Comparison of data processing on cloud using Spark with S3 and Redshift

Dublin City University
School of Computing
CA674: Cloud Architecture
Project report
Date of submission – 16th Dec 2018
Course – MCM

Name	Student Id no	Email id
Aishwarya Gupta	18210298	aishwarya.gupta3@mail.dcu.ie
Apurva Gawad	18210295	apurva.gawad2@mail.dcu.ie
Gautam Shanbhag	18210455	gautam.shanbhag2@mail.dcu.ie

Contents

Introduction

Cloud computing is the on-demand delivery of compute power, database storage, applications, and other IT resources through a cloud services platform via the internet with pay-as-you-go pricing [5]. Cloud has opened gateways to big data processing due to its distributed framework. In this project, we will be comparing PaaS systems Spark (using data storage as S3) on Databricks, which is an open source cluster-computing framework for data processing with Redshift, which is a data ware housing system, which stores and analyses large amounts of data.

Amazon Redshift

Amazon Redshift is a data warehouse cloud service that helps companies to store and analyse large amounts of data, up to the petabyte scale. It forms a part of the large cloud-computing platform Amazon Web Services. It is built on top of technology from the massive parallel processing (MPP) data warehouse company ParAccel (later acquired by Actian), to handle large-scale data sets and database migrations. It is an enterprise-class relational database query and management system. It is based on a PostgreSQL 8.0.2, but has made many modifications in that version. It makes use of JDBC and ODBC connections to interact with other applications. It supports client connections with many types of applications, including business intelligence (BI), reporting, data, and analytics tools. There is a consensus that high performance and low costs are key advantages of using Amazon Redshift. Amazon Redshift achieves efficient storage and optimum query performance through a combination of massively parallel processing, columnar data storage, and very efficient, targeted data compression encoding schemes

Redshift Architecture:

Amazon Redshift works on the premise of collecting several nodes and creating an Amazon Redshift cluster. Clients can provision this cluster and upload data, then use complex data analysis queries to obtain business intelligence analysis

The client applications interact with the data warehouse cluster using JDBC or ODBC connections through the leader node. The leader node develops execution plans to carry out database operations. Based on the execution plan, the leader node compiles code, distributes the compiled code to the compute nodes, and assigns a portion of the data to each compute node. The leader node distributes SQL statements to the compute nodes only when a query references tables that are stored on the compute nodes. All other queries run exclusively on the leader node. The compute nodes execute the compiled code and send intermediate results back to the leader node for final aggregation. A compute node is partitioned into node slices. Each slice is allocated a portion of the node's memory and disk space, where it processes a portion of the workload assigned to the node. A cluster contains one or more databases.

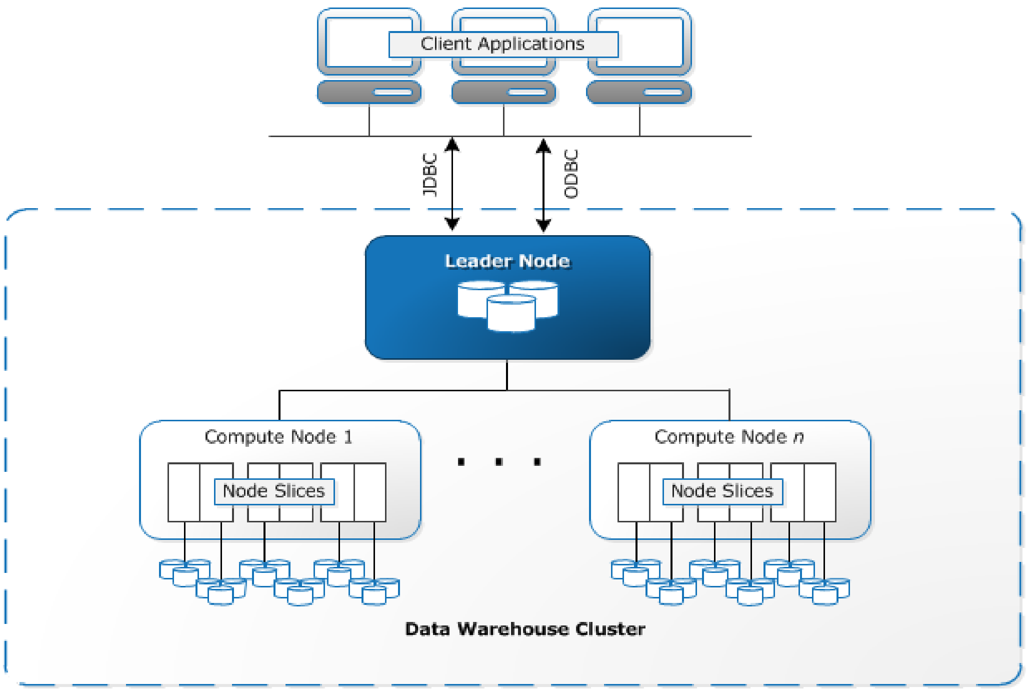
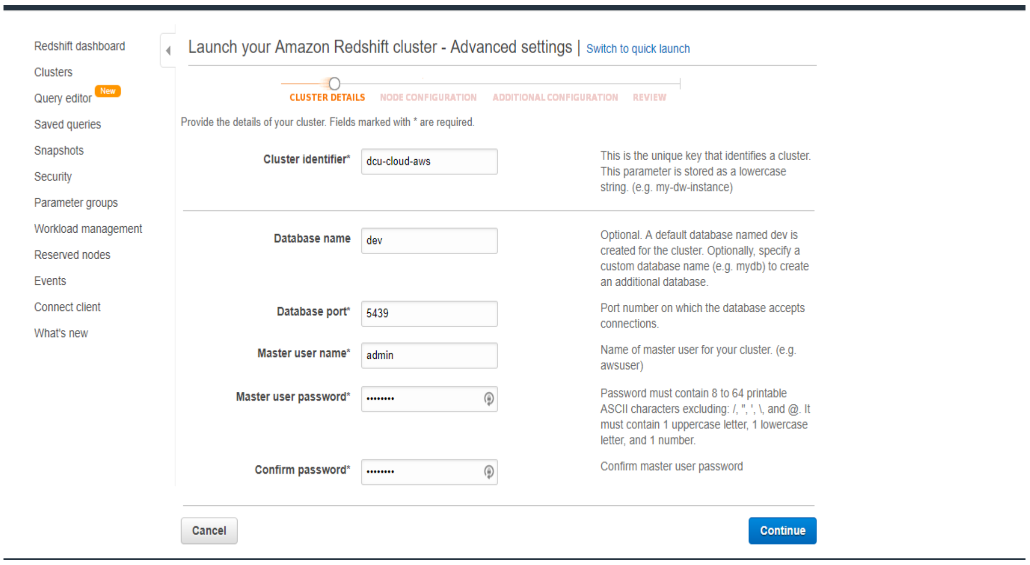


Figure 1. Redshift Architecture

Amazon Redshift is a relational database management system (RDBMS), so it is compatible with other RDBMS applications. Although it provides the same functionality as a typical RDBMS, including online transaction processing (OLTP) functions such as inserting and deleting data, Amazon Redshift is used for Analytical processing (OLAP), it is optimized for high-performance analysis and reporting of very large datasets.

Amazon Redshift Setup:

1. Before starting with Amazon Redshift Configuration, we need to setup an AWS account.
2. After Setting up AWS, you need to complete IAM configurations (Identity and Access Management) AWS Dashboard → Security, Identity & Compliance → IAM
3. Create Roles for Redshift and S3 Access Using IAM role setting. CLICK Roles → Create Role → Select Redshift → Select Case as Redshift - Customizable → Next: Permissions → Check 1. AmazonRedshiftFullAccess & 2. AmazonS3FullAccess → Next: Permissions → Tags (Optional) Next: Review
4. Newly Created roles will appear on the IAM Console
5. Create Cluster using Redshift → Click on Launch Cluster
6. Provide cluster name, provide DB name and password (by default DB name: dev)
7. Select Node as dc2.large, Cluster type: Single (Free others are paid) → Continue
8. Basic Configuration → Select Available IAM role to the one previously created with permissions → Click Continue



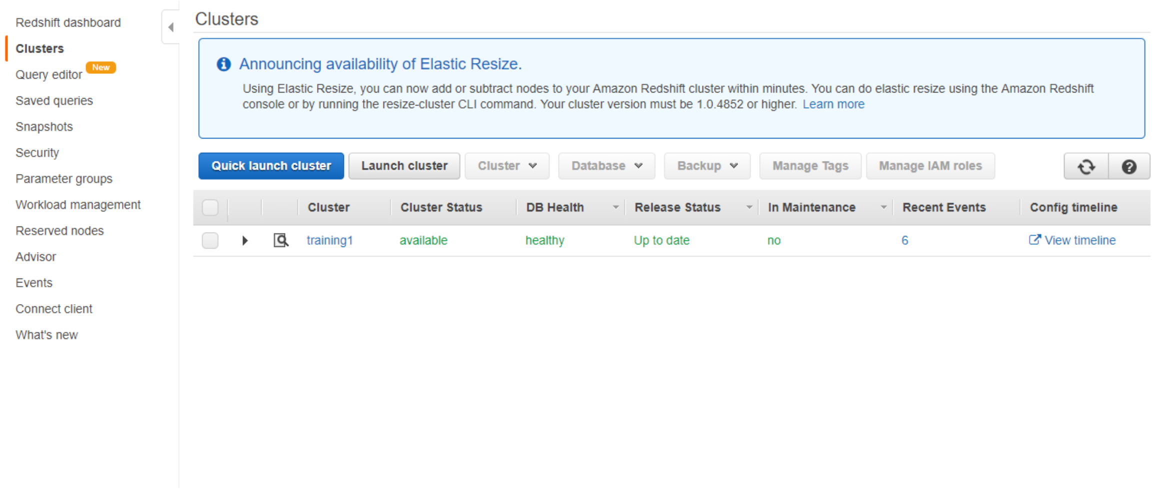
Launch your Amazon Redshift cluster - Advanced settings | Switch to quick launch

Provide the details of your cluster. Fields marked with * are required.

Cluster identifier*	dcu-cloud-aws	This is the unique key that identifies a cluster. This parameter is stored as a lowercase string. (e.g. my-dw-instance)
Database name	dev	Optional. A default database named dev is created for the cluster. Optionally, specify a custom database name (e.g. mydb) to create an additional database.
Database port*	5439	Port number on which the database accepts connections.
Master user name*	admin	Name of master user for your cluster. (e.g. awsuser)
Master user password*	••••••••	Password must contain 8 to 64 printable ASCII characters excluding: /, ", ', \, and @. It must contain 1 uppercase letter, 1 lowercase letter, and 1 number.
Confirm password*	••••••••	Confirm master user password

Figure 2. Step 2. for Redshift Setup

9. Review Configurations → Click Launch Cluster
10. The cluster would be created successfully and would appear on dashboard with status available and healthy.
11. We will connect with this Redshift cluster from our local system using DBVisualizer tool.



Clusters

Announcing availability of Elastic Resize.

Using Elastic Resize, you can now add or subtract nodes to your Amazon Redshift cluster within minutes. You can do elastic resize using the Amazon Redshift console or by running the resize-cluster CLI command. Your cluster version must be 1.0.4852 or higher. Learn more

Cluster	Cluster Status	DB Health	Release Status	In Maintenance	Recent Events	Config timeline
training1	available	healthy	Up to date	no	6	View timeline

Figure 3. Step 6. Cluster Dashboard

Spark

Spark is an open source unified analytics engine maintained by Apache software Foundation for large-scale data processing. It is a general-purpose cluster computing framework. On top of the Spark core data processing engine, there are libraries for SQL, machine learning, graph computation, and stream processing, which can be used together in an application. Application developers and data scientists incorporate Spark into their applications to rapidly query, analyse, and transform data at scale. Tasks most frequently associated with Spark include ETL and SQL batch jobs across large data sets, processing of streaming data from sensors, IoT, or financial systems, and machine learning tasks.

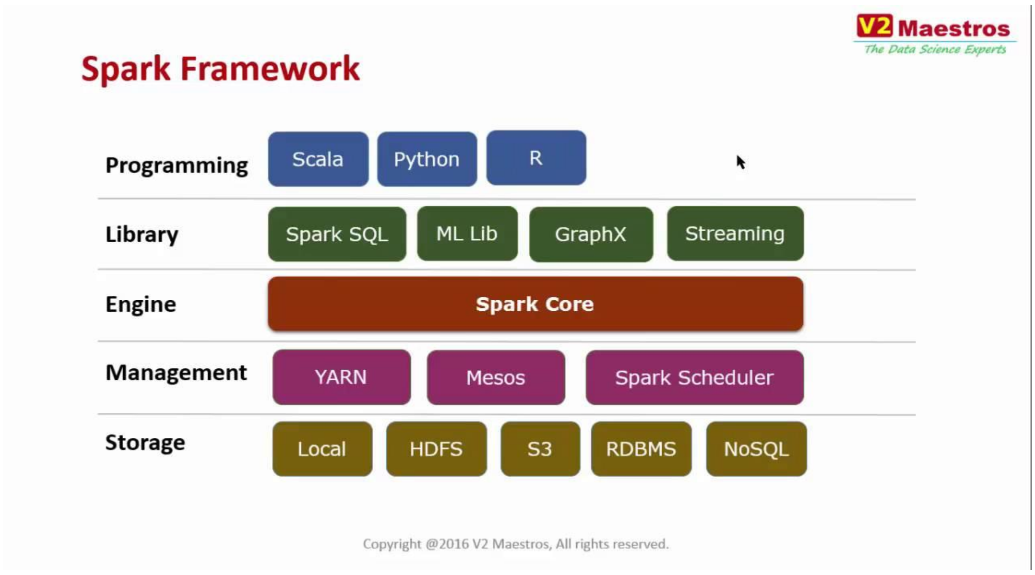


Figure 4. Spark Framework

Spark Framework and Architecture:

Spark uses a Master/Worker architecture, the driver works as the master and the executers work as the slaves.

The driver is a JVM process that holds the SparkContext for a spark application, it splits the incomings tasks and schedules them to run on executers. It coordinates between the workers and manages the overall schedule of tasks.

Executer is a distributed agent that is responsible for the execution of a task. When an executer begins, it communicates directly with the driver to execute tasks.

RDDs or Resilient distributed datasets are the back-bone spark core, it is the primary data structure that is used by spark. A RDD is a resilient and distributed collection of records spread over one or many partitions. Data frames that are used in SparkSQL are internally processed as RDDs but provide a schema to the data set.

Spark SQL is apache library for structured data processing. Unlike the basic Spark RDD API, the interfaces provided by Spark SQL provide Spark with more information about the structure of both the data and the computation being performed [4]. One use of Spark SQL is to execute SQL queries.

Spark can be used in two ways, either by running it on your local machine, where the JVM holds both the driver and executer programs, or in cluster mode – where the executers run in the cluster and you have the freedom to choose where to run the driver. Either on your JVM (generally in case of interactive notebooks) or on the cluster (in case of batch processing or streaming applications).

Spark needs a cluster manager to work as you can see in the figure 1, the cluster manager can either be an external manager like Mesos or Hadoop YARN, or we can use sparks standalone manager that comes with spark. Selection of a cluster manager for a spark application is dependent on the purpose of the application because every cluster manager provides a different set of scheduling capabilities.

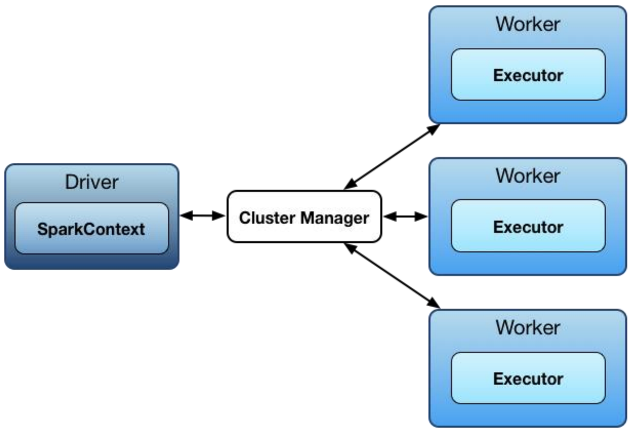


Figure 5. Spark architecture

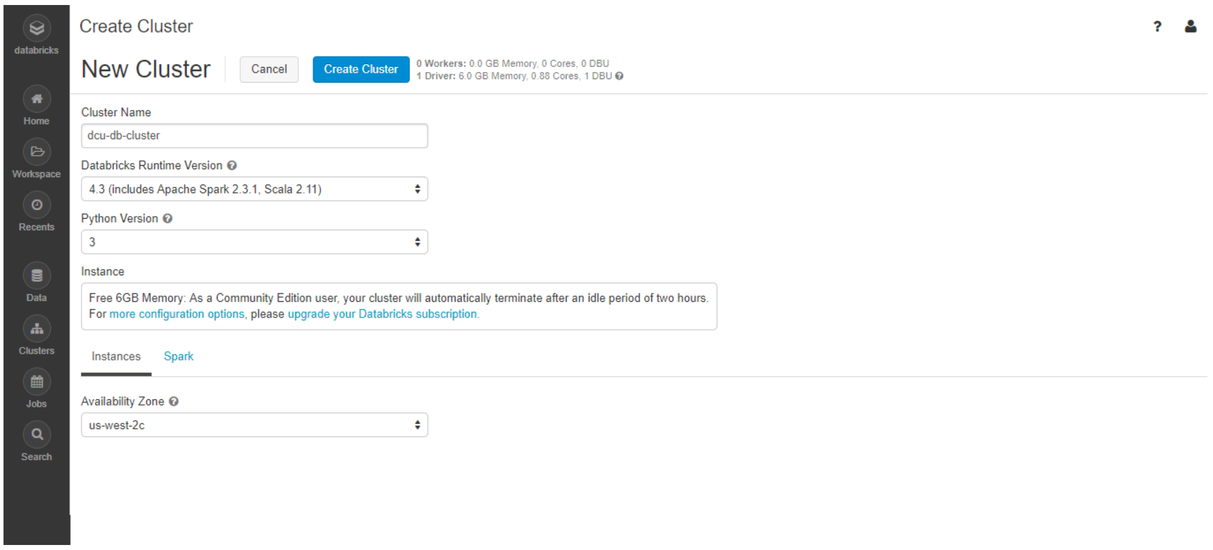
Unlike other data processing engines e.g. Apache Hadoop or Redshift, Spark does not have an internal storage system which in a way gives the user the freedom to use multiple storage systems like S3, HDFS, etc. This makes spark more flexible, however also gives additional burden of storage maintenance.

For this project, we will be using Apache Spark with S3 data storage, and executing it on cluster using Databricks community Edition.

Spark setup on Databricks:

Spark is a framework that can be used over a cluster using interactive notebooks. For our project we will be using Spark over a cluster on Databricks notebook. Databricks is a company founded by the creators of Apache Spark, that aims to help clients with cloud-based big data processing using Spark [6]. It provides a platform on which we can run spark for various analytical purposes.

1. Create a Databricks account and login to view the Dashboard
2. Click on Cluster → Create Cluster
3. Provide Cluster Name, Spark Version, Python Version & Zone → Create Cluster



Create Cluster

New Cluster

0 Workers: 0.0 GB Memory, 0 Cores, 0 DBU
1 Driver: 6.0 GB Memory, 0.88 Cores, 1 DBU

Cluster Name: dcu-db-cluster

Databricks Runtime Version: 4.3 (includes Apache Spark 2.3.1, Scala 2.11)

Python Version: 3

Instance

Free 6GB Memory: As a Community Edition user, your cluster will automatically terminate after an idle period of two hours. For more configuration options, please upgrade your Databricks subscription.

Availability Zone: us-west-2c

Figure 6. Cluster configuration

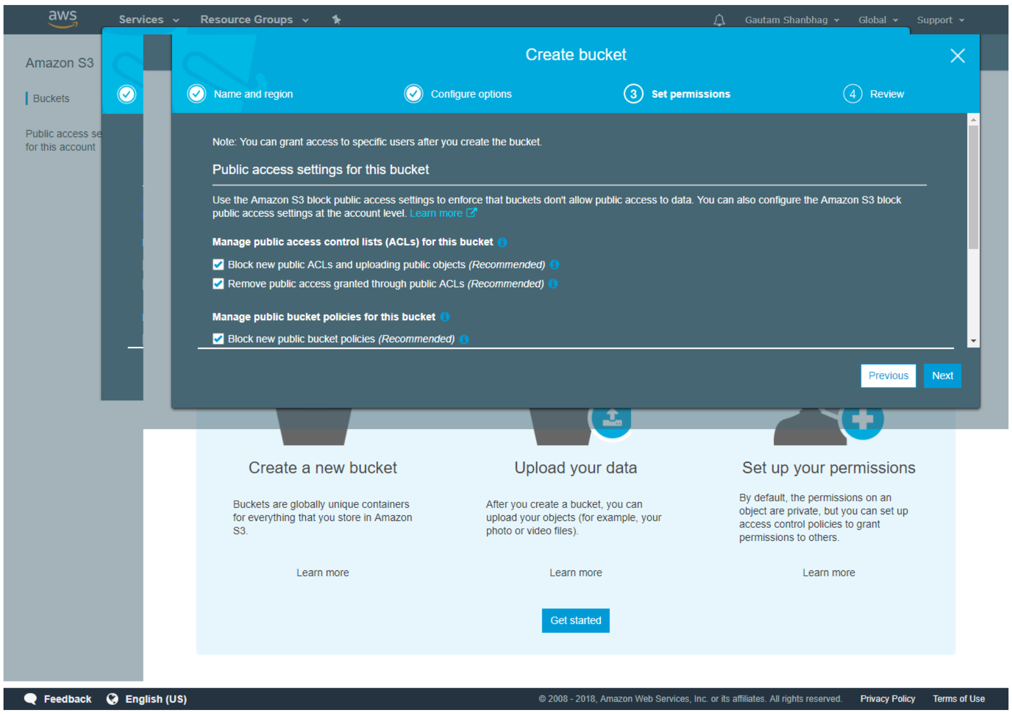
4. Created Cluster would be visible on Cluster Page and Status will change from pending to Running
5. On Dashboard → Click Create Notebook, provide Name, Programming language (we are using Python) and select cluster from the dropdown.
6. The notebook can be used for unified analytics processing using spark.

Amazon S3 - Simple Storage Service

Amazon Simple Storage Service (Amazon S3) is an object storage service that offers industry-leading scalability, data availability, security, and performance. This means customers of all sizes and industries can use it to store and protect any amount of data for a range of use cases, such as websites, mobile applications, backup and restore, archive, enterprise applications, IoT devices, and big data analytics. Amazon S3 provides easy-to-use management features so you can organize your data and configure finely-tuned access controls to meet your specific business, organizational, and compliance requirements [3].

Amazon S3 Setup:

1. Before starting with Amazon S3 Configuration, we need to setup an AWS account.
2. After Setting up AWS, you need to complete IAM configurations (Identity and Access Management) AWS Dashboard → Security, Identity & Compliance → IAM
3. Create Roles for Redshift and S3 Access Using IAM role setting. CLICK Roles → Create Role → Select Redshift → Select Case as Redshift - Customizable → Next: Permissions → Check 1. AmazonRedshiftFullAccess & 2. AmazonS3FullAccess → Next: Permissions → Tags (Optional) Next: Review



Note: You can grant access to specific users after you create the bucket.

Public access settings for this bucket

Use the Amazon S3 block public access settings to enforce that buckets don't allow public access to data. You can also configure the Amazon S3 block public access settings at the account level. Learn more

Manage public access control lists (ACLs) for this bucket

- ✓ Block new public ACLs and uploading public objects *(Recommended)*
- ✓ Remove public access granted through public ACLs *(Recommended)*

Manage public bucket policies for this bucket

- ✓ Block new public bucket policies *(Recommended)*

Buckets are globally unique containers for everything that you store in Amazon S3.

After you create a bucket, you can upload your objects (for example, your photo or video files).

By default, the permissions on an object are private, but you can set up access control policies to grant permissions to others.

© 2008 - 2018, Amazon Web Services, Inc. or its affiliates. All rights reserved.

Figure 7. Step 8. Setting permissions for S3

4. Newly Created roles will appear on the IAM Console.
5. Select S3 from Dashboard → Click on Create Bucket.
6. Provide a unique relevant bucket name and select the closest geographical region.
7. Basic configuration → Click Next.
8. Set Permissions → Click Next.
9. Click on Create bucket.
10. S3 Created bucket would appear on the S3 Dashboard

Comparative Analysis of data processing capabilities of Amazon Redshift and Spark with S3

Implementation Plan:

After completing the setup, we are ready to load our data and perform ETL and Analytical processing of data. For our analysis we will be using a large data set which we have acquired from Kaggle.com. This data set contains information about transactions made in a retail store on Black Friday Sale. The data set contains 550,000 observations.

First, we need to load our data on S3 bucket, we will load data into Redshift and Spark from our S3 bucket.

RedShift Readings:

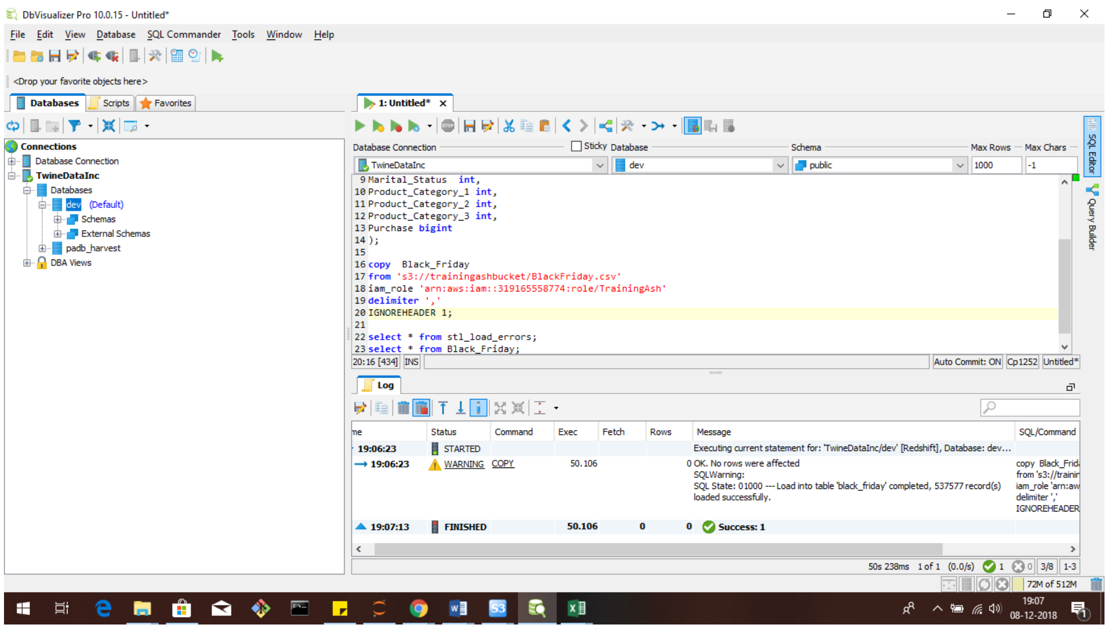
Data Loading from S3 to Redshift:

1. To load the data from S3, we need to first create a table structure. Since, we need to create a table structure it implies that we should be previously aware of the type of data we are loading and cannot be used for Dynamic loading of data.

2. After creating the table, we will use the COPY command to load the data from S3 to Redshift.
3. Query used:

```
copy  Black_Friday from
's3://trainingashbucket/BlackFriday.csv' iam_role
'arn:aws:iam::319165558774:role/TrainingAsh' delimiter
',' IGNOREHEADER 1;
```

4. Time consumed: 50.238 s



```
Marital_Status  int,
Product_Category_1 int,
Product_Category_2 int,
Product_Category_3 int,
Purchase bigint
);

copy  Black_Friday
from 's3://trainingashbucket/BlackFriday.csv'
iam_role 'arn:aws:iam::319165558774:role/TrainingAsh'
delimiter ','
IGNOREHEADER 1;

select * from stl_load_errors;
select * from Black_Friday;
```

```
me        Status    Command  Exec    Fetch  Rows  Message                                                              SQL/Command
19:06:23  STARTED                                 Executing current statement for: 'TwineDataInc/dev' [Redshift], Database: dev...
19:06:23  WARNING   COPY     50.106            0  OK. No rows were affected                                            copy  Black_Frid
                                                  SQLWarning:                                                          from 's3://trainin
                                                  SQL State: 01000 --- Load into table 'black_friday' completed, 537577 record(s)   iam_role 'arn:aw
                                                  loaded successfully.                                                 delimiter ','
                                                                                                                       IGNOREHEADER
19:07:13  FINISHED           50.106  0      0     Success: 1

50s 238ms  1 of 1  (0.0/s)  1  0  3/8  1-3
```

Figure 8. Redshift data loading from S3

DDL Commands:

1) Create table command:
   a) Query used:

```
create table Black_Friday(
User_ID bigint, Product_ID varchar,Gender   char(1),
Age    varchar,    Occupation    int,    City_Category
char(1),Stay_In_Current_City_Years              varchar,
Marital_Status      int,    Product_Category_1    int,
Product_Category_2 int,  Product_Category_3      int,
Purchase bigint );
```

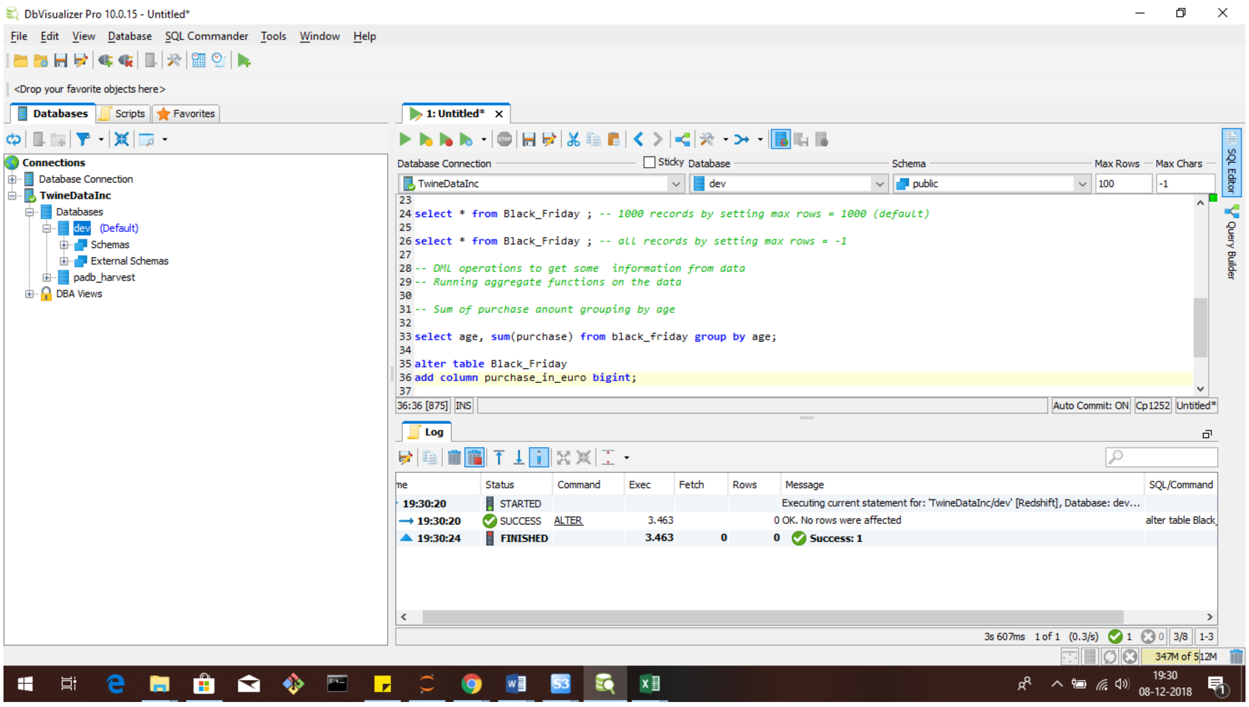
   b) First Execution time: 474 ms
   c) Average Time: 172 ms

2) Alter table command: We created another column purchase_in_euro which stored the purchase amount in Euros.
   a) Query used:

```
alter table Black_Friday
add column purchase_in_euro bigint;
```

b) Execution time: 3s 607ms

c) Average Time: 172 ms



```
select * from Black_Friday ; -- 1000 records by setting max rows = 1000 (default)

select * from Black_Friday ; -- all records by setting max rows = -1

-- DML operations to get some  information from data
-- Running aggregate functions on the data

-- Sum of purchase anount grouping by age

select age, sum(purchase) from black_friday group by age;

alter table Black_Friday
add column purchase_in_euro bigint;
```

me	Status	Command	Exec	Fetch	Rows	Message	SQL/Command
19:30:20	STARTED					Executing current statement for: 'TwineDataInc/dev' [Redshift], Database: dev...	
19:30:20	SUCCESS	ALTER	3.463		0	OK. No rows were affected	alter table Black_
19:30:24	FINISHED		3.463	0	0	Success: 1	

```
3s 607ms   1 of 1   (0.3/s)   1   0   3/8   1-3
```

Figure 9. Redshift Alter command

3) Delete command: We use delete command with various conditions to test how much of its performance is affected by using multiple conditions. Since, Redshift is a columnar database, such conditions increase the computation time.

a) Query used:

```
delete from black_friday where
(black_friday.product_category_1  is null or
black_friday.product_category_2 is null or
black_friday.product_category_3  is null) and
black_friday.city_category != 'A' ;
```

b) Exec time: 2s 183ms

c) Average time: 357 ms

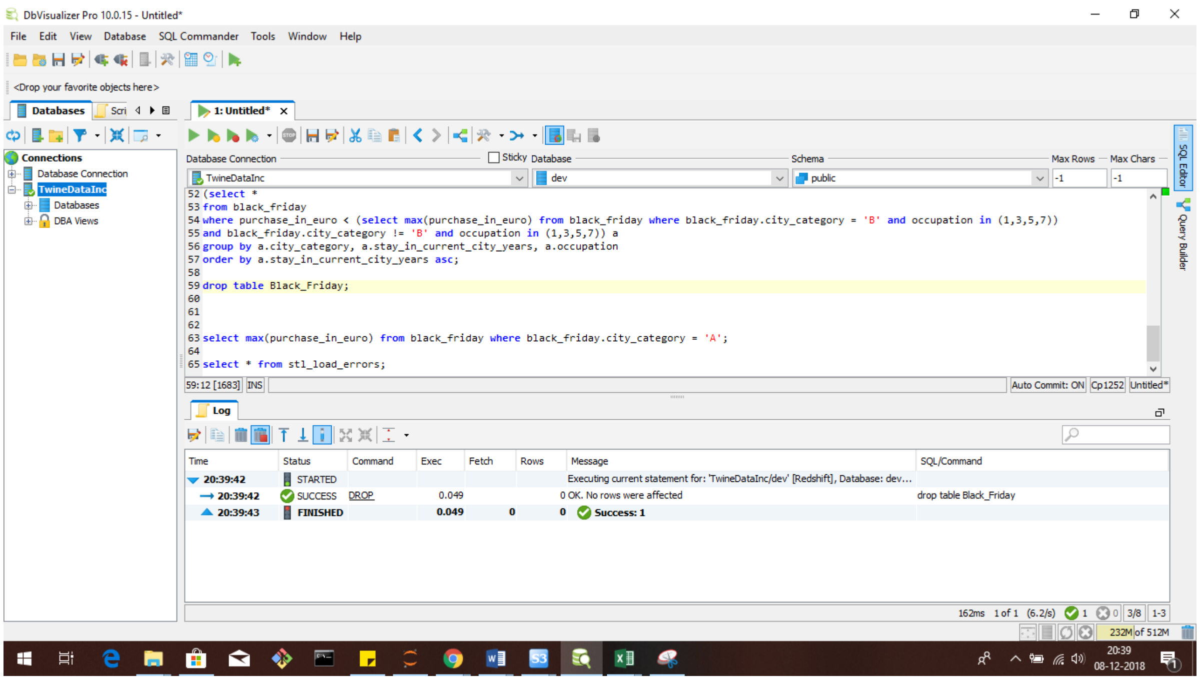
4) Drop command: Drop command in Redshift frees up the memory when executed. This command is very fast owing to its multiple parallel processing and columnar data storage structure.

a) Query used:

```
drop table Black_Friday;
```

b) Exec time: 162 ms

c) Average time: 153 ms



```
(select *
from black_friday
where purchase_in_euro < (select max(purchase_in_euro) from black_friday where black_friday.city_category = 'B' and occupation in (1,3,5,7))
and black_friday.city_category != 'B' and occupation in (1,3,5,7)) a
group by a.city_category, a.stay_in_current_city_years, a.occupation
order by a.stay_in_current_city_years asc;

drop table Black_Friday;



select max(purchase_in_euro) from black_friday where black_friday.city_category = 'A';

select * from stl_load_errors;
```

```
Time        Status    Command  Exec   Fetch  Rows  Message                                                                  SQL/Command
20:39:42    STARTED                                Executing current statement for: 'TwineDataInc/dev' [Redshift], Database: dev...
 20:39:42   SUCCESS   DROP     0.049             0 OK. No rows were affected                                                drop table Black_Friday
 20:39:43   FINISHED           0.049      0      0 Success: 1
```

Figure 10. Redshift Drop Command

DML commands:

1) Selecting 1000 records from table : We especially ran two queries to see if there is any major difference in the time consumed to fetch fraction of data and complete data from Redshift table.
   a. Query used:

      ```
      select * from Black_Friday limit 1000;
      ```

   b. Execution time: 3m 708s
   c. Average execution time: 480 ms

2) Fetching entire data set of 537577 records.
   a. Query Used:

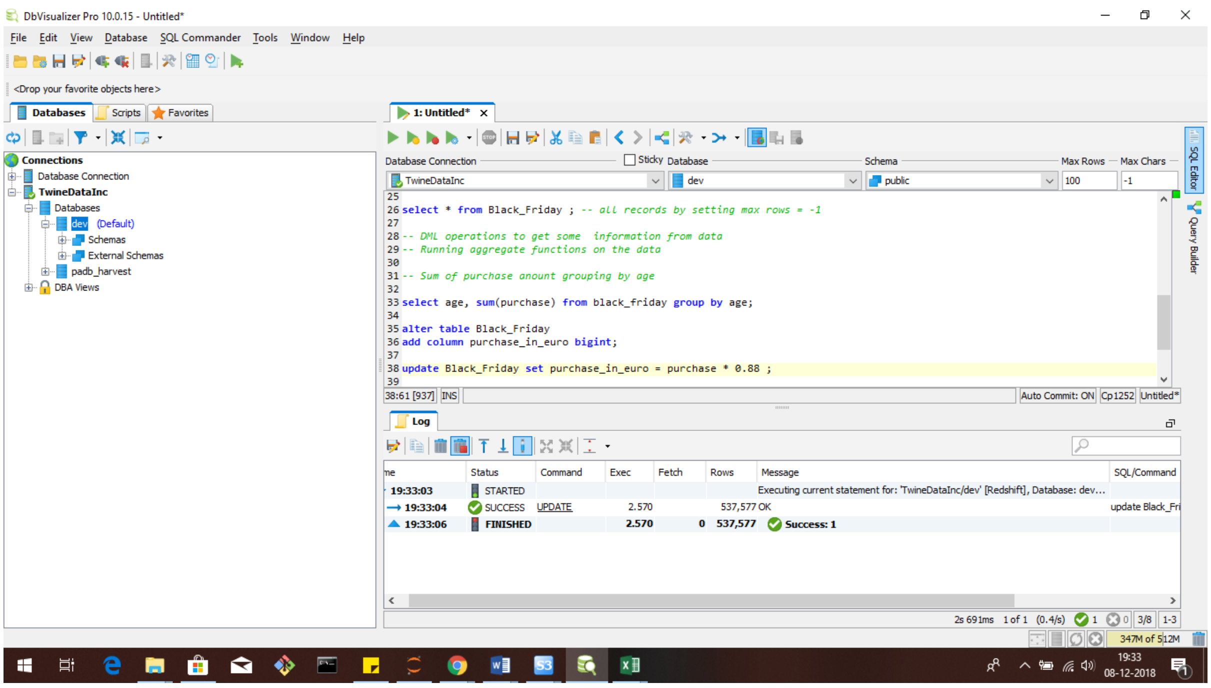
      ```
      select * from Black_Friday
      ```

   b. Execution time: 21 s 257ms
   c. Average execution time: 3s 454 ms

3) Update Query: We wanted to insert data in the column that we had created using the alter command. Since this is a column level function, we expected this command to perform well.
   a. Query used:

      ```
      update Black_Friday set purchase_in_euro = purchase *
      0.88;
      ```

   b. Exec time: 2s 691ms
   c. Average exec time : 1s 290ms



```
select * from Black_Friday ; -- all records by setting max rows = -1

-- DML operations to get some  information from data
-- Running aggregate functions on the data

-- Sum of purchase anount grouping by age

select age, sum(purchase) from black_friday group by age;

alter table Black_Friday
add column purchase_in_euro bigint;

update Black_Friday set purchase_in_euro = purchase * 0.88 ;
```

```
me        Status    Command  Exec   Fetch  Rows     Message                                                                   SQL/Command
19:33:03  STARTED                                   Executing current statement for: 'TwineDataInc/dev' [Redshift], Database: dev...
19:33:04  SUCCESS   UPDATE   2.570         537,577  OK                                                                        update Black_Fri
19:33:06  FINISHED           2.570  0      537,577  Success: 1
```

Figure 11. Redshift Update command

4) Aggregate functions: Redshift being a columnar database is fast in terms of aggregate functions, hence this was an important comparative factor for us with spark.

   a. Query used:

      `select gender, age, avg(purchase_in_euro) as average_spent from black_friday where black_friday.marital_status = 1 group by gender, age having avg(purchase_in_euro) > 1000 order by age asc, average_spent asc;`

   b. Exec time: 8s 193 ms

   c. Average time: 393 ms

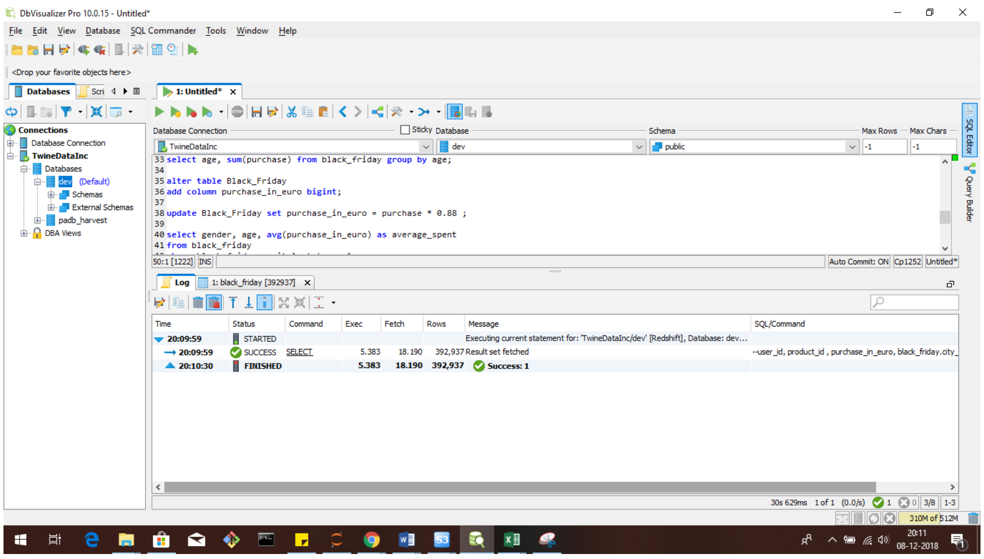
5) Nested Query: We wanted to see how Redshift performs when nested queries are applied, we have used three levels nest to see the performance of Redshift.

   a. Query used:

      `select a.city_category, a.stay_in_current_city_years, a.occupation, max(a.purchase_in_euro), a.gender from (select * from black_friday where purchase_in_euro < (select max(purchase_in_euro) from black_friday where black_friday.city_category = 'C' and occupation in (2,4,5,7)) and black_friday.city_category != 'C' and occupation in (2,4,5,7)) a group by a.city_category, a.stay_in_current_city_years, a.occupation, a.gender having count(a.user_id) > 2 order by a.stay_in_current_city_years asc;`

   b. Exec time: 30s 629 ms

   c. Average execution time: 406 ms



```
select age, sum(purchase) from black_friday group by age;

alter table Black_Friday
add column purchase_in_euro bigint;

update Black_Friday set purchase_in_euro = purchase * 0.88 ;

select gender, age, avg(purchase_in_euro) as average_spent
from black_friday
```

```
Time       Status    Command  Exec   Fetch   Rows     Message                                                                      SQL/Command
20:09:59   STARTED                                    Executing current statement for: 'TwineDataInc/dev' [Redshift], Database: dev...
20:09:59   SUCCESS   SELECT   5.383  18.190  392,937  Result set fetched                                                           --user_id, product_id , purchase_in_euro, black_friday.city_
20:10:30   FINISHED           5.383  18.190  392,937  Success: 1
```

Figure 12. Redshift Nested query command

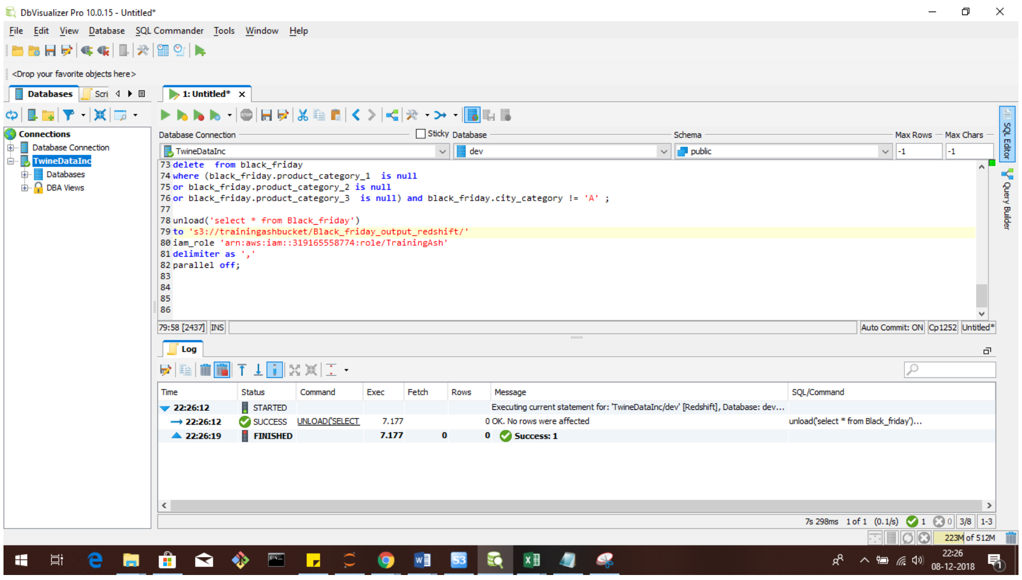
Unloading data from Redshift to S3:

One of the benefits of using Redshifts is that it is a data ware house hence, we can store are processed data in Redshift during the Loading of the ETL process. However, when using Spark, since it does not have its own storage we need to unload the data from spark and export or load it to another data storage system which is S3 for our case. To compare the performance of data unloading between spark and S3 we have unloaded data from Redshift to S3.

1) Query Used:

```
unload('select * from Black_friday') to 's3://trainingashbucket/Black_friday_output_redshift/'iam_role 'arn:aws:iam::319165558774:role/TrainingAsh' delimiter as ',' parallel off;
```

2) Exec time : 7s 298 ms
3) Average time : 1s 578ms



```
delete  from black_friday
where (black_friday.product_category_1  is null
or black_friday.product_category_2 is null
or black_friday.product_category_3  is null) and black_friday.city_category != 'A' ;

unload('select * from Black_friday')
to 's3://trainingashbucket/Black_friday_output_redshift/'
iam_role 'arn:aws:iam::319165558774:role/TrainingAsh'
delimiter as ','
parallel off;
```

```
Time       Status    Command          Exec   Fetch  Rows  Message                                                                      SQL/Command
22:26:12   STARTED                                          Executing current statement for: 'TwineDataInc/dev' [Redshift], Database: dev...
22:26:12   SUCCESS   UNLOAD('SELECT   7.177            0  OK. No rows were affected                                                    unload('select * from Black_friday')...
22:26:19   FINISHED                   7.177  0         0  Success: 1
```

```
7s 298ms   1 of 1   (0.1/s)
```

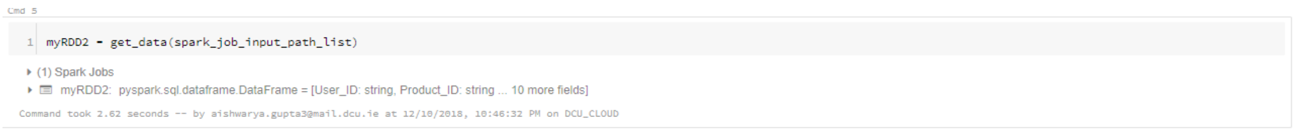
Figure 13. Redshift Unload command

Spark Readings:

Data Loading from S3 to Spark:

1) Data can be loaded into spark without creating any previous table. The dataframe gets created dynamically at run time when the data is being loaded.
2) Exec Time: 2.62s
3) Command used:

```
spark.read.csv(spark_job_input_path, header = "TRUE")
```



```
Cmd 5
myRDD2 = get_data(spark_job_input_path_list)

▸ (1) Spark Jobs
▸ myRDD2:  pyspark.sql.dataframe.DataFrame = [User_ID: string, Product_ID: string ... 10 more fields]
Command took 2.62 seconds -- by aishwarya.gupta3@mail.dcu.ie at 12/10/2018, 10:46:32 PM on DCU_CLOUD
```

Figure 14. Spark data loading from S3

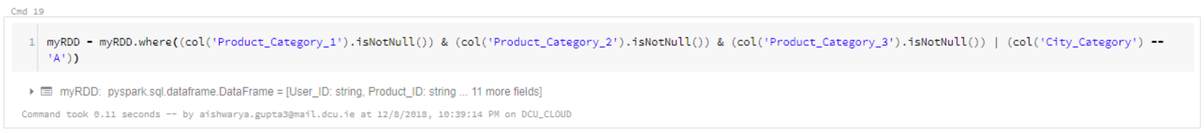
DDL Commands:

1) Alter Table command – As we are working with data frames, there is no direct DDL command to alter the data frame, however we can use spark commands to alter the data frames.
    a. Exec time – 0.02 s
    b. Command used:

```
myRDD = myRDD.withColumn('purchase_in_euro',lit(0))
```

2) Delete command – Spark does not have a direct mechanism to delete rows based on condition, however we can select the rows that we want and create a new data frame replacing the old one.
    a. Exec time – 0.11s
    b. Command used:

```
myRDD =
myRDD.where((col('Product_Category_1').isNotNull()) &
(col('Product_Category_2').isNotNull()) &
(col('Product_Category_3').isNotNull()) |
(col('City_Category') == 'A'))
```



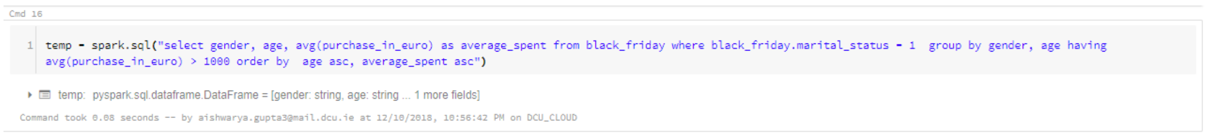
```
Cmd 19
myRDD = myRDD.where((col('Product_Category_1').isNotNull()) & (col('Product_Category_2').isNotNull()) & (col('Product_Category_3').isNotNull()) | (col('City_Category') ==
'A'))

▸ myRDD:  pyspark.sql.dataframe.DataFrame = [User_ID: string, Product_ID: string ... 11 more fields]
Command took 0.11 seconds -- by aishwarya.gupta3@mail.dcu.ie at 12/8/2018, 10:39:14 PM on DCU_CLOUD
```

Figure 15. Spark Delete command

3) Drop command – In Spark there is no way to directly drop the table, however the reference could be changed and cache cleared to get the same result as that of drop.

DML Commands:

1) Fetching 1000 rows without any condition
    a. Exec time – 0.03 s
    b. Command used:
    ```
    temp = spark.sql("select * from Black_Friday limit
    1000")
    ```
2) Fetching 1000 rows without any condition – We can see that there is no major difference in the time consumed by spark to fetch the entire data and part of the data compared to Redshift.
    a. Exec time – 0.03 s
    b. Command used:
    ```
    temp = spark.sql("select * from Black_Friday")
    ```
3) Update Query: Since we are working on data frame, we cannot update the columns directly using sql commands, however we can manipulate the data frame and load values.
    a. Exec time – 0.03 s
    b. Command used:
    ```
    myRDD = myRDD.withColumn('purchase_in_euro',
    col('purchase')*0.88)
    ```
4) Aggregate Query: Even though, Redshift is a columnar data base, we can see here that spark is clearly faster when it comes to aggregate queries. This is due to the portioning of work mechanism that spark uses.
    a. Exec time – 0.08 s
    b. Command used:
    ```
    temp = spark.sql("select gender, age,
    avg(purchase_in_euro) as average_spent from black_friday
    where black_friday.marital_status = 1  group by gender,
    age having   avg(purchase_in_euro) > 1000 order by  age
    asc, average_spent asc")
    ```



```
Cmd 16

temp = spark.sql("select gender, age, avg(purchase_in_euro) as average_spent from black_friday where black_friday.marital_status = 1  group by gender, age having
avg(purchase_in_euro) > 1000 order by  age asc, average_spent asc")

temp:  pyspark.sql.dataframe.DataFrame = [gender: string, age: string ... 1 more fields]
Command took 0.08 seconds -- by aishwarya.gupta3@mail.dcu.ie at 12/10/2018, 10:56:42 PM on DCU_CLOUD
```

Figure 16. Spark Aggregate Query

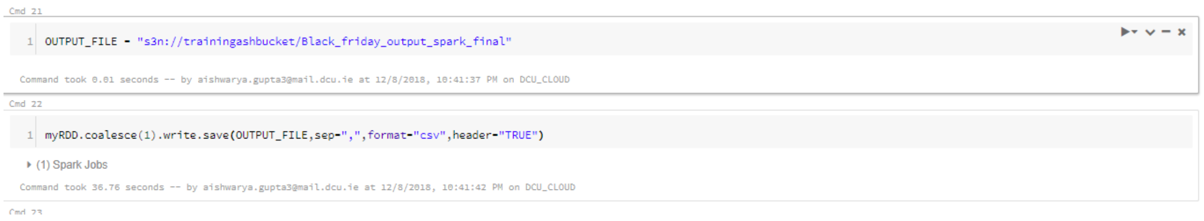
5) Nested Query : We can see that spark is extremely fast in nested queries.
    a. Exec time – 0.10s
    b. Command used:
    ```
    temp = spark.sql("select a.city_category,
    a.stay_in_current_city_years, a.occupation,
    max(a.purchase_in_euro), a.gender  from  (select * from
    black_friday  where purchase_in_euro < (select
    max(purchase_in_euro) from black_friday where
    black_friday.city_category = 'C' and occupation in
    (2,4,5,7)) and black_friday.city_category != 'C' and
    ```

```
occupation in (2,4,5,7)) a group by a.city_category,
a.stay_in_current_city_years, a.occupation, a.gender
having count(a.user_id) > 2 order by
a.stay_in_current_city_years asc ")
```

Data unloading to S3 from Spark:
As Spark does not have its own data base or storage, we have to export our results to an external storage or else all work will be lost.

1) Exec time - 36.76 s
2) Command used:

```
myRDD.coalesce(1).write.save(OUTPUT_FILE,sep=",",format
="csv",header="TRUE")
```



```
Cmd 21
OUTPUT_FILE = "s3n://trainingashbucket/Black_friday_output_spark_final"
Command took 0.01 seconds -- by aishwarya.gupta3@mail.dcu.ie at 12/8/2018, 10:41:37 PM on DCU_CLOUD
Cmd 22
myRDD.coalesce(1).write.save(OUTPUT_FILE,sep=",",format="csv",header="TRUE")
▸ (1) Spark Jobs
Command took 36.76 seconds -- by aishwarya.gupta3@mail.dcu.ie at 12/8/2018, 10:41:42 PM on DCU_CLOUD
Cmd 23
```

Figure 17. Spark unload command

We can see that spark takes significant amount of time to export data; this is due to the fact that the RDDs are partitioned and collaborating the RDDS and sending the information takes time.

This phenomenon can also be observed when we try to show() the contents of the data frame, as again this requires the merging of RDDs which becomes time consuming for spark.

Summary of the review above for some parameters:-

The parameters for Redshift, which are in bracket, tells the first execution time of the query, it can be observed that repeated execution of same query tremendously reduces the execution time. This is due to the feature of result caching in Redshift. Despite result caching, we can observe from the readings that spark is significantly faster than Redshift when it comes to processing large amounts of data.

Comparison factor	**Amazon Redshift**	**Apache Spark**
Data Load time from S3	6s 663ms (50s 238 ms *)	2s 620 ms
Fetching entire data set	12 s 557ms (21 s 257ms *)	20 ms
Aggregate function execution	393 ms (8s 193 ms *)	80 ms
Nested query execution	406 ms (30s 629 ms *)	100 ms
Deletion of values based on condition	357 ms (2s 183 ms *)	110 ms
Drop table	153 ms (162 ms *)	10 ms
Alter command	185 ms (3s 607ms *)	20 ms
Update command	1s 290 ms (2s 691ms *)	30 ms
Data unload to S3	1s 578ms (7s 298 ms *)	36s 760 ms

Conceptual Comparison:-

In the below table we can see the comparison of various conceptual factors between Amazon Redshift and Apache spark.

Comparison factor	**Amazon Redshift**	**Apache Spark**
Category	Analytical database	Data processing engine
Supported languages	SQL-Like language	Extensible via pre built libraries in Java, Python, R, Scala
Interoperability	Compatible with standard JDBC, ODBC drivers Integrated with most BI, ETL tools	Runs Everywhere. Can access diverse data sources. Runs on Hadoop, Mesos, Kubernetes, standalone or in the cloud.
Distribution model	Commercial/managed	Opensource
Storage Support	Inbuilt Storage	External Storage Engine Required
Live app Database	Not supported	Supports streaming data
Concurrency	Efficient multi user support	Performance degrades as concurrency increases
Typical application	Data warehouse	applications

Conclusion:

Even though spark and Redshift are comparable when it comes to data processing, it all boils down to the requirement of the user. Redshift being a data ware house is preferable when the goal is to archive the data and requires periodical processing of large amount of data, having easy scalability it is suited for applications where the data is ever growing and of different types. However, spark comes handy when we need to process large amounts of data continuously; it is capable of handling streaming data and building machine learning algorithms.

References:

[1] Online reference - https://spark.apache.org/

[2] Image reference - https://www.youtube.com/watch?v=ZTFGwQaXJm8

[3] https://aws.amazon.com/s3/

[4] https://spark.apache.org/docs/latest/sql-programming-guide.html

[5] https://aws.amazon.com/what-is-cloud-computing/

[6] https://databricks.com/product/unified-analytics-platform

[7] https://dbengines.com/en/system/Amazon+Redshift%3BSpark+SQL

[8] https://docs.aws.amazon.com/redshift/latest/mgmt/welcome.html

[9] https://docs.aws.amazon.com/redshift/latest/mgmt/managing-clusters-console.html

[10] https://docs.aws.amazon.com/redshift/latest/mgmt/generating-user-credentials.html