UNIVERSITEIT VAN AMSTERDAM

Big Data - Lab Assignment 5

Group 19

Jasper Adema (UvA ID: 11879394) & Daphnee Chabal (UvA ID: 11200898)

UNIVERSITEIT VAN AMSTERDAM

Question 1

A The query runtime results are pictured in table 1 and plotted in figure 1 and are quite dissimilar. Each query was run three times for both Redshift and Athena. Redshift is slower on the first run of the query, but much faster on the second and third run. Athena is significant faster than Redshift on the first query run as displayed in table 2. The query performance for Athena is stable and does not decrease if it is run more often.

B. In figure 2 the differences for the first run times per query are compared. There is some odd thing going on with SF 10 at the first runtime with Redshift, which runtime is shorter than that of SF 1. But besides this anomaly it seems that in general Redshift first runtimes increase exponentially with dataset size while Athena scales linearly.

However, the first run results for Redshift are misleading and therefore its performance must be assessed based on the second and third run. When analyzing these runtimes another picture appears as shown in figure 3. The larger the dataset the better the performance of Redshift. It seems that Redshift is more geared towards larger datasets. Yet, it must also be taken into consideration that the cluster was scaled up for the SF 100 dataset which may offset the reported results a bit.

C. Athena seems to return similar runtimes for each time exact the same query is executed and faster at the first query run (table 2). Redshift's performance improves enormously when the query is repeated. The table 3 shows this phenomenon.

Athena is serverless and runs the query directly on the data stored in S3 [1]. Redshift on the other hand requires a cluster to be set up and also the data and tables need to be uploaded to the cluster [2]. However, this difference should enable Redshift to query faster on the first run so why is there still a difference in performance?

While Athena is good for ad-hoc queries, Redshift is optimized for large scale data analytics (to run queries on petabytes of data). According to Amazon's Redshift documentation [3] one of the elements to achieve this optimization is to generate and compile code for each query execution plan. After the code is compiled it is stored in the least recently used (LRU) cache and distributed among the cluster. If a query is re-run the code generation and compilation steps can be skipped causing the query to execute much faster. In other words when executing a query in Redshift for the first time there is

some overhead cost. This can be misleading, the second and subsequent runs are therefore a better indicator of normal performance.

		Run	Redshift	Athena
SF 1	**Query 1**	1	10.16s	5.28s
		2	0.25s	5.11s
		3	0.23s	5.04s
	Query 5	1	26.97s	4.63s
		2	0.27s	4.46s
		3	0.25s	4.5s
SF 10	**Query 1**	1	24.1s	13.41s
		2	0.27s	14.53s
		3	0.23s	14.07s
	Query 5	1	14.26s	10.44s
		2	0.27s	11.82s
		3	0.26s	11.55s
SF 100	**Query 1**	1	52.78s	18.77s
		2	0.28s	18.85s
		3	0.26s	19.16s
	Query 5	1	50.12s	17.83s
		2	0.26s	16.97s
		3	0.23s	17.66s

Table 1: Query runtimes Redshift and Athena

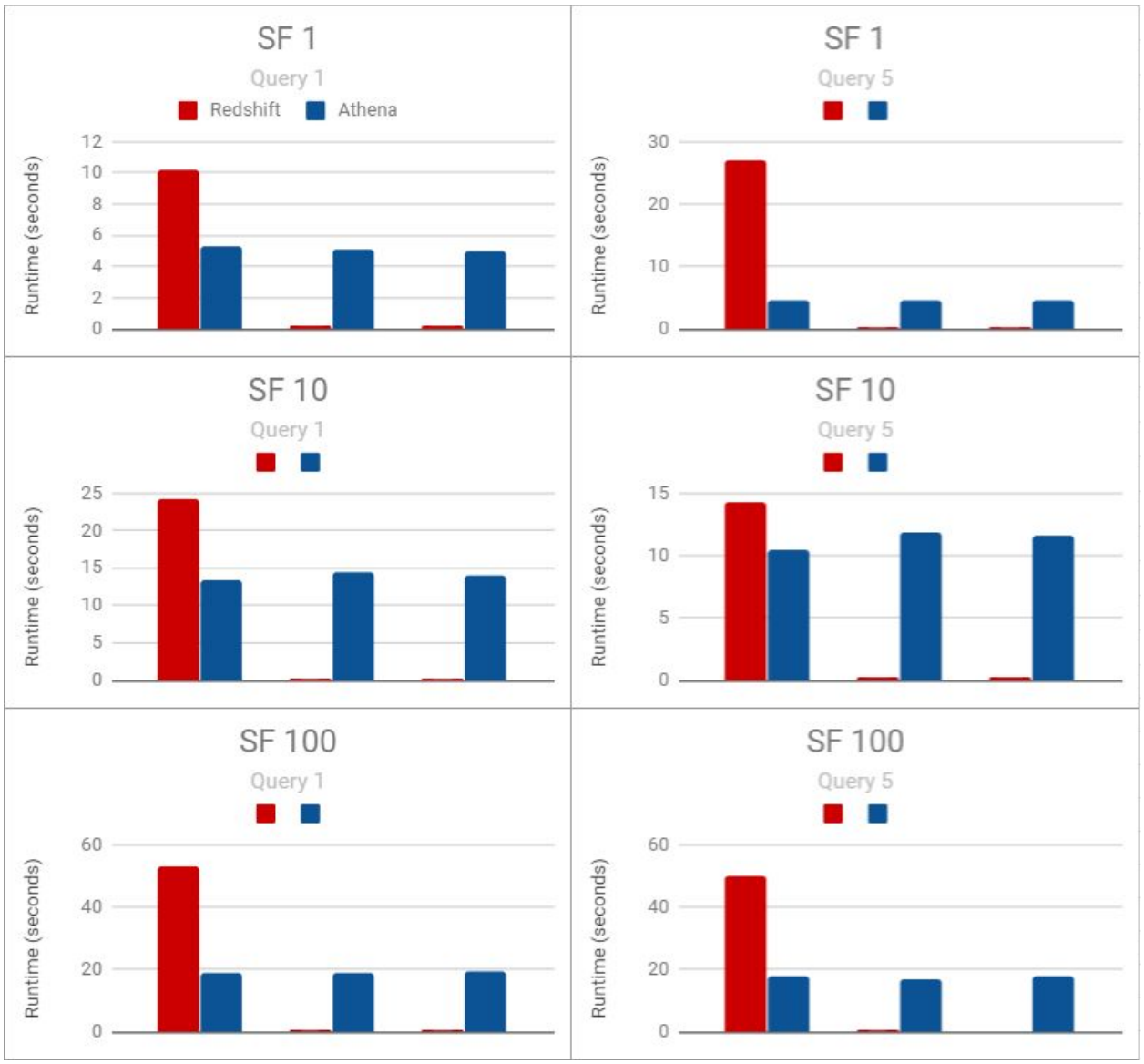

Figure 1: Query runtimes Redshift (red) versus Athena (blue)

	Query 1	**Query 5**
SF 1	92%	483%
SF 10	80%	37%
SF 100	181%	181%

Table 2: Percentage that Athena is faster than Redshift on first query run

	Query 1	Query 5
SF 1	1944%	1552%
SF 10	5281%	4278%
SF 100	6632%	6427%

Table 3: Percentage that Redshift is faster than Athena on <u>second</u> query run

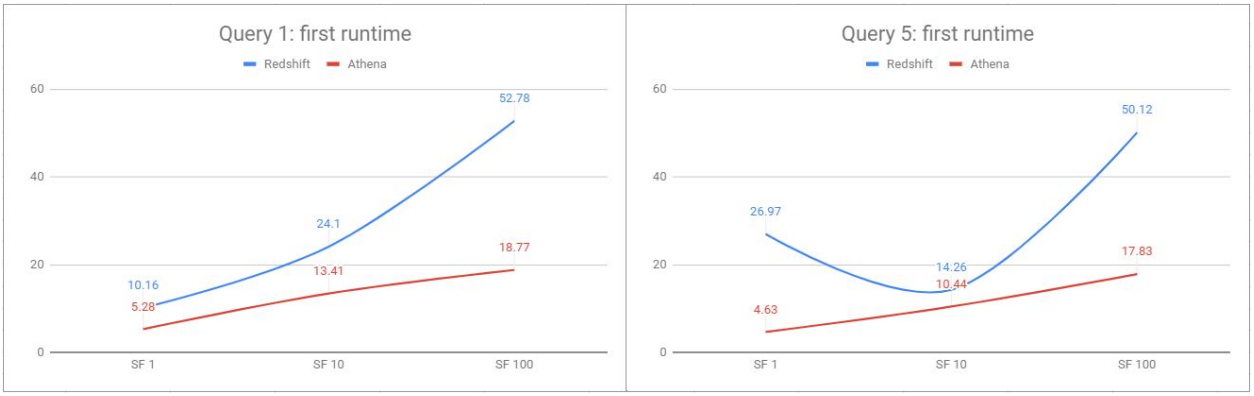

Figure 2: <u>first</u> runtimes versus dataset size

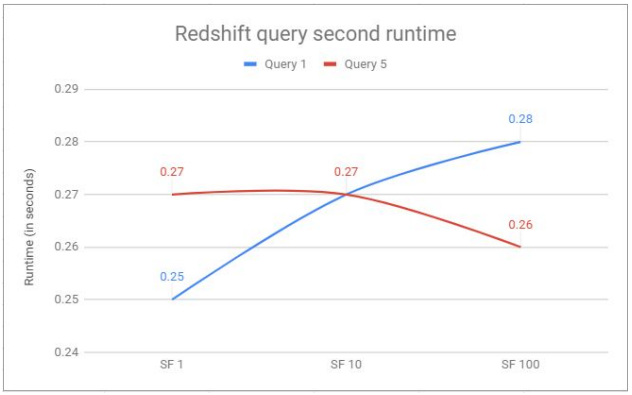

Figure 3: Redshift <u>second</u> runtime versus dataset size

Question 2

A In figure 4 the differences in runtimes for querying over CSV-files for Spark, Athena, and Redshift are displayed. The results show that Spark is considerably slower in executing the queries. The difference in executing time is especially clear for query 5. It appears that Spark lacks optimality for querying multiple tables compared with Redshift. The comparison with Athena is not entirely fair because regarding processing power Spark is restricted to a certain number of nodes in its cluster, while Athena does not have this constraint.

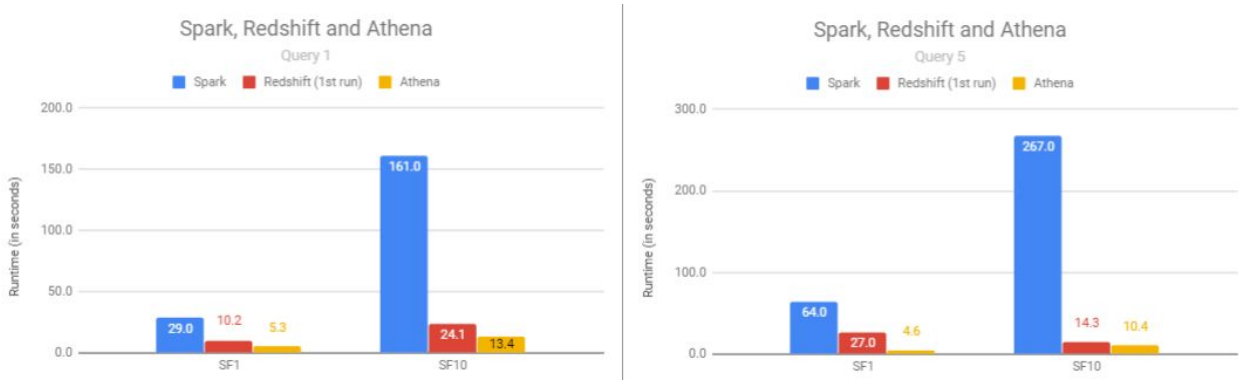

Figure 4: Runtime results for queries using CSV-files

B. To be able to convert the CSV-files to Parquet-files the structure of the tables needs to be specified. This can be realized by providing a scheme when reading the CSV-files. This scheme will be utilized to transform the data to columnar storage in Parquet-files. The code for converting the CSV-files to Parquet-files is below.

```
%pyspark
#Set schema for the CSV files
from pyspark.sql.types import *

region_s = StructType([
    StructField('r_regionkey', IntegerType(), True),
    StructField('r_name', StringType(), True),
    StructField('r_comment', StringType(), True)
    ])

nation_s = StructType([
    StructField('n_nationkey', IntegerType(), True),
    StructField('n_name', StringType(), True),
    StructField('n_regionkey', IntegerType(), True),
    StructField('n_comment', StringType(), True)
    ])

supplier_s = StructType([
    StructField('s_suppkey', IntegerType(), True),
    StructField('s_name', StringType(), True),
    StructField('s_address', StringType(), True),
    StructField('s_nationkey', IntegerType(), True),
    StructField('s_phone', StringType(), True),
    StructField('s_acctbal', FloatType(), True),
    StructField('s_comment', StringType(), True)
    ])

customer_s = StructType([
    StructField('c_custkey', IntegerType(), True),
    StructField('c_name', StringType(), True),
    StructField('c_address', StringType(), True),
    StructField('c_nationkey', IntegerType(), True),
    StructField('c_phone', StringType(), True),
    StructField('c_acctbal', FloatType(), True),
    StructField('c_mktsegment', StringType(), True),
    StructField('c_comment', StringType(), True)
    ])

part_s = StructType([
    StructField('p_partkey', IntegerType(), True),
```

```
    StructField('p_name', StringType(), True),
    StructField('p_mfgr', StringType(), True),
    StructField('p_brand', StringType(), True),
    StructField('p_type', StringType(), True),
    StructField('p_size', IntegerType(), True),
    StructField('p_container', StringType(), True),
    StructField('p_retailprice', FloatType(), True),
    StructField('p_comment', StringType(), True)
    ])


partsupp_s = StructType([
    StructField('ps_partkey', IntegerType(), True),
    StructField('ps_suppkey', IntegerType(), True),
    StructField('ps_availqty', IntegerType(), True),
    StructField('ps_supplycost', FloatType(), True),
    StructField('ps_comment', StringType(), True)
    ])

orders_s = StructType([
    StructField('o_orderkey', IntegerType(), True),
    StructField('o_custkey', IntegerType(), True),
    StructField('o_orderstatus', StringType(), True),
    StructField('o_totalprice', FloatType(), True),
    StructField('o_orderdate', DateType(), True),
    StructField('o_orderpriority', StringType(), True),
    StructField('o_clerk', StringType(), True),
    StructField('o_shippriority', IntegerType(), True),
    StructField('o_comment', StringType(), True)
    ])

lineitem_s = StructType([
    StructField('l_orderkey', IntegerType(), True),
    StructField('l_partkey', IntegerType(), True),
    StructField('l_suppkey', IntegerType(), True),
    StructField('l_linenumber', IntegerType(), True),
    StructField('l_quantity', IntegerType(), True),
    StructField('l_extendedprice', FloatType(), True),
    StructField('l_discount', FloatType(), True),
    StructField('l_tax', FloatType(), True),
```

```
    StructField('l_returnflag', StringType(), True),
    StructField('l_linestatus', StringType(), True),
    StructField('l_shipdate', DateType(), True),
    StructField('l_commitdate', DateType(), True),
    StructField('l_receiptdate', DateType(), True),
    StructField('l_shipinstruct', StringType(), True),
    StructField('l_shipmode', StringType(), True),
    StructField('l_comment', StringType(), True)
    ])

#Convert CSV's to Parquet files
data = "s3://bigdatacourse2019/tpch_csv/SF10/"
S3 = "s3://group19-lab5/"

spark.read.csv(data + "region", sep='|', schema=region_s).write.parquet(S3
+ "region")
spark.read.csv(data + "nation", sep='|', schema=nation_s).write.parquet(S3
+ "nation")
spark.read.csv(data + "supplier", sep='|',
schema=supplier_s).write.parquet(S3 + "supplier")
spark.read.csv(data + "customer", sep='|',
schema=customer_s).write.parquet(S3 + "customer")
spark.read.csv(data + "part", sep='|', schema=part_s).write.parquet(S3 +
"part")
spark.read.csv(data + "partsupp", sep='|',
schema=partsupp_s).write.parquet(S3 + "partsupp")
spark.read.csv(data + "orders", sep='|', schema=orders_s).write.parquet(S3
+ "orders")
spark.read.csv(data + "lineitem", sep='|',
schema=lineitem_s).write.parquet(S3 + "lineitem")
```

In figure 5 the runtimes for querying over CSV versus Parquet-files are shown. Spark shows a remarkable performance increase. With the CSV files all data is stored row-wise and the data types were all parsed as string-type, whilst with the Parquet files is stored column-wise and the data type per column is specified. Spark seems to benefit clearly in the form reduced query runtimes thanks to the compression and structuration of the data. The code for executing the queries can be found in appendix 2.6.

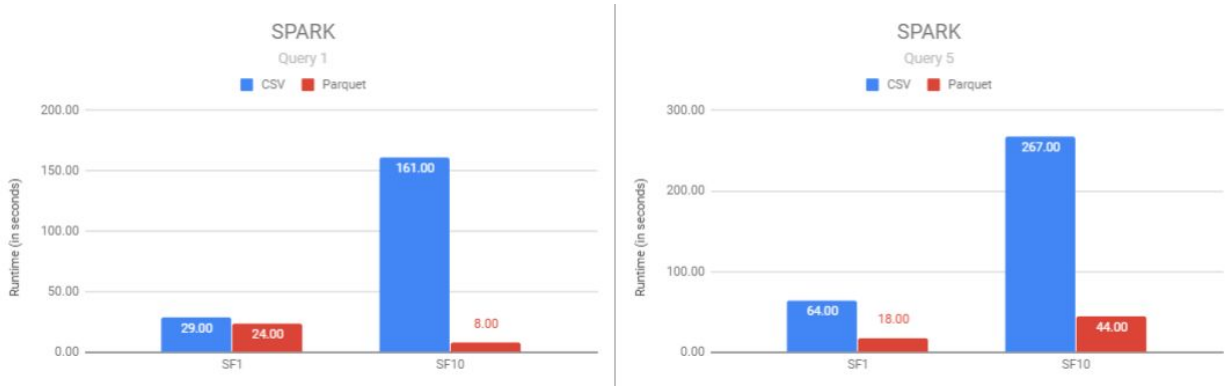

Figure 5: CSV-files versus Parquet-files runtime results for Spark queries

C. When the CSV-files are loaded into Athena the datatypes are specified, and likewise for the Parquet-files. The runtimes of query 1 and 5 for both data types are contrasted in figure 6. It seems that the larger the dataset the more the performance gain. This is in part explained by the reduced files size. Another explanation is the utilization of Parquet's scheme, for example Query 1 accesses less data columns than query 5 and the Parquet characteristic of only reading the necessary columns pays out here the most (table 4).

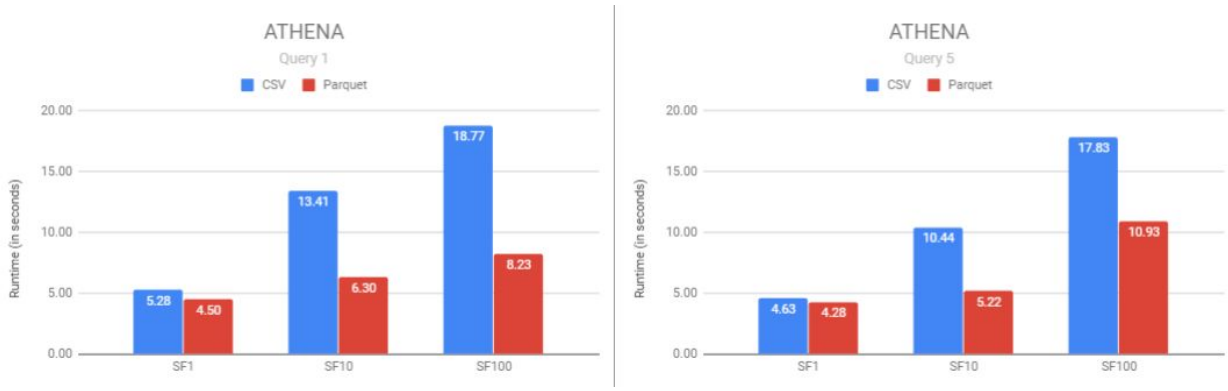

Figure 6: CSV-files versus Parquet-files runtime results for Athena queries

	Athena	
	Query 1	**Query 5**
SF1	17%	8%
SF10	113%	100%
SF100	128%	63%

Table 4: Increase in runtime speed using Parquet files versus CSV files

Question 3

For question two the CSV-files were converted to Parquet-files using Spark, thus for the partitioning Spark can also be applied. In order to decide how to partition, query one and five were analyzed. Query one groups by "l_returnflag" and “l_linestatus”. Query five groups by "n_name". Knowing the group by statement the data of the table Lineitem was partitioned by “l_returnflag”, the table Nation by “n_nation” and the table Region by “r_name”. Below is the code used to partition the tables.

```
spark.read.csv(data + "region", sep='|',
schema=region_s).write.partitionBy("r_name").parquet(S3 + "region")
spark.read.csv(data + "nation", sep='|',
schema=nation_s).write.partitionBy("n_name").parquet(S3 + "nation")
spark.read.csv(data + "lineitem", sep='|',
schema=lineitem_s).write.partitionBy("l_returnflag").parquet(S3 +
"lineitem")
```

The Redshift documentation highlights that table partitioning is not supported [17]. Therefore there will be no comparison of Redshift with Athena, and only the results for Athena will be discussed. When analyzing the results of Athena a performance increase can be seen. The query results are shown in figure 7 and table 5.

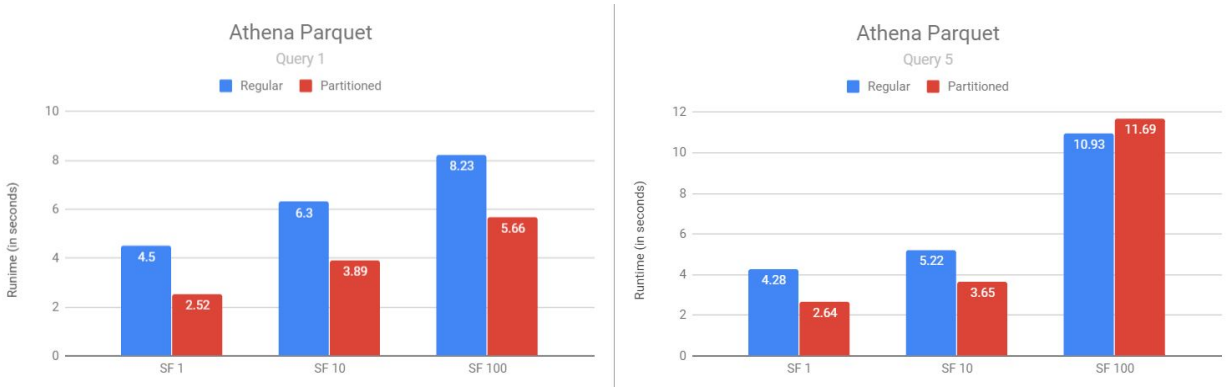

Figure 7: Athena: Partitioned Parquet files compared with regular Parquet files

	Query 1	**Query 5**
SF 1	79%	62%
SF 10	62%	43%
SF 100	45%	-7%

Table 5: Increase in runtime speed using partitioned Parquet files

Query 1 that only queries on the table Lineitem seems to benefit most from the partitioning. However the performance gains decrease when the dataset grows and it would be expected that the trend would be the other way around. This is especially true for query where the performance on the SF100 dataset seems to worsen due to the partitioning. Query 5 refers to multiple tables. This could cause the performance to decay.

Question 4

For one-off standard ad-hoc SQL queries Athena is the best solution, without any setup procedure it can perform SQL queries quickly [1]. Redshift is a data warehouse solution that is great for pulling together data from many different sources and is the best option for complex queries on massive collections of structured data [7]. Redshift Spectrum is actually a feature of Redshift and can run SQL queries on data in Redshift and S3 [7]. Spark SQL is more geared towards custom code to process and analyze extremely large datasets [7].

Before running a query the data needs to be uploaded to a Redshift cluster [4], Spark SQL [6] reads data into a cache whilst Athena [4] and Redshift Spectrum [5] read directly from S3. When running a query Redshift and Redshift Spectrum generate and

compile code for efficient query execution [4][5]. Athena does basic SQL querying [4] and Spark SQL organizes the query execution in a logical plan priorly to running it [6].

Athena and Spark SQL can run queries on structured, semi-structured and unstructured data. Redshift Spectrum can process structured and semi-structured data [8]. And finally Redshift and Spark SQL can only handle structured data [4][9].

ETL stands for Extract, Transform, and Load and is performed priorly to extract value and insight from data [11]. This preprocessing can be done for Athena via Glue, which explores the data and catalogs table definitions and schemas making it searchable for Athena [12]. Redshift has more ETL options, it is possible to build an own ETL pipeline, Glue can be used or third party tools [13]. It is also possible to use Redshift Spectrum instead for ad hoc processing outside the regular Redshift ETL process [13]. It is also possible to do ETL with Spark SQL but this demands some writing of custom queries for this purpose [14].

Overall Athena is best geared for fast ad-hoc querying on a single data source regardless of the organization of the data. Redshift has the best performance on structured datasets, Redshift Spectrum sits a bit in between Athena and Redshift. And Spark has the edge in analyzing complex, iterative and/or interactive workloads.

With Athena and Redshift Spectrum you pay per query [4][10]. Minimum price per query is lower for Athena making it suitable for the smaller queries, although the minimum price per query is higher for Redshift Spectrum it is cheaper when running large queries. Redshift and Spark SQL require setting up a cluster for which an hourly rate is charged [4][6].

Question 5

A. Athena [4][15]

To go through only parts of the data multiple times, we need athena as it supports columnar storage for that purpose. Athena is good for sessions involving several short queries, which is often the case of ad-hoc exploratory data analyses. Athena has libraries for parsing data from all data formats mentioned above and more. If the data is unstructured, Athena can handle it, while Redshift cannot. It also only accepts large queries, which works in this case, if using 1% of the data, which is 160GB, while being large enough to overcome Athena's lower limit of 10MB scanned per query. Since Athena charges per TB, the cost is relatively low.

A query of 1% of the data would cost 80 cents of a euro, and going through the whole data would cost 80 euros. Only the data scanned during a query costs, so scanning data multiple times during the same session should not cost extra. Several queries per day, let's say 10, would cost 8 euros, which means 240 euros per month.

B. Redshift [4][15][16]

Because the data is already structured and the same queries are repeated with each new data to process. We also need continuous storage, thus we can save the data as Parquet on S3. Redshift is built to support BI tools for business. Redshift also handles table structures. The regularity of the queries would make Athena too expensive. On the other hand, Redshift is extremely efficient at feeding new (highly structured) data into already established queries, once the data is loaded on Redshift. And Redshift does not have a lower limit for the amount of data used per query. Because of the hourly query pattern, Redshift's hyperspeed in executing queries is recommended (i.e. making sure a query is finished before starting a new one).

Redshift on Amazon has a yearly subscription which would reduce the cost of this hourly pattern of analyses. Different types of instances are available for $999 for Reserved Instances per TB per year for a 3-year commitment.

C. Athena [4][15]

Because Redshift does not support data partitioning. Athena is also serverless, and will therefore not slow down as several users are accessing and running queries on the same data in S3 (i.e. we recommend the data to be moved to s3).

Assuming a query is 40GB (0,04 TB), 8 queries a day will cost 1 euro 60 cents per user per day, 160 euros across users per day, which accumulates to, a maximum (overestimated) budget of 4800 euros.

References

[1] What is Amazon Athena? (n.d.). Retrieved March 5, 2019, from https://docs.aws.amazon.com/athena/latest/ug/what-is.html

[2] What Is Amazon Redshift? (n.d.). Retrieved March 5, 2019, from https://docs.aws.amazon.com/redshift/latest/mgmt/welcome.html

[3] Factors Affecting Query Performance. (n.d.). Retrieved March 5, 2019, from https://docs.aws.amazon.com/redshift/latest/dg/c-query-performance.html

[4] Burns, B. (2018, November 09). Redshift vs Athena. Retrieved March 6, 2019, from https://dataschool.com/redshift-vs-athena/

[5] Amazon Redshift System Overview. Accessed from https://docs.aws.amazon.com/redshift/latest/dg/c_redshift_system_overview.html

[6] What is Apache Spark? | Introduction to Apache Spark and Analytics | AWS. Accessed from https://aws.amazon.com/big-data/what-is-spark/

[7] Amazon Redshift FAQs - Amazon Web Services. Accessed from https://aws.amazon.com/redshift/faqs/

[8] Using Amazon Redshift Spectrum to Query External Data. Accessed from https://docs.aws.amazon.com/redshift/latest/dg/c-using-spectrum.html

[9] Karau, H., Konwinski, A., Wendell, P., & Zaharia, M. (2015). Learning spark: lightning-fast big data analysis. " O'Reilly Media, Inc.".

[10] Barr, J. (2018, 19. march). Amazon Redshift Spectrum – Exabyte-Scale In-Place Queries of S3 Data | Amazon Web Services. Accessed from https://aws.amazon.com/blogs/aws/amazon-redshift-spectrum-exabyte-scale-in-place-queries-of-s3-data/

[11] Rajamani, S. & Singh, B. (2018, 3. may). How to extract, transform, and load data for analytic processing using AWS Glue (Part 2) | Amazon Web Services. Accessed from https://aws.amazon.com/blogs/database/how-to-extract-transform-and-load-data-for-analytic-processing-using-aws-glue-part-2/

[12] AWS Glue - Amazon Web Services. Accessed from https://aws.amazon.com/glue/

[13] 3 Ways to do Redshift ETL in 2018. Accessed from https://panoply.io/data-warehouse-guide/redshift-etl/

[14] Snively, B. (2019, 5. march). Using Spark SQL for ETL | Amazon Web Services. Accessed from https://aws.amazon.com/blogs/big-data/using-spark-sql-for-etl/

[15] Amazon Athena vs. Redshift. Accessed from https://blog.skeddly.com/2018/01/amazon-athena-vs-amazon-redshift.html

[16] Amazon Redshift – Now Faster and More Cost-Effective than Ever. Accessed from https://aws.amazon.com/blogs/aws/amazon-redshift-now-faster-and-more-cost-effective-than-ever/

[17] Features That Are Implemented Differently. (n.d.). Retrieved March 10, 2019, from

https://docs.aws.amazon.com/redshift/latest/dg/c_redshift-sql-implementated-differently.html

UNIVERSITEIT VAN AMSTERDAM

Appendix

2.1. Output spark query 1 / SF 1 - CSV-file

```
%pyspark
output_1.take(10)
```

SPARK JOB FINISHED

```
[Row(l_returnflag=u'A', l_linestatus=u'F', sum_qty=37734107.0, sum_base_price=56586554400.73021, sum_disc_price=53758257134.86995, sum_charge=5590906522
2.82765, avg_qty=25.522005853257337, avg_price=38273.12973462181, avg_disc=0.04998529583844344, count_order=1478493), Row(l_returnflag=u'N', l_linestatu
s=u'F', sum_qty=991417.0, sum_base_price=1487504710.3799996, sum_disc_price=1413082168.0540998, sum_charge=1469649223.1943738, avg_qty=25.51647192052298
5, avg_price=38284.467760848296, avg_disc=0.050093426674216526, count_order=38854), Row(l_returnflag=u'N', l_linestatus=u'O', sum_qty=73533166.0, sum_ba
se_price=110287596362.17973, sum_disc_price=104774847005.94498, sum_charge=108969626230.3592, avg_qty=25.502145202331544, avg_price=38249.00312934221, a
vg_disc=0.049995841730577174, count_order=2883411), Row(l_returnflag=u'R', l_linestatus=u'F', sum_qty=37719753.0, sum_base_price=56568041380.90001, sum_
disc_price=53741292684.604065, sum_charge=55889619119.831894, avg_qty=25.50579361269077, avg_price=38250.854626099666, avg_disc=0.0500094058301726, coun
t_order=1478870)]
```

Took 29 sec. Last updated by anonymous at March 05 2019, 5:31:15 PM.

2.2. Output spark query 5 / SF 1 - CSV-file

```
%pyspark
output_5.take(10)
```

SPARK JOBS FINISHED

```
[Row(n_name=u'IRAQ', revenue=377721186.8950001), Row(n_name=u'EGYPT', revenue=358930747.2158), Row(n_name=u'SAUDI ARABIA', revenue=353692877.67349994),
Row(n_name=u'IRAN', revenue=334897367.4182), Row(n_name=u'JORDAN', revenue=322217530.6327999)]
```

Took 30 sec. Last updated by anonymous at March 05 2019, 5:32:26 PM.

2.3. Output spark query 1 / SF 10 - CSV-file

```
%pyspark
output_1.take(10)
```

SPARK JOB FINISHED

```
[Row(l_returnflag=u'A', l_linestatus=u'F', sum_qty=377518399.0, sum_base_price=566065727797.2485, sum_disc_price=537759104278.06885, sum_charge=55927667
0892.1189, avg_qty=25.500975103007097, avg_price=38237.15100895845, avg_disc=0.050006574540269584, count_order=14804077), Row(l_returnflag=u'N', l_lines
tatus=u'F', sum_qty=9851614.0, sum_base_price=14767438399.169985, sum_disc_price=14028805792.2114, sum_charge=14590490998.366735, avg_qty=25.52244830284
0946, avg_price=38257.8106600811, avg_disc=0.04997336773765433, count_order=385998), Row(l_returnflag=u'N', l_linestatus=u'O', sum_qty=733701060.0, sum_
base_price=1100171316938.8064, sum_disc_price=1045158178577.217, sum_charge=1086976829223.3827, avg_qty=25.49765126206936, avg_price=38233.26160635132,
avg_disc=0.04999947906591815, count_order=28775241), Row(l_returnflag=u'R', l_linestatus=u'F', sum_qty=377732830.0, sum_base_price=566431054975.9958, su
m_disc_price=538110922664.7652, sum_charge=559634780885.0875, avg_qty=25.50838478968014, avg_price=38251.21927355948, avg_disc=0.04999679231411412, coun
t_order=14808183)]
```

Took 2 min 41 sec. Last updated by anonymous at March 05 2019, 5:01:27 PM.

2.4. Output spark query 5 / SF 10 - CSV-file

```
%pyspark
output_5.take(10)
```

SPARK JOBS FINISHED

```
[Row(n_name=u'IRAQ', revenue=3532508150.7862005), Row(n_name=u'SAUDI ARABIA', revenue=3471277920.322701), Row(n_name=u'EGYPT', revenue=3468615667.170498
4), Row(n_name=u'IRAN', revenue=3467855027.2954993), Row(n_name=u'JORDAN', revenue=3459000319.8578005)]
```

Took 5 min 5 sec. Last updated by anonymous at March 05 2019, 5:29:47 PM.

2.5. Spark code for running the queries on the CSV files

```
#Load data - SF 1
%pyspark
region  =
sqlContext.read.format("com.databricks.spark.csv").option("header",
"false").option("delimiter",
"|").load("s3://bigdatacourse2019/tpch_csv/SF1/region/").withColumnRenamed(
```

```
"_c0", "r_regionkey").withColumnRenamed("_c1",
"r_name").withColumnRenamed("_c2", "r_comment").drop('_c3')
nation =
sqlContext.read.format("com.databricks.spark.csv").option("header",
"false").option("delimiter",
"|").load("s3://bigdatacourse2019/tpch_csv/SF1/nation/").withColumnRenamed(
"_c0", "n_nationkey").withColumnRenamed("_c1",
"n_name").withColumnRenamed("_c2", "n_regionkey").withColumnRenamed("_c3",
"n_comment").drop('_c4')
supplier =
sqlContext.read.format("com.databricks.spark.csv").option("header",
"false").option("delimiter",
"|").load("s3://bigdatacourse2019/tpch_csv/SF1/supplier/").withColumnRename
d("_c0", "s_suppkey").withColumnRenamed("_c1",
"s_name").withColumnRenamed("_c2", "s_address").withColumnRenamed("_c3",
"s_nationkey").withColumnRenamed("_c4", "s_phone").withColumnRenamed("_c5",
"s_acctbal").withColumnRenamed("_c6", "s_comment").drop('_c7')
customer =
sqlContext.read.format("com.databricks.spark.csv").option("header",
"false").option("delimiter",
"|").load("s3://bigdatacourse2019/tpch_csv/SF1/customer/").withColumnRename
d("_c0", "c_custkey").withColumnRenamed("_c1",
"c_name").withColumnRenamed("_c2", "c_address").withColumnRenamed("_c3",
"c_nationkey").withColumnRenamed("_c4", "c_phone").withColumnRenamed("_c5",
"c_acctbal").withColumnRenamed("_c6",
"c_mktsegment").withColumnRenamed("_c7", "c_comment").drop('_c8')
part = sqlContext.read.format("com.databricks.spark.csv").option("header",
"false").option("delimiter",
"|").load("s3://bigdatacourse2019/tpch_csv/SF1/part/").withColumnRenamed("_
c0", "p_partkey").withColumnRenamed("_c1",
"p_name").withColumnRenamed("_c2", "p_mfgr").withColumnRenamed("_c3",
"p_brand").withColumnRenamed("_c4", "p_type").withColumnRenamed("_c5",
"p_size").withColumnRenamed("_c6", "p_container").withColumnRenamed("_c7",
"p_retailprice").withColumnRenamed("_c8", "p_comment").drop('_c9')
partsupp =
sqlContext.read.format("com.databricks.spark.csv").option("header",
"false").option("delimiter",
"|").load("s3://bigdatacourse2019/tpch_csv/SF1/partsupp/").withColumnRename
d("_c0", "ps_partkey").withColumnRenamed("_c1",
"ps_suppkey").withColumnRenamed("_c2",
```

```
"ps_availqty").withColumnRenamed("_c3",
"ps_supplycost").withColumnRenamed("_c4", "ps_comment").drop('_c5')
orders  =
sqlContext.read.format("com.databricks.spark.csv").option("header",
"false").option("delimiter",
"|").load("s3://bigdatacourse2019/tpch_csv/SF1/orders/").withColumnRenamed(
"_c0", "o_orderkey").withColumnRenamed("_c1",
"o_custkey").withColumnRenamed("_c2",
"o_orderstatus").withColumnRenamed("_c3",
"o_totalprice").withColumnRenamed("_c4",
"o_orderdate").withColumnRenamed("_c5",
"o_orderpriority").withColumnRenamed("_c6",
"o_clerk").withColumnRenamed("_c7",
"o_shippriority").withColumnRenamed("_c8", "o_comment").drop('_c9')
lineitem  =
sqlContext.read.format("com.databricks.spark.csv").option("header",
"false").option("delimiter",
"|").load("s3://bigdatacourse2019/tpch_csv/SF1/lineitem/").withColumnRename
d("_c0", "l_orderkey").withColumnRenamed("_c1",
"l_partkey").withColumnRenamed("_c2", "l_suppkey").withColumnRenamed("_c3",
"l_linenumber").withColumnRenamed("_c4",
"l_quantity").withColumnRenamed("_c5",
"l_extendedprice").withColumnRenamed("_c6",
"l_discount").withColumnRenamed("_c7", "l_tax").withColumnRenamed("_c8",
"l_returnflag").withColumnRenamed("_c9",
"l_linestatus").withColumnRenamed("_c10",
"l_shipdate").withColumnRenamed("_c11",
"l_commitdate").withColumnRenamed("_c12",
"l_receiptdate").withColumnRenamed("_c13",
"l_shipinstruct").withColumnRenamed("_c14",
"l_shipmode").withColumnRenamed("_c15", "l_comment").drop('_c16')

#registerTempTable
%pyspark
region.registerTempTable('region')
nation.registerTempTable('nation')
supplier.registerTempTable('supplier')
customer.registerTempTable('customer')
partsupp.registerTempTable('partsupp')
orders.registerTempTable('orders')
```


```
lineitem.registerTempTable('lineitem')

#query one
%pyspark
query = "SELECT lineitem.l_returnflag, lineitem.l_linestatus,
SUM(lineitem.l_quantity) AS sum_qty, SUM(lineitem.l_extendedprice) AS
sum_base_price, SUM(lineitem.l_extendedprice * (1 - lineitem.l_discount))
AS sum_disc_price, SUM(lineitem.l_extendedprice * (1 - lineitem.l_discount)
* (1 + lineitem.l_tax)) AS sum_charge, AVG(lineitem.l_quantity) AS avg_qty,
AVG(lineitem.l_extendedprice) AS avg_price, AVG(lineitem.l_discount) AS
avg_disc, COUNT(*) AS count_order FROM lineitem WHERE lineitem.l_shipdate
<= date '1998-12-01' - interval '108' day GROUP BY lineitem.l_returnflag,
lineitem.l_linestatus ORDER BY lineitem.l_returnflag,
lineitem.l_linestatus"

output_1 = sqlContext.sql(query)
output_1.take(10)

#query five
query_5 = "SELECT nation.n_name, sum(lineitem.l_extendedprice * (1 -
lineitem.l_discount)) as revenue FROM customer, orders, lineitem, supplier,
nation, region WHERE customer.c_custkey = orders.o_custkey AND
lineitem.l_orderkey = orders.o_orderkey AND lineitem.l_suppkey =
supplier.s_suppkey AND customer.c_nationkey = supplier.s_nationkey AND
supplier.s_nationkey = nation.n_nationkey AND nation.n_regionkey =
region.r_regionkey AND region.r_name = 'MIDDLE EAST' GROUP BY nation.n_name
ORDER BY revenue DESC"

output_5 = sqlContext.sql(query_5)
output_5.take(10)


#Load data SF 10
region  =
sqlContext.read.format("com.databricks.spark.csv").option("header",
"false").option("delimiter",
"|").load("s3://bigdatacourse2019/tpch_csv/SF10/region/").withColumnRenamed
("_c0", "r_regionkey").withColumnRenamed("_c1",
"r_name").withColumnRenamed("_c2", "r_comment").drop('_c3')
nation =
```


```
sqlContext.read.format("com.databricks.spark.csv").option("header",
"false").option("delimiter",
"|").load("s3://bigdatacourse2019/tpch_csv/SF10/nation/").withColumnRenamed
("_c0", "n_nationkey").withColumnRenamed("_c1",
"n_name").withColumnRenamed("_c2", "n_regionkey").withColumnRenamed("_c3",
"n_comment").drop('_c4')
supplier =
sqlContext.read.format("com.databricks.spark.csv").option("header",
"false").option("delimiter",
"|").load("s3://bigdatacourse2019/tpch_csv/SF10/supplier/").withColumnRenam
ed("_c0", "s_suppkey").withColumnRenamed("_c1",
"s_name").withColumnRenamed("_c2", "s_address").withColumnRenamed("_c3",
"s_nationkey").withColumnRenamed("_c4", "s_phone").withColumnRenamed("_c5",
"s_acctbal").withColumnRenamed("_c6", "s_comment").drop('_c7')
customer =
sqlContext.read.format("com.databricks.spark.csv").option("header",
"false").option("delimiter",
"|").load("s3://bigdatacourse2019/tpch_csv/SF10/customer/").withColumnRenam
ed("_c0", "c_custkey").withColumnRenamed("_c1",
"c_name").withColumnRenamed("_c2", "c_address").withColumnRenamed("_c3",
"c_nationkey").withColumnRenamed("_c4", "c_phone").withColumnRenamed("_c5",
"c_acctbal").withColumnRenamed("_c6",
"c_mktsegment").withColumnRenamed("_c7", "c_comment").drop('_c8')
part = sqlContext.read.format("com.databricks.spark.csv").option("header",
"false").option("delimiter",
"|").load("s3://bigdatacourse2019/tpch_csv/SF10/part/").withColumnRenamed("
_c0", "p_partkey").withColumnRenamed("_c1",
"p_name").withColumnRenamed("_c2", "p_mfgr").withColumnRenamed("_c3",
"p_brand").withColumnRenamed("_c4", "p_type").withColumnRenamed("_c5",
"p_size").withColumnRenamed("_c6", "p_container").withColumnRenamed("_c7",
"p_retailprice").withColumnRenamed("_c8", "p_comment").drop('_c9')
partsupp =
sqlContext.read.format("com.databricks.spark.csv").option("header",
"false").option("delimiter",
"|").load("s3://bigdatacourse2019/tpch_csv/SF10/partsupp/").withColumnRenam
ed("_c0", "ps_partkey").withColumnRenamed("_c1",
"ps_suppkey").withColumnRenamed("_c2",
"ps_availqty").withColumnRenamed("_c3",
"ps_supplycost").withColumnRenamed("_c4", "ps_comment").drop('_c5')
orders  =
```


```
sqlContext.read.format("com.databricks.spark.csv").option("header",
"false").option("delimiter",
"|").load("s3://bigdatacourse2019/tpch_csv/SF10/orders/").withColumnRenamed
("_c0", "o_orderkey").withColumnRenamed("_c1",
"o_custkey").withColumnRenamed("_c2",
"o_orderstatus").withColumnRenamed("_c3",
"o_totalprice").withColumnRenamed("_c4",
"o_orderdate").withColumnRenamed("_c5",
"o_orderpriority").withColumnRenamed("_c6",
"o_clerk").withColumnRenamed("_c7",
"o_shippriority").withColumnRenamed("_c8", "o_comment").drop('_c9')
lineitem  =
sqlContext.read.format("com.databricks.spark.csv").option("header",
"false").option("delimiter",
"|").load("s3://bigdatacourse2019/tpch_csv/SF10/lineitem/").withColumnRenam
ed("_c0", "l_orderkey").withColumnRenamed("_c1",
"l_partkey").withColumnRenamed("_c2", "l_suppkey").withColumnRenamed("_c3",
"l_linenumber").withColumnRenamed("_c4",
"l_quantity").withColumnRenamed("_c5",
"l_extendedprice").withColumnRenamed("_c6",
"l_discount").withColumnRenamed("_c7", "l_tax").withColumnRenamed("_c8",
"l_returnflag").withColumnRenamed("_c9",
"l_linestatus").withColumnRenamed("_c10",
"l_shipdate").withColumnRenamed("_c11",
"l_commitdate").withColumnRenamed("_c12",
"l_receiptdate").withColumnRenamed("_c13",
"l_shipinstruct").withColumnRenamed("_c14",
"l_shipmode").withColumnRenamed("_c15", "l_comment").drop('_c16')

#registerTempTable
%pyspark
region.registerTempTable('region')
nation.registerTempTable('nation')
supplier.registerTempTable('supplier')
customer.registerTempTable('customer')
partsupp.registerTempTable('partsupp')
orders.registerTempTable('orders')
lineitem.registerTempTable('lineitem')

#query one
```

```
%pyspark
query = "SELECT lineitem.l_returnflag, lineitem.l_linestatus,
SUM(lineitem.l_quantity) AS sum_qty, SUM(lineitem.l_extendedprice) AS
sum_base_price, SUM(lineitem.l_extendedprice * (1 - lineitem.l_discount))
AS sum_disc_price, SUM(lineitem.l_extendedprice * (1 - lineitem.l_discount)
* (1 + lineitem.l_tax)) AS sum_charge, AVG(lineitem.l_quantity) AS avg_qty,
AVG(lineitem.l_extendedprice) AS avg_price, AVG(lineitem.l_discount) AS
avg_disc, COUNT(*) AS count_order FROM lineitem WHERE lineitem.l_shipdate
<= date '1998-12-01' - interval '108' day GROUP BY lineitem.l_returnflag,
lineitem.l_linestatus ORDER BY lineitem.l_returnflag,
lineitem.l_linestatus"

output_1 = sqlContext.sql(query)
output_1.take(10)

#query five
query_5 = "SELECT nation.n_name, sum(lineitem.l_extendedprice * (1 -
lineitem.l_discount)) as revenue FROM customer, orders, lineitem, supplier,
nation, region WHERE customer.c_custkey = orders.o_custkey AND
lineitem.l_orderkey = orders.o_orderkey AND lineitem.l_suppkey =
supplier.s_suppkey AND customer.c_nationkey = supplier.s_nationkey AND
supplier.s_nationkey = nation.n_nationkey AND nation.n_regionkey =
region.r_regionkey AND region.r_name = 'MIDDLE EAST' GROUP BY nation.n_name
ORDER BY revenue DESC"

output_5 = sqlContext.sql(query_5)
output_5.take(10)
```

2.6. Spark code for running queries on Parquet files

```
%pyspark
SF1 = "s3://bigdatacourse2019/tpch_parquet/SF1/"
SF10 = "s3://group19-lab5/"
dataset = SF1

#load data
region = spark.read.parquet(dataset + 'region/').drop('skip')
nation = spark.read.parquet(dataset + 'nation/').drop('skip')
supplier = spark.read.parquet(dataset + 'supplier/').drop('skip')
customer = spark.read.parquet(dataset + 'customer/').drop('skip')
```

```
part = spark.read.parquet(dataset + 'part/').drop('skip')
partsupp = spark.read.parquet(dataset + 'partsupp/').drop('skip')
orders = spark.read.parquet(dataset + 'orders/').drop('skip')
lineitem = spark.read.parquet(dataset + 'lineitem/').drop('skip')

#create temp tables
region.registerTempTable('region')
nation.registerTempTable('nation')
supplier.registerTempTable('supplier')
customer.registerTempTable('customer')
part.registerTempTable('part')
partsupp.registerTempTable('partsupp')
orders.registerTempTable('orders')
lineitem.registerTempTable('lineitem')

#Query 1
query = "SELECT lineitem.l_returnflag, lineitem.l_linestatus,
SUM(lineitem.l_quantity) AS sum_qty, SUM(lineitem.l_extendedprice) AS
sum_base_price, SUM(lineitem.l_extendedprice * (1 - lineitem.l_discount))
AS sum_disc_price, SUM(lineitem.l_extendedprice * (1 - lineitem.l_discount)
* (1 + lineitem.l_tax)) AS sum_charge, AVG(lineitem.l_quantity) AS avg_qty,
AVG(lineitem.l_extendedprice) AS avg_price, AVG(lineitem.l_discount) AS
avg_disc, COUNT(*) AS count_order FROM lineitem WHERE lineitem.l_shipdate
<= date '1998-12-01' - interval '108' day GROUP BY lineitem.l_returnflag,
lineitem.l_linestatus ORDER BY lineitem.l_returnflag,
lineitem.l_linestatus"

output_1 = sqlContext.sql(query)
output_1.take(10)

#query five
query_5 = "SELECT nation.n_name, sum(lineitem.l_extendedprice * (1 -
lineitem.l_discount)) as revenue FROM customer, orders, lineitem, supplier,
nation, region WHERE customer.c_custkey = orders.o_custkey AND
lineitem.l_orderkey = orders.o_orderkey AND lineitem.l_suppkey =
supplier.s_suppkey AND customer.c_nationkey = supplier.s_nationkey AND
supplier.s_nationkey = nation.n_nationkey AND nation.n_regionkey =
region.r_regionkey AND region.r_name = 'MIDDLE EAST' GROUP BY nation.n_name
ORDER BY revenue DESC"
```

```
output_5 = sqlContext.sql(query_5)
output_5.take(10)
```

2.7. Output spark query 1 / SF 1 - Parquet file

```
%pyspark

#Query 1

query = "SELECT lineitem.l_returnflag, lineitem.l_linestatus, SUM(lineitem.l_quantity) AS sum_qty, SUM(lineitem.l_extendedprice) AS sum_base_price, SUM(lineitem.l_extendedprice * (1 - lineitem
    .l_discount)) AS sum_disc_price, SUM(lineitem.l_extendedprice * (1 - lineitem.l_discount) * (1 + lineitem.l_tax)) AS sum_charge, AVG(lineitem.l_quantity) AS avg_qty, AVG(lineitem.l_extendedprice) AS
    avg_price, AVG(lineitem.l_discount) AS avg_disc, COUNT(*) AS count_order FROM lineitem WHERE lineitem.l_shipdate <= date '1998-12-01' - interval '108' day GROUP BY lineitem.l_returnflag, lineitem
    .l_linestatus ORDER BY lineitem.l_returnflag, lineitem.l_linestatus"

output_1 = sqlContext.sql(query)
output_1.take(10)
```

FINISHED

```
[Row(l_returnflag=u'A', l_linestatus=u'F', sum_qty=37734107, sum_base_price=Decimal('56586554400.73'), sum_disc_price=Decimal('53758257134.8700'), sum_charge=Decimal('55909065222.827692'), avg_qty=25.522
005853257337, avg_price=Decimal('38273.129735'), avg_disc=Decimal('0.049985'), count_order=1478493), Row(l_returnflag=u'N', l_linestatus=u'F', sum_qty=991417, sum_base_price=Decimal('1487504710.38'), sum
_disc_price=Decimal('1413082168.0541'), sum_charge=Decimal('1469649223.194375'), avg_qty=25.516471920522985, avg_price=Decimal('38284.467761'), avg_disc=Decimal('0.050093'), count_order=38854), Row(l_ret
urnflag=u'N', l_linestatus=u'O', sum_qty=73533166, sum_base_price=Decimal('110287596362.18'), sum_disc_price=Decimal('104774847005.9449'), sum_charge=Decimal('108969626230.358561'), avg_qty=25.5021452023
31544, avg_price=Decimal('38249.003129'), avg_disc=Decimal('0.049996'), count_order=2883411), Row(l_returnflag=u'R', l_linestatus=u'F', sum_qty=37719753, sum_base_price=Decimal('56568041380.90'), sum_dis
c_price=Decimal('53741292684.6040'), sum_charge=Decimal('55889619119.831932'), avg_qty=25.50579361269077, avg_price=Decimal('38250.854626'), avg_disc=Decimal('0.050009'), count_order=1478870)]
```

Took 24 sec. Last updated by anonymous at March 06 2019, 9:19:53 AM.

2.8. Output spark query 5 / SF 1 - Parquet file

```
%pyspark
#Query 5

query_5 = "SELECT nation.n_name, sum(lineitem.l_extendedprice * (1 - lineitem.l_discount)) as revenue FROM customer, orders, lineitem, supplier, nation, region WHERE customer.c_custkey = orders
    .o_custkey AND lineitem.l_orderkey = orders.o_orderkey AND lineitem.l_suppkey = supplier.s_suppkey AND customer.c_nationkey = supplier.s_nationkey AND supplier.s_nationkey = nation.n_nationkey AND
    nation.n_regionkey = region.r_regionkey AND region.r_name = 'MIDDLE EAST' GROUP BY nation.n_name ORDER BY revenue DESC"

output_5 = sqlContext.sql(query_5)
output_5.take(10)
```

FINISHED

```
[Row(n_name=u'IRAQ', revenue=Decimal('377721186.8950')), Row(n_name=u'EGYPT', revenue=Decimal('358930747.2158')), Row(n_name=u'SAUDI ARABIA', revenue=Decimal('353692877.6735')), Row(n_name=u'IRAN', reven
ue=Decimal('334897367.4182')), Row(n_name=u'JORDAN', revenue=Decimal('322217530.6328'))]
```

Took 18 sec. Last updated by anonymous at March 06 2019, 9:20:30 AM.

2.9. Output spark query 1 / SF 10 - Parquet file

```
%pyspark

#Query 1

query = "SELECT lineitem.l_returnflag, lineitem.l_linestatus, SUM(lineitem.l_quantity) AS sum_qty, SUM(lineitem.l_extendedprice) AS sum_base_price, SUM(lineitem.l_extendedprice * (1 - lineitem
    .l_discount)) AS sum_disc_price, SUM(lineitem.l_extendedprice * (1 - lineitem.l_discount) * (1 + lineitem.l_tax)) AS sum_charge, AVG(lineitem.l_quantity) AS avg_qty, AVG(lineitem.l_extendedprice) AS
    avg_price, AVG(lineitem.l_discount) AS avg_disc, COUNT(*) AS count_order FROM lineitem WHERE lineitem.l_shipdate <= date '1998-12-01' - interval '108' day GROUP BY lineitem.l_returnflag, lineitem
    .l_linestatus ORDER BY lineitem.l_returnflag, lineitem.l_linestatus"

output_1 = sqlContext.sql(query)
output_1.take(10)
```

FINISHED

```
[Row(l_returnflag=u'A', l_linestatus=u'F', sum_qty=377518399, sum_base_price=566065727788.8718, sum_disc_price=537759103628.59125, sum_charge=559276663930.3895, avg_qty=25.500975103007097, avg_price=3823
7.15100839261, avg_disc=0.0500065746084936, count_order=14804077), Row(l_returnflag=u'N', l_linestatus=u'F', sum_qty=9851614, sum_base_price=14767438399.140808, sum_disc_price=14028805774.311157, sum_cha
rge=14590490817.507568, avg_qty=25.522448302840946, avg_price=38257.81066000551, avg_disc=0.049973367803137, count_order=385998), Row(l_returnflag=u'N', l_linestatus=u'O', sum_qty=733701060, sum_base_pri
ce=1100171316908.52, sum_disc_price=1045158177321.076, sum_charge=1086976815718.1617, avg_qty=25.49765126206936, avg_price=38233.26160529881, avg_disc=0.04999947913372348, count_order=28775241), Row(l_re
turnflag=u'R', l_linestatus=u'F', sum_qty=377732830, sum_base_price=566431054961.8364, sum_disc_price=538110922026.81055, sum_charge=559634773945.3057, avg_qty=25.50838478968014, avg_price=38251.21927260
329, avg_disc=0.04999679238145127, count_order=14808183)]
```

Took 8 sec. Last updated by anonymous at March 06 2019, 9:16:57 AM. (outdated)

2.10. Output spark query 5 / SF 1 - Parquet file

```
%pyspark
#Query 5

query_5 = "SELECT nation.n_name, sum(lineitem.l_extendedprice * (1 - lineitem.l_discount)) as revenue FROM customer, orders, lineitem, supplier, nation, region WHERE customer.c_custkey = orders
    .o_custkey AND lineitem.l_orderkey = orders.o_orderkey AND lineitem.l_suppkey = supplier.s_suppkey AND customer.c_nationkey = supplier.s_nationkey AND supplier.s_nationkey = nation.n_nationkey AND
    nation.n_regionkey = region.r_regionkey AND region.r_name = 'MIDDLE EAST' GROUP BY nation.n_name ORDER BY revenue DESC"

output_5 = sqlContext.sql(query_5)
output_5.take(10)
```

FINISHED

```
[Row(n_name=u'IRAQ', revenue=3532508146.326233), Row(n_name=u'SAUDI ARABIA', revenue=3471277916.66333), Row(n_name=u'EGYPT', revenue=3468615663.070801), Row(n_name=u'IRAN', revenue=3467855022.1932983), R
ow(n_name=u'JORDAN', revenue=3459000316.529175)]
```

Took 44 sec. Last updated by anonymous at March 06 2019, 9:22:09 AM.