Lab Assignment 5
Due Date: 08/03/2019
Jeroen Schmidt (122808560) & Dimitris Michailidis (12325929)
Group 10

Big Data
Prof. P. Boncz
TA: Dean De Leo

1 Athena & Redshift Execution Time Comparison

Before we conduct the experiments and analyze the results, we take a look at the provided queries (TPC-H Query 1 & Query 5) and try to decode and interpret them.

1.1 TPC-H Query 1

Returns a summary report for all items in `lineitem` that where shipped before or at '1998-08-15' for the Redshift query and before or at '1998-09-02' for the Athena query.

The summary items are grouped first by their return flag and then their line status, leading to the following combinations (A-F, N-F, N-O, R-F). The calculated fields are the same for both systems and contain information such as the sum and the average quantity of the order, the sum of the base price, the sum of the price after discount and tax, the average price and the total order count.

This query contains basic operations like `filter(from)`, `where`, `group_by` and `order_by`.

1.2 TPC-H Query 5

Returns a summary report of the total revenue on countries in the Middle East, groupped by the country name(note the revenue is calculated here as the sum of the base price minus the discount price). The countries returned are Iraq, Egypt, Saudi Arabia, Iran and Jordan.

This is an aggregated query which combines joining multiple tables, aggregating, grouping and ordering.

The results of the experiments are presented in Tables 1, 2 and 3 and are discussed in the following subsections.

1.3 Results Comparison

When it comes to results of running Athena and Redshift, we can observe that the results are the same, even though we set different dates in the queries. That is, because the database is a predefined benchmark so we can expect orders to be placed in predefined dates. The results of running Query 1 and Query 5 on the Scale Factor 1 database can be seen in Listings 1 and 2.

Listing 1: SF1 Results of Executing Query 1.

```
"L_RETURNFLAG","L_LINESTATUS","_col2","_col3","_col4","_col5","_col6","_col7","_col8","_col9"
"A","F","37734107","56586554400.73","53758257134.8700","55909065222.827692","25.522005853257337","38273.13","0.05","1478493"
"N","F","991417","1487504710.38","1413082168.0541","1469649223.194375","25.516471920522985","38284.47","0.05","38854"
"N","O","74476040","111701729697.74","106118230307.6056","110367043872.497010","25.50222676958499","38249.12","0.05","2920374"
"R","F","37719753","56568041380.90","53741292684.6040","55889619119.831932","25.50579361269077","38250.85","0.05","1478870"
```

Listing 2: SF1 Results of Executing Query 5.

```
"n_name","revenue"
"IRAQ","377721186.8950"
"EGYPT","358930747.2158"
"SAUDI ARABIA","353692877.6735"
"IRAN","334897367.4182"
"JORDAN","322217530.6328"
```

1.4 Execution Time Comparison - Different Scale Factor

In this section we will compare the first run of the queries on Redshift versus the average Athena run times. The reason why we choose only the first Redshift execution is explained in Subsection 1.5 and in more details in Section 4. As we can see in Table 1, on Scale Factor 1 database Query 1 runs twice as fast on Athena than it does on Redshift, while Query 5 runs more than 5 times faster. On Scale Factor 10, Athena still outperforms Redshift but by a lesser margin (2 and 3 times respectively). Finally, on Scale Factor 100 both systems need about the same time to run Query 1, while Athena still outperforms Redshift on the execution time of Query 5. The results for both systems are plotted in Figures 1 and 2. Figure 3 shows the total data scanned by the Athena system.

1.5 Execution Time Comparison - Between Executions

As we can observe in Tables 1, 2 and 3, Redshift executes the query very slowly on the first time, and then almost instantly on all of the rest executions. Moreover, the execution times of the rest of the runs are almost the same, no matter what the scale factor is. This happens for the following reasons[1]:

- **Code Compilation**: Redshift will generate and compile code for each new query that is executed by the user. Therefore, whenever the query is ran for the first time we have to amount for the interpreter overhead cost in time. After the query is ran once, it does not have to be compiled again.

- **Results Cache**: Redshift will cahce the results of a query in order to return the same results on the same query very fast. Therefore, when we execute the same query multiple times, it will only have to load data on the first one. Then it will just return the same cached data. This is something that can be disabled using a parameter.

On the other hand, Athena queries run at the same speed regardless of if they have been run before. They are much faster, because they query S3 directly and do not require any compilation before. Although Amazon does not exactly disclose how Athena queries are run, it is known that it uses Presto to query tables, which combines data from multiple data sources and therefore is suitable for this system.

[1]https://docs.aws.amazon.com/redshift/latest/dg/c-query-performance.html

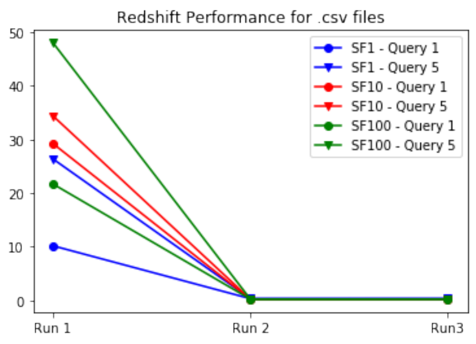


Figure 1: Redshift performance on .csv files

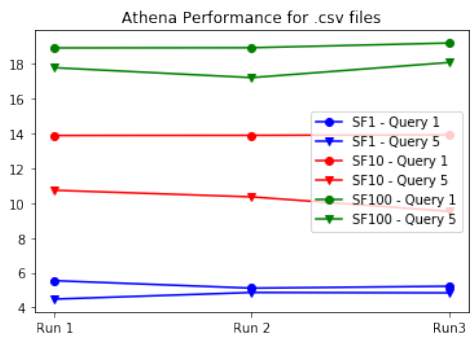


Figure 2: Athena performance on .csv files

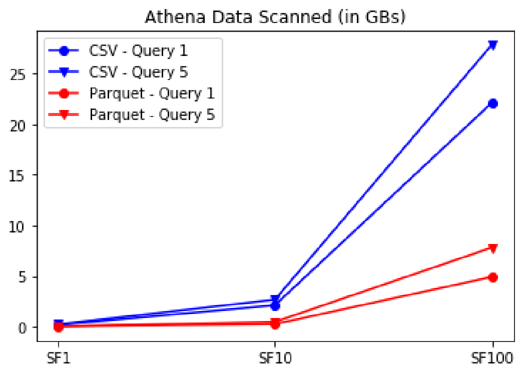


Figure 3: Total data scanned by Athena(in GBs) for CSV and Parquet files.

Redshift	Run 1	Run 2	Run 3	Average
Q1	10 s 17 ms	312 ms	310 ms	3.597s
Q5	26 s 358 ms	314 ms	325 ms	9s
Athena	Run 1	Run 2	Run 3	Average
Q1	5.54s(211.96 MB)	5.11s(211.96 MB)	5.22s(211.96 MB)	5.29s(211.96 MB)
Q5	4.48s(266.6 MB)	4.86s(266.6 MB)	4.84s(266.6 MB)	4.73s(266.6 MB)
SparkSQL	Run 1	Run 2	Run 3	Average
Q1	56.339s	33.615s	39.423s	43.126s
Q5	40.476s	45.219s	46.267s	43.988s

Table 1: Query Execution Times for SF1

Redshift	Run 1	Run 2	Run 3	Average
Q1	29.22s	0.2s	0.2s	9.89s
Q5	34.38s	0.22s	0.22s	11.6s
Athena	Run 1	Run 2	Run 3	Average
Q1	13.89s(2.14 GB)	13.9s(2.14 GB)	13.93s(2.14 GB)	13.9(2.14GB)
Q5	10.74s(2.69 GB)	10.36s(2.69 GB)	9.53s(2.69 GB)	10.21(2.69 GB)
SparkSQL	Run 1	Run 2	Run 3	Average
Q1	02:42.877m	02:39.984m	02:53.161m	02:45:341m
Q5	02:20.916m	02:21.090m	03:16.405m	02:52:803m

Table 2: Query Execution Times for SF10

Redshift	Run 1	Run 2	Run 3	Average
Q1	21.69s	0.22s	0.22s	7.38s
Q5	48.01s	0.21s	0.21s	16.14s
Athena	Run 1	Run 2	Run 3	Average
Q1	18.93s(22.07 GB)	18.94s(22.07 GB)	19.21s(22.07 GB)	19.03s(22.07 GB)
Q5	17.79s(27.78 GB)	17.22s(27.78 GB)	18.09s(27.78 GB)	17.7s(27.78 GB)

Table 3: Query Execution Times for SF100

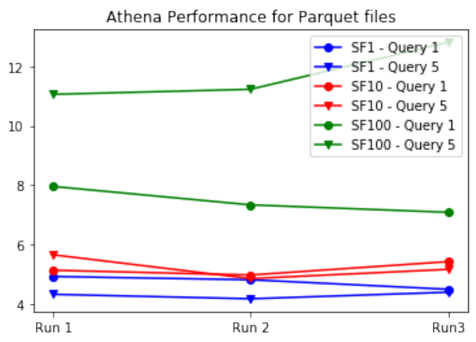


Figure 4: Athena performance on Parquet files.

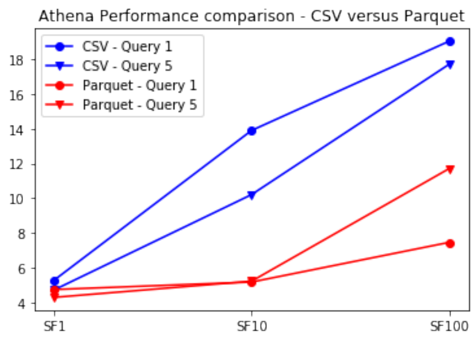


Figure 5: Athena performance comparison between CSV and Parquet files.

2 Spark & Athena Execution Time - Parquet Data Format

Tables 1 and 2 show the execution times for Query 1 and Query 5 in SparkSQL. As we can see, SparkSQL is way slower than both Redshift and Athena, with SF100 needing about 3 minutes to run the queries. Therefore the execution time is not comparable to the other two systems.

2.1 Converting JSON Data to Parquet

A 3 node cluster was created for EMR using Spark 3.3.2 consisting of one m3.xlarge master and 3 m3.xlarge workers. The source code that we used to convert .csv data to parquet is listed under Listing 5 in the Appendix.

2.2 Loading Parquet Data in Athena

In order to load Parquet data to Athena we need to modify the create scripts as follows: 1) remove the 'ROW FORMAT DELIMITED FIELDS TERMINATED BY '—" command, 2) replace it with 'STORED AS PARQUET' and 3) add the command 'tblproperties ("parquet.compress"="SNAPPY");' in the end of the query.

2.3 Executing Queries on Parquet Data

SparkSQL

We can observe in Tables 4, 5 and 6 that SparkSQL also outperforms itself when it loads the data from parquet files, compared to data loaded from csv files. Specifically for SparkSQL, we can see that the performance almost doubles when we use parquet. This is because of parquet's very efficient column storage which includes compression and encoding schemes.

Athena

Figure 4 shows the performance of Athena when ran on Parquet files, while Figure 5 compares the performance of Athena between .csv and parquet files. The total data scanned from parquet source is presented in Figure 3. We can clearly observe how Athena outferforms itself when loading from parquet files compared to when loading from csv files.

Another interesting observation we can make for Athena regards the scaling factors. As Tables 4 and 5 show it needs almost the same time to run the queries in SF1 and SF10. Q1 also runs at similar times on SF100, however the time doubles when it comes to Q5.

SparkSQL	Run 1	Run 2	Run 3	Average
Q1	37.127s	8.543s	6.430s	17.37s
Q5	24.889s	16.988s	11.433s	17.77s
Athena	Run 1	Run 2	Run 3	Average
Q1	4.93s(50.51 MB)	4.82s(50.51 MB)	4.5s(50.51MB)	4.75s(50.51MB)
Q5	4.33(67.06MB)	4.18s(67.06MB)	4.4s(67.06MB)	4.3s(67.06MB)

Table 4: Q1 and Q5 Ran on Parquet - Query Execution Times for SF1

SparkSQL	Run 1	Run 2	Run 3	Average
Q1	52.52s	55.78s	21.71s	
Q5	2:24.14m	1:31.27m	01:20.24m	
Athena	Run 1	Run 2	Run 3	Average
Q1	5.14s(284.27MB)	4.98s(284.27MB)	5.43s(284.27MB)	5.18(284.27MB)
Q5	5.66s(506.82MB)	4.86s(506.82MB)	5.17s(506.82MB)	5.23(506.82MB)

Table 5: Q1 and Q5 Ran on Parquet - Query Execution Times for SF10

Athena	Run 1	Run 2	Run 3	Average
Q1	7.96s(4.93GB)	7.34s(4.93GB)	7.09s(4.93GB)	7.46(4.93GB)
Q5	11.06s(7.82GB)	11.23s(7.82GB)	12.81s(7.82GB)	11.7(7.82GB)

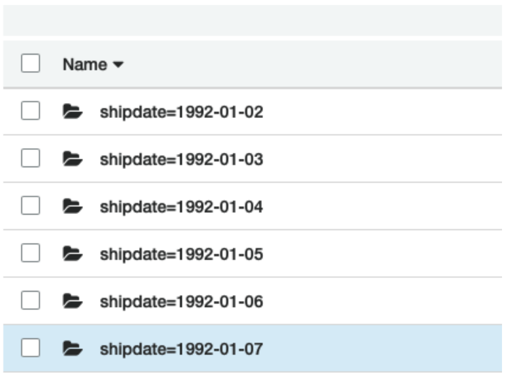
Table 6: Q1 and Q5 Ran on Parquet - Query Execution Times for SF100

3 Athena & Redshift Data Partitioning

3.1 Partitioning Strategies

The first partitioning strategy we tried was partitioning the lineitem table by the full date. This created the folder structure as seen in Figure 6. The idea behind this strategy is that Q1 filters by ShipDate before returning results to the user. We explain why this strategy does not work in the following subsections.

We also considered partitioning the date only by year or month but that would still not give desired results for the same reason. Finally, a strategy we considered was partitioning the Region table by Region name, since in Q5 we only query the Middle East countries. This is also not a good idea because it increases the reading overhead by having each different record in its own partition. We concluded that for these two queries there is no valid partitioning strategy that would provide a comparable performance boost.



Name ▾
shipdate=1992-01-02
shipdate=1992-01-03
shipdate=1992-01-04
shipdate=1992-01-05
shipdate=1992-01-06
shipdate=1992-01-07

Figure 6: Lineitem partinioned by L_ShipDate.

3.2 Partitioning Code

Before we load the data into the two systems, we need to partition them. We will partition the `lineitem` table by shipdate and suppkey. We define it the same way we show in Listing 5 and then run the code presented in Listing 3.

Listing 3: Code used to partition the lineitem table by date and suppkey.

```
 lineitem = spark.read.csv(dir + "lineitem", sep='|',schema=lineitem_s)
 lineitem.write.parquet("s3://group10-lab5/partioned/date-strategy/lineitem",
                        partitionBy="L_SHIPDATE")
lineitem.write.parquet("s3://group10-lab5/partioned/suppkey-strategy/lineitem",
                        partitionBy="L_SUPPKEY")
```

3.3 Load Partitioned Data

To create the schema and load the partitions into Athena, we used the code under Listing 4. Then we used the command MSCK REPAIR TABLE lineitem to load the partitions. TODO WRITE ABOUT LOADING IN REDSHIFT.

Listing 4: Code used to load partitioned data in Athena.

```
# CREATE EXTERNAL TABLE table {
# table definition
}
```

```
PARTITIONED BY (shipdate date)
STORED AS PARQUET
LOCATION 's3://group10-lab5/partitioned2/date-strategy/lineitem'
tblproperties ("parquet.compress"="SNAPPY");
```

3.4 Performance Comparisons

Table 7 shows the performance of the queries in Athena, using the date strategy. As we can see, the performance is much slower than with the non-partitioned data, therefore this strategy has failed. The reason behind it is that, even though we partition the data, Query 1 filters by date and requests almost all the dates. Specifically, the date range of L_SHIPDATE is from 1992-01-02 to 1998-12-01, with more than 2500 days in between. The query filters for dates before 1998-09-02, thus making this partitioning strategy not efficient.

Athena	Run 1	Run 2	Run 3	Average
Q1	20.94s(342.8MB)	21.78s(342.8MB)	16.54s(342.8MB)	19.753(342.8MB)
Q5	19.61s(854.68 MB)	20.64s(854.68 MB)	25.38s(854.68 MB)	21.877(854.68 MB)

Table 7: Q1 and Q5 Ran on Athena with the Lineitem Date Stratgegy

4 Comparison of Athena, Redshift, Redshift Spectrum and Spark SQL

We will start comparing the systems by giving a generic overview of what each one of them is.

- **Athena:** an interactive query service that enables you to query and analyze data stored in Amazon S3, using standard SQL queries. Athena is serverless, so there is no infrastructure to manage and you pay only per ran query[2].

- **Redshift:** a cloud based, fully managed and scalable data warehouse that enables you to analyze all the data across a warehouse or data lake. It is based on PostgreSQL and is designed for fast query and I/O performance on any size, again using standard SQL. To run it requires a cluster to be setup[3].

- **Redshift Spectrum:** a built in feature of Amazon Redshift. It enables you to perform SQL queries on multiple data sources and even combine them, eliminating the need to transfer data from a storage service to a database(just like Athena). It is also serverless, based on simple SQL queries and can execute highly sophisticated queries against an exabyte of data very fast[4].

- **SparkSQL:** a module in Spark that enables the processing of structured data, usually stored in relational databases. Unlike the basic Spark RDD API, SparkSQL provide Spark with more information about the structure of the data and the computation being performed[5].

[2]https://aws.amazon.com/athena/
[3]https://dataschool.com/redshift-vs-athena/
[4]https://aws.amazon.com/blogs/big-data/amazon-redshift-spectrum-extends-data-warehousing-out-to-exabytes-no-loading-required/
[5]https://spark.apache.org/docs/2.3.0/sql-programming-guide.html

4.1 Query Workload

Athena queries are based on Facebook's Presto, a distributed SQL query engine designed to query large data sets distributed over one or more heterogeneous data sources[6]. Presto's distributed nature allows for SQL queries to be parsed and planned before parallel tasks are scheduled to worker nodes. Workers then jointly process rows from the data sources and return results to the client. It is faster than traditional Apache Hive models because it does not write intermediate results to the disk[7].

In Redshift, queries need to be compiled before executed. Figure 7 shows the query execution process, which is explained in detail below[8]:

- The execution engine begins by parsing the SQL code.
- Then produces a query tree that is a logical representation of the original query and sents it to the optimizer.
- The query is rewritten if necessary after evaluation.
- The optimizer generates a query plan for the execution with the best performance.
- The query plan is transalted into: 1) steps: individual operations during query execution, 2) segments: conbinations of steps that can be done in a single process, 3) streams: collection of segments to be parceled out over the available compute node slices.
- The compute node slices execute the segments in parallel.

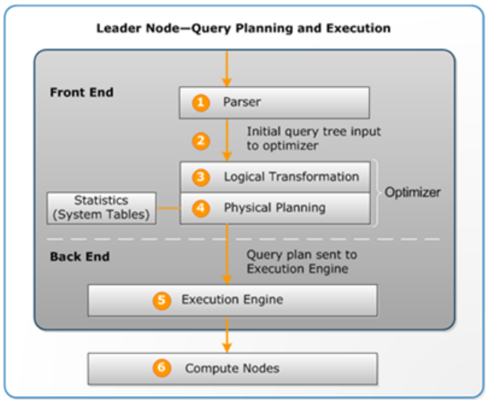

Leader Node—Query Planning and Execution
Front End
1 Parser
2 Initial query tree input to optimizer
3 Logical Transformation
4 Physical Planning
Optimizer
Statistics (System Tables)
Back End
Query plan sent to Execution Engine
5 Execution Engine
6 Compute Nodes

Figure 7: Query planning and Execution in Redshift.

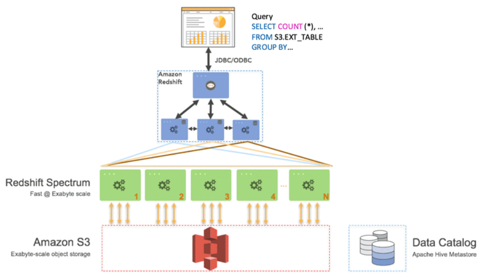


Figure 8: Query Planning and Execution in Redshift Spectrum.

In Reshift Spectrum, a similar process is followed until data needs to be read from external tables. Then the compute nodes obtain the information describing the external tables, and prune the non-relevant partitions, based on the filters and joins in the queries. They then generate multiple requests depending on the number of objects that need to be processed before submitting them concurrently to Redshift Spectrum, which then scan, filter and aggregate the data from S3, streaming the required data for processing back to the cluster. Finally, in the

[6]https://prestodb.github.io/docs/0.172/functions.html
[7]https://en.wikipedia.org/wiki/Presto_(SQL_query_engine)
[8]https://docs.aws.amazon.com/redshift/latest/dg/c-query-planning.html

cluster, join and merge operations are performed and results are returned to the client[9]. The execution plan is presented in Figure 8.

SparkSQL is a high level API for executing processes in spark Resilient Distributed Datasets(RDDs). SparkSQL provide Spark with information about the structure of the data and the computation being performed. Queries built in SparkSQL are optimized by Catalyst. Catalyst employs advanced programming language features and behaves as an extensible query optimizer, making it easy to enhance the optimization techniques, as well as extend the them with new features'[10].

4.2 Data Characteristics

Athena reads data directly from S3 and uses SerDe[11] to deserialize different formats like CSV, JSON, Parquet, etc. It also supports complex data types like arrays, maps and structs. Redshift on the other hand also uses SerDe and can copy data from the same formats, but it does not suport arrays or object identifier types[12]. Redshift Spectrum solves this problem by being able to work with both scalar data from Redshift and complex/nested data saved in S3[13]. Finally, SparkSQL supports different data sources like Hive, Avro, Parquet, ORC, JSON, as well as JDBC, assisting to join the data from these sources. It also supports complex types like arrays and maps.

4.3 ETL

Athena is a query service and does not store or copy any data, nor does it run on top of an infrastructure. Before any data is read, a schema needs to be created so that SerDe knows how to deserialise the fields.

Redshift runs on top of a cluster, therefore it needs to load the data before performing any operations. Usually, data is stored in S3 and then copied to Redshift for query execution. As Redshift also uses SerDe, a schema needs to be defined before copying data. There are 3 ways of doing ETL in Redshift[14]:

- Custom Redshift ETL Pipeline: the developers write a pipeline to copy or bulk load data from other sources.
- AWS Glue: a managed ETL service based on a serverless architecture, can be leveraged to do the ETL.
- Third Party Tools: for when it is needed to integrate with external data sources, while Glue is only integrated with Amazon infrastructure.

SparkSQL can be employed to perform effective and performant ETL. By creating an Amazon EMR cluster and combining it with the benefits of Spark SQLs optimized execution engine(Catalyst), we can achieve very fast ETL tasks.

[9]https://aws.amazon.com/blogs/big-data/amazon-redshift-spectrum-extends-data-warehousing-out-to-exabytes-no-loading-required/
[10]https://data-flair.training/blogs/spark-sql-tutorial
[11]https://github.com/serde-rs/serde
[12]https://dataschool.com/redshift-vs-athena/
[13]https://aws.amazon.com/about-aws/whats-new/2018/08/amazon-redshift-announces-support-for-nested-data-with-redshift-spectrum/
[14]https://panoply.io/data-warehouse-guide/redshift-etl/

4.4 Overall Performance

Athena is built for running queries on a single data source, regardless of data organization. It is supposed to be faster than Redshift for simple and aggregated reads. However, Redshift outperforms Athena on complex queries with multiple data sources, for example Join queries (even though it's not demonstrated in this paper, as the join query we run is fairly simple). Athena and Spectrum perform at about the same rate for data that is not partitioned, with Athena being a bit faster[15]. SparkSQL's performance leverages the power of RDDs and can be optimised by using data caching.

4.5 Usability

All three technologies are offered as managed services though Amazon Web Services, as such they are quick to setup and start using them. However, Apache Spark (used through Spark) could become more difficult to configure and change for bespoke use cases because it would need the infrastructure manager to take into account several other technology stacks that Spark interacts with(such as hadoop, hive and distributed networking). The same goes for Redshift, since it requires a solid ETL process to effectively copy only the needed data in the cluster. Athena and Redshift spectrum are in general more user friendly, as users can write queries in SQL, while the frameworks take care of executing them.

4.6 Cost

Spark (EMR):
The following equations include cost of running EMR and the S3 storage needed for the data. Let sub-script i represent a EC2 machine (from a set of machine in the cluster) that is being used.

$$\begin{aligned}
&\text{EMR Cost} * \text{Number of EMR Machines} + ... \\
&\sum_i (\text{Number of EC2 Machine of Type}_i * \text{Cost of EC2 Machine Type}_i) * \text{Hours Run} + ... \\
&\text{S3 Storage Used (Gb)} * \text{S3 Cost for Storage Bracket}
\end{aligned}$$

Redshift:
The following calculation is based on the on-demand pricing structure. Redshift systems are usually run 24hours a day because they are used as data warehouses but there are architectures spin them up and down. Let sub-script i represent a Redshift machine (from a set of machine in the cluster) that is being used.

$$\sum_i (\text{Number of RS Machine Type}_i * \text{Cost of RS Machine Type}_i) * \text{hours run}$$

$$\text{s.t. Data Size} < \sum_i \text{Storage in RS Machine Type}_i * \text{Number of RS Machine Type}_i$$

Athena:
One needs to account for the query cost and the s3 storage cost for the data that Athena queries against. This results in the following cost calculation formula:

$$\begin{aligned}
&\$5 * \text{Scanned Data (Tb)} + ... \\
&\text{S3 Storage Used (Gb)} * \text{S3 Cost for Storage Bracket}
\end{aligned}$$

[15] https://blog.powerupcloud.com/1-1-billion-new-york-taxi-rides-on-redshift-spectrum-and-athena-1d6cfc89a812

5 Explorng Use Cases

5.1 Data Exploring Scenario

"You have the availability of 16 TB of data in S3, in a variety of formats (csv, json, text files, sql dumps). You want to occasionally perform several ad-hoc queries per day. Multiple queries are often, but not always, executed in narrow sessions. Each session typically refers to only about 1% of the overall dataset, but different queries in the same session may touch part of the same data multiple times."

Proposed Solution: This scenario requires a tool that handles large amounts of unstructured data in various formats. It also requires the support of long running jobs (resilience) and caching (same parts of data being queried). The idea solution for this situation would be a big data compute engine like **Apache Spark**. It is an ideal solution because it supports out of the box reading of many popular unsturcutred data formats and allows users to also pragmatically implement methods to read exotic unstructured data sources. Apache Spark also allows for long running resilient jobs to be run that can self recover in events where the job fails. Spark also supports the use of cacheing which will allow for quick repeated runs on the same sets of data during exploritory sessions.

Athena also supports reading unstructured data and could be used in this scenario. The main goal is to perform data exploration, therefore requires queries to be run multiple times and usually on the same data. In general, data exploration should not be governed by the overall cost of running the queries. Furthermore, since Athena does not use caching, data will have to be read over and over again and thus increase the overall cost. But if we know exactly what kind of data exploration we want to conduct and in which dataset, then Athena is a better choice with regards to cost.

5.2 Business Reports Scenario

"Your data set is organised in a star schema. It has one fact table, which grows of about 2GB of new homogenous data every day. More dimension tables are also present, their size is much smaller than the fact table and their content seldom changes. You want to perform several queries per hour. Each query may touch several gigabytes. About 90% of the overall data scanned in the main 'fact' table is from the last month."

Proposed Solution: This use case indicates that we are dealing with a situation that requires a transnational data warehouse. This assessment is based on the fact that the data must be highly structured (star configuration), it receives data insertions at consistent rates and a large portion of the data is scanned on a monthly basis. Out of the three solutions discussed in this report, **AWS Redshift** would be the best solution. It is the ideal choice because it is a fully managed database system in the cloud, ACID compliant and it is highly scalable and is one of the best performing data wearhousing solutions on the market.

Spark would not be an ideal choice because it is not ACID complaint nor can it enforce structured relationships between datasets/databases. Athena is also not ideal because it becomes prohibitively expensive when uses start to query/scan large amounts of data.

5.3 Organisation Scenario

"About 100 users in an organisation share the same data set of about 32 TB. The users run, on average, 8 analytical ad-hoc queries per day, scanning roughly 40GB of compressed data per query. Typically they cumulatively refer to only 4 TB of this dataset in a month, and even inside this subset, the data referred is highly skewed. Currently the whole data set is hosted in an in-house solution. Their system suffers of temporary slow-downs when multiple users operate concurrently, degrading the whole experience. The organisation is wondering whether it is suitable for them to migrate their in-house solution to the cloud, provisioning a (tentative) budget cap of $3800 per month."

Proposed Solution: The scenario requires a solution that allows for multiple concurrent runs on a alimited budget. If each query needs to scan 40GBs of data then using per-query charging services like Athena or Redshift Spectrum would lead to enormous cost very fast. For this reason, we purpose a Redshift + Redshift Spectrum solution. The 4Tb of offten accessed data would be held within the Redshift system and any edge case queries will be run using spectrum on the remaining archived data on S3.

The following calculations show how using redshift for 24/7 in a month is cheaper then using Anthena.
1) If we ignore the edge case spectrum queries then the cost is:

$$\begin{aligned}
&2 * \text{Cost of ds2.xlarge} * (\text{Hours in Month}) + \text{S3 Storage on Archived Data} \\
&= 2 * \$0.85 * (24 * 30) + 28Tb * (1024) * \$0.023 \\
&= \$1883.45
\end{aligned}$$

2) For comparison if a pure Athena solution was used it would cost:

$$\begin{aligned}
&\$5 * \text{Scanned Data (Tb) by Athena} + \$0.023 * \text{S3 Storage Used} \\
&= \$5 * (\text{Number of Employees} * \text{Number of Daily Queries} * ... \\
&... * \text{Number of Working Days} * \text{Data Query Scan (Tb)}) + ... \\
&... + \$0.023 * \text{S3 Storage Used} \\
&= \$5 * (100 * 8 * (40/1024)) + \$0.023 * (32 * 1024) \\
&= \$3878.66
\end{aligned}$$

References

A Code for Converting CSV data to Parquet

Listing 5: Code used to convert csv data to parquet.

```
def write_parquet(dir):
    customer_s = '''
    C_CustKey INT ,
    C_Name STRING ,
    C_Address STRING ,
    C_NationKey INT ,
    C_Phone STRING ,
    C_AcctBal DECIMAL,
    C_MktSegment STRING ,
    C_Comment STRING,
    skip STRING
    '''

    lineitem_s = '''
    L_OrderKey INT ,
    L_PartKey INT ,
    L_SuppKey INT ,
    L_LineNumber INT ,
    L_Quantity INT ,
    L_ExtendedPrice DECIMAL,
    L_Discount DECIMAL,
    L_Tax DECIMAL,
    L_ReturnFlag STRING ,
    L_LineStatus STRING ,
    L_ShipDate DATE,
    L_CommitDate DATE,
    L_ReceiptDate DATE,
    L_ShipInstruct STRING ,
    L_ShipMode STRING,
    L_Comment STRING,
    skip STRING
    '''

    nation_s='''
    N_NationKey INT ,
    N_Name STRING ,
    N_RegionKey INT ,
    N_Comment STRING,
    skip STRING
    '''


    order_s = '''
    O_OrderKey INT,
    O_CustKey INT,
    O_OrderStatus STRING,
    O_TotalPrice DECIMAL,
    O_OrderDate DATE,
    O_OrderPriority STRING,
    O_Clerk STRING,
    O_ShipPriority INT,
    O_Comment STRING,
    skip STRING
    '''

    part_s = '''
    P_PartKey INT,
    P_Name STRING,
    P_Mfgr STRING ,
    P_Brand STRING ,
    P_Type STRING ,
    P_Size INT ,
    P_Container STRING ,
        P_RetailPrice DECIMAL,
        P_Comment STRING ,
        skip STRING
        '''

        parts_s = '''
        PS_PartKey INT ,
        PS_SuppKey INT ,
        PS_AvailQty INT ,
        PS_SupplyCost DECIMAL,
        PS_Comment STRING,
        skip STRING
        '''

        region_s = '''
        R_RegionKey INT ,
        R_Name STRING ,
        R_Comment STRING,
        skip STRING
        '''

        supply_s = '''
        S_SuppKey INT ,
        S_Name STRING ,
        S_Address STRING ,
        S_NationKey INT ,
        S_Phone STRING,
        S_AcctBal DECIMAL,
        S_Comment STRING,
        skip STRING
        '''

        customer = spark.read.csv(dir + "customer", sep='|',schema=customer_s)
        lineitem = spark.read.csv(dir + "lineitem", sep='|',schema=lineitem_s)
        nation = spark.read.csv(dir + "nation", sep='|',schema=nation_s)
        orders = spark.read.csv(dir + "orders", sep='|',schema=order_s)
        part = spark.read.csv(dir + "part", sep='|',schema=part_s)
        partsupp = spark.read.csv(dir + "partsupp", sep='|',schema=parts_s)
        region = spark.read.csv(dir + "region", sep='|',schema=region_s)
        supplier = spark.read.csv(dir + "supplier", sep='|', schema=supply_s)

        customer.write.parquet("s3://group10-lab5/customer")
        customer.write.parquet("s3://group10-lab5/lineitem")
        nation.write.parquet("s3://group10-lab5/nation")
        orders.write.parquet("s3://group10-lab5/orders")
        part.write.parquet("s3://group10-lab5/part")
        partsupp.write.parquet("s3://group10-lab5/partsupp")
        region.write.parquet("s3://group10-lab5/region")
        supplier.write.parquet("s3://group10-lab5/supplier")

    write_parquet(dir = 's3://bigdatacourse2019/tpch_csv/SF10/')
```