_DRIVER );

options.put( "url", ORACLE\_CONNECTION\_URL );

options.put( "dbtable", "employee" );

//options.put( "dbtable", "IDEA\_AGEING\_BADDEBT\_201415" );

// Load MySQL query result as DataFrame

DataFrame jdbcDF = sqlContext.load( "jdbc", options );

// cache the data frame

//jdbcDF.cache();

List<Row> employeeFullNameRows = jdbcDF.collectAsList();

displayDataFrame( employeeFullNameRows );

// filter only employees with name “ABC”

/\*DataFrame filteredDf = jdbcDF.filter( jdbcDF.col( "BILLCITY" ).equalTo( "PUNE" ) );

List<Row> employeeFilterNameRows = filteredDf.collectAsList();

displayDataFrame( employeeFilterNameRows );

LOGGER.info( "Total Element with City Pune is : " + filteredDf.count() );\*/

// Group By bill city

/\*List<Row> groupByDf = jdbcDF.groupBy( "BILLCITY" ).count().collectAsList();

displayDataFrame( groupByDf );\*/

// Get the total element in the data frame

/\*long Count = jdbcDF.count();

LOGGER.info( "Total Element is : " + Count );\*/

sqlContext.registerDataFrameAsTable( jdbcDF, "tempTable" );

/\*DataFrame selectTableDF = sqlContext.sql( "Select \* from tempTable" );

List<Row> selectItem = selectTableDF.collectAsList();

displayDataFrame( selectItem );\*/

/\*DataFrame groupByDF = sqlContext.sql( "Select BILLCITY,count(\*) from tempTable group by BILLCITY" );

displayDataFrame( groupByDF.collectAsList() );

LOGGER.info( groupByDF.count() );

\*/

/\*DataFrame groupByDF1 = sqlContext.sql( "Select ISNP,BILLCITY,Max(ISNP),AVG(ISNP) from tempTable group by ISNP,BILLCITY" );

displayDataFrame( groupByDF1.collectAsList() );

LOGGER.info( groupByDF1.count() );\*/

/\*DataFrame groupByDF1 = sqlContext.sql( "Select EMPCOUNTRY,Max(EMPSALARY),EMPNAME from tempTable group by EMPCOUNTRY,EMPNAME" );

displayDataFrame( groupByDF1.collectAsList() );

\*/

//groupByDF1.javaRDD().saveAsTextFile( args[0] );

//Sort the table by BILLCITY

/\*DataFrame sortDF = sqlContext.sql( "select \* from tempTable order by BILLCITY asc" );

displayDataFrame( sortDF.collectAsList() );

\*/

//get the particular column

/\*DataFrame columnDF = jdbcDF.select( "UNBILLED\_AMOUNT" );

displayDataFrame( columnDF.collectAsList() );

//to print the data frame result into the file

columnDF.rdd().saveAsTextFile( args[0] );\*/

//Aggregate on table

Map<String, String> aggregateCondition = new HashMap<>();

aggregateCondition.put( "count", "CCCITY" );

//DataFrame aggregateDF = jdbcDF.groupBy( "BILLCITY" ).count();

//displayDataFrame( aggregateDF.collectAsList() );

//DataFrame aggregateCountDF = jdbcDF.groupBy(jdbcDF.col("CCCITY")).count();//.agg( aggregateCondition );

//DataFrame aggregateCountDF = sqlContext.sql("select ISNP ,count(ISNP),min(ISNP),max(ISNP) ,avg(ISNP),Sum(ISNP) from tempTable group by ISNP ");

//DataFrame aggregateCountDF = sqlContext.sql( "select EMPCOUNTRY ,count(EMPAGE),min(EMPAGE),max(EMPAGE) ,avg(EMPAGE),Sum(EMPAGE) from tempTable group by EMPCOUNTRY");

//displayDataFrame( aggregateCountDF.collectAsList() );

}

private static void displayDataFrame( List<Row> selectItem )

{

for ( Row DataItem : selectItem )

{

LOGGER.info( DataItem );

}

}

}