DATA 516 Final Project – Part I
Amazon Redshift Timing Exercises

Rex Thompson
rext@uw.edu

DATA 516, Autumn 2017
University of Washington
12/12/2017

Abstract - **We analyze Amazon Redshift query execution and parallel load performance by timing different tasks on several Redshift cluster implementations. We expand upon prior work in which we explored basic Redshift functionality by running six simple queries on a two-node cluster using databases of three different sizes, and running the same queries on clusters of five different sizes while holding database size fixed. We execute the same six queries and three modifications of each to test differences in timing to complete query execution vs. data display in SQL Workbench/J. We show that the process of loading the results to the screen in SQL Workbench/J may take up to 34% or more of the total query execution time, especially for queries that return the results of large joins. We also use an unexpected observation to show that Redshift performs at least low-level optimization. We then split a 7.8 GB data table into 32 separate files and upload it to clusters of 14 different sizes to test Redshift parallel load timing. We show that Redshift load times are generally a function of the number of file partitions compared to the number of node slices.**

1. INTRODUCTION

Amazon Redshift is a cloud-based petabyte-scale data warehouse solution that offers fast and simple SQL-compliant massively parallel processing (MPP). Added to the Amazon Web Services (AWS) suite of services in early 2013, Redshift boasts competitive pricing, remarkably fast spin-up time, and incredible ease of use and reliability. These factors have made it a strong competitor to traditional in-house data warehouse systems, and as a result, Redshift was AWS's fastest growing service for several years [1] until being overtaken by the new Aurora service in late 2015 [2].

In a prior work, we recorded the execution time for six simple queries on a two-node Redshift cluster using TPC-H databases of three different sizes. We also timed the same six queries on clusters of five different sizes while holding database size fixed. These two experiments allowed us to explore Redshift's basic functionality and audit query timings against our expectations for the MPP system. A summary of our prior work and its results are presented in Section 2 below.

In this paper, we explore different components of Amazon Redshift's performance by answering the following two research questions which came out of our prior work:

RQ1. **Query Execution vs. Data Display:** How much of a query's execution time is simply due to the burden of displaying the query's results to the screen?
RQ2. **Parallel Loading:** How do file partitions and node slice counts interact to affect Redshift load times?

In Section 3 we describe the ecosystem, data, and queries used throughout this study. We summarize our experiment setup and findings for each of the two research questions presented above in Sections 4 and 5, respectively. Finally, we summarize our conclusions in Section 6.

2. PRIOR WORK

In our prior work, we performed two basic experiments to evaluate Redshift's performance against the behavior we expected from such a MPP system.

First, we ran six simple queries on a two-node dc2.xlarge cluster using TPC-H databases sizes of 1, 10 and 100 GB. The queries are shown in Table 1 below:

#	Query
1	SELECT S.S_Name, COUNT(*) FROM supplier S LEFT JOIN partsupp PS ON S.S_SuppKey = PS.PS_SuppKey GROUP BY S.S_Name;
2	SELECT S.S_Name, MAX(PS.PS_SupplyCost) FROM supplier S LEFT JOIN partsupp PS ON S.S_SuppKey = PS.PS_SuppKey GROUP BY S.S_Name;
3	SELECT MAX(PS_SupplyCost) FROM partsupp;
4	SELECT N.N_Name, COUNT(*) FROM nation N LEFT JOIN customer C ON N.N_NationKey = C.C_NationKey GROUP BY N.N_Name;
5	SELECT S.S_Name, SUM(L.L_Quantity) FROM supplier S LEFT JOIN lineitem L ON S.S_SuppKey = L.L_SuppKey WHERE L.L_ShipDate BETWEEN '10/10/1996' AND '11/10/1996' GROUP BY S.S_Name;
6	SELECT AVG(DATEDIFF(day, O.O_OrderDate, L.L_ShipDate)) FROM orders O, lineitem L

Table 1: Basic queries used in Prior Work and Research Question 1

The point of this experiment was to explore how execution times changed for different queries when run on databases of varying sizes. Specifically, *does execution time increase linearly with database size?* Figure 1 at right displays the timing results for this experiment. As expected, query execution time increased with increasing database size for all six queries, and was generally linear for most of the queries. The two queries that did not seem to follow the trend were Queries 3 and 4. As shown in Table 1 above, Query 3 was the simplest of the queries, while one of the tables joined in Query 4 was quite small (only 25 rows) and was the same size across all three databases.

We then ran the same six queries on the TPC-H 100 GB dataset with 2-, 4-, 8-, 12- and 16-node dc2.xlarge clusters to evaluate speed-up in Redshift. Figure 2 at right displays the timing results for this experiment. As shown, more nodes resulted in faster execution times, until a certain point at which the execution times leveled out or even began to increase with additional nodes. This demonstrated the fact that adding more nodes does not always translate to faster execution times: there is a point at which the system becomes saturated and the overhead of communicating and transferring data to/from the many nodes begins to dominate the execution time. Again, Queries 3 and 4 stood out on the plot as the fastest and most consistent, likely for the same reasons as described above. Query 6 also stood out as the only one for which execution time did not appear to be leveling out at 16 nodes. As shown in Table 1 above, this is the query that utilized the DATEDIFF function, so presumably adding additional nodes allowed for faster parallelization of this expensive computation.

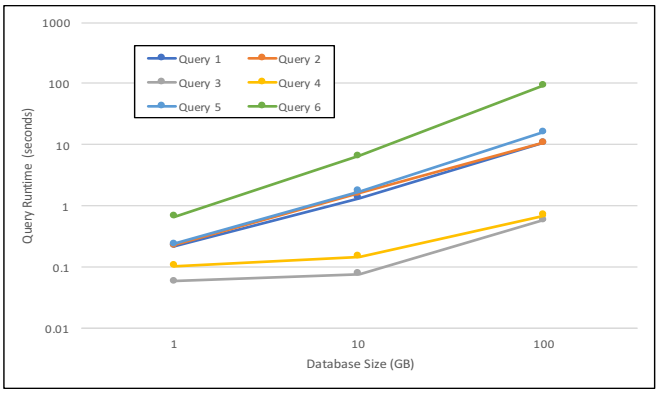

Figure 1: Query Runtime vs. Database Size, 2-node cluster

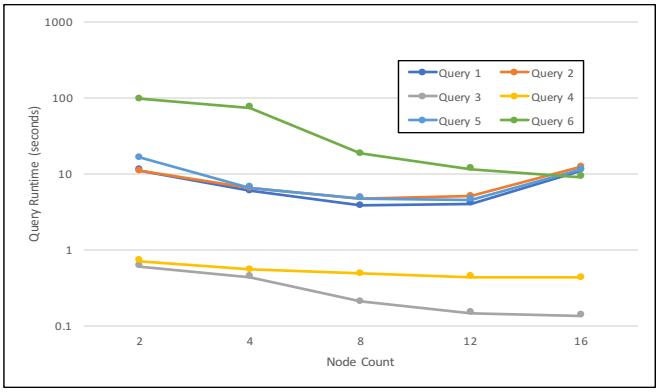

Figure 2: Query Runtime vs. Cluster Size, 100 GB database

3. Experiment Ecosystem, Data and Queries

For our research questions in this work, we used Amazon Redshift clusters of type dc1.large and dc2.large, accessed via SQL Workbench/J software (Build 123) and a JDBC connection (JDBC 4.2-compatible driver, Java version 1.8.0_151). All work was performed on a 2015 MacBook Pro running MacOS Sierra (Version 10.12.6).

We used database sizes of 1, 10 and 100 GB from The Transaction Processing Performance Council's (TPC) TPC-H dataset (the same data as was used in the prior work). The data was stored on Amazon's Simple Storage Service (S3).

Node type/size and database size varied between experiments. Details of each experiment's setup are described within the relevant sections below.

For Research Question 1 we used the same six basic queries as we did in our prior work. As shown in Table 1 above, the simplest of the queries consisted of a simple aggregation (i.e. MAX) of a single column from just one table; three of the queries consisted of a simple LEFT JOIN of two tables, with a GROUP-BY projection and a single aggregation returned for each (i.e. COUNT or MAX); one of the queries consisted of a slightly more complicated two-table LEFT JOIN / GROUP-BY / projection / aggregation, with an additional BETWEEN clause limiting the results prior to the GROUP-BY; and the last query consisted of a two-table equi-join, with the final result representing the mean of the output determined from a built-in function (i.e. DATEDIFF). Again, refer to Table 1 above for the complete queries.

We used three slightly modified versions of each of the six queries described above for Research Question 1. These queries are described in more detail in Section 4 below.

4. RQ1: Query Execution vs. Data Display

I. Background

In our prior work, we recorded all query timings from the status bar at the bottom of the SQL Workbench/J software, which is equivalent to the query's "Execution time" as reported in the Message window. However, towards the end of our experiments we noticed that partway through processing some of the queries, the SQL Workbench/J status indicator would switch from "Executing statement…" to "Loading row x", where the value of "x" would quickly progress through all the row numbers being loaded until all results had been loaded, at which point the results were shown in the Results window and the final "Execution time" was displayed in the status bar and the Message window.

This loading process was hardly noticeable for results with a small number of rows, but it took up to a few seconds for queries that returned large results, such as the large joins on the 100 GB dataset. Importantly, it appeared this loading time was being included in the "Execution time" as reported in the status bar and the Message window. We also did not notice a decrease in load time with increasing cluster size. As

a result, we suspected that the load process might be inherent to SQL Workbench/J itself, and thus, independent of actual Redshift performance. If true, we worried this could be skewing our results – especially those in which we used the 100 GB data to test the speed with which Redshift executed queries on clusters of varying sizes.

This led us to ask the following question: **How much of a query's execution time is simply due to the burden of displaying the query's results to the screen?**

II. Experiment Setup

To answer this question, we devised an experiment to help understand the actual query execution time in Redshift compared to the time required to load the query results to the screen in SQL Workbench/J. We evaluated the difference in total "Execution time" (i.e. query execution time plus the time to load the results to the screen) between the original queries and the same queries with the following three clauses added (one at a time):

- **SELECT TOP 1**
- **SELECT COUNT(*)**
- **SELECT INTO**

For example, Table 2 below presents Query 1 in its original form and with the three modifications. The changes from the original query are shown in red for each of the three modified queries. The other five queries were modified in the same way as shown for Query 1 below.

Version	Query
Original	SELECT S.S_Name, COUNT(*) FROM supplier S LEFT JOIN partsupp PS ON S.S_SuppKey = PS.PS_SuppKey GROUP BY S.S_Name;
SELECT TOP 1	SELECT **TOP 1** S.S_Name, COUNT(*) FROM supplier S LEFT JOIN partsupp PS ON S.S_SuppKey = PS.PS_SuppKey GROUP BY S.S_Name;
SELECT COUNT(*)	**SELECT COUNT(*)** **FROM** **(**SELECT S.S_Name, COUNT(*) FROM supplier S LEFT JOIN partsupp PS ON S.S_SuppKey = PS.PS_SuppKey GROUP BY S.S_Name**)**;
SELECT INTO	SELECT S.S_Name, COUNT(*) **INTO NewTable1** FROM supplier S LEFT JOIN partsupp PS ON S.S_SuppKey = PS.PS_SuppKey GROUP BY S.S_Name;

Table 2: Example original query and three modifications - Query 1

SELECT TOP 1 limits the output to the first result. This allows execution to cease as soon as the first result row is loaded, such that the "Execution time" effectively represents the time to evaluate the query in Redshift alone, with negligible time spent on displaying the (single) result.

SELECT COUNT(*) again limits the output to a single row to effectively eliminate the load time, but also includes a count clause to ensure that the entire query completes in Redshift before the process is aborted.

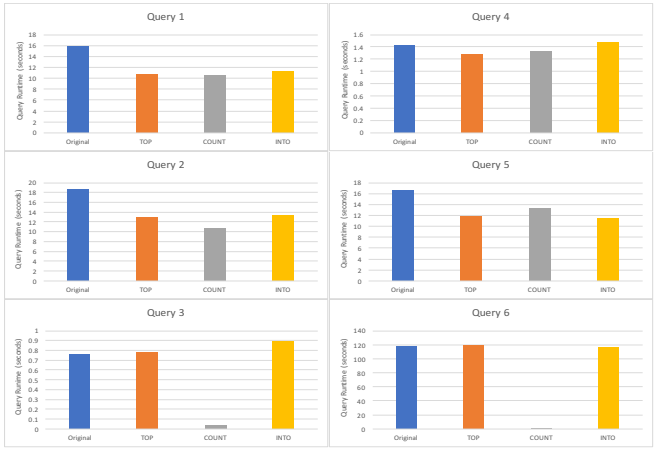

Figure 3: RQ1 Average warm-cache query execution times

SELECT INTO again requires that the original query be fully executed in Redshift, but eliminates the time to load the results to the screen altogether by instead saving the returned result to a new relation within Redshift.

We performed this experiment on a two-node dc1.large cluster with the 100 GB TCP-H dataset. We ran each query in each configuration six times: once to fill the cache, and five additional times to get average warm-cache timing.

III. Results

The average warm-cache query execution runtimes for the four combinations of the six queries are displayed in Figure 3 above. As shown, the query modifications generally improved performance, with the only major exceptions being queries with execution times under two seconds in duration.

The queries with the greatest general improvements were those which consisted of the largest merges (i.e. Queries 1, 2 and 5). These are also the queries which returned the largest relations, as shown in Table 3 below.

Query	Rows in Result
1	1,000,000
2	1,000,000
3	1
4	25
5	999,634
6	1

Table 3: Cardinality of query results, 100 GB dataset

If we take the base Redshift query execution time to be the mean execution time for the three modified queries, then we can calculate that the SQL Workbench/J load time accounts for approximately one quarter to one third of the total execution time as displayed in the status bar for the original queries. For example, the three modified queries for Query 1 yielded a mean execution time of 10.96 seconds, while the original query yielded a mean execution of time of 15.89 seconds. This suggests that only ~69% of the original query execution time for Query 1 went towards actually executing the query in Redshift, while the additional 31% went towards loading the results to the screen in SQL

Workbench/J. This same calculation for Queries 2 and 5 yields load proportions of 34% and 26%, respectively.

This confirmed our suspicion that SQL Workbench/J "Execution time" may not be representative of the time it takes Redshift to complete the queries, as a significant portion of this time may consist of the time it takes to load the results to the screen. In this case we saw load proportions up to ~34%, but presumably even higher values could be produced, such as from a merge or even a simple SELECT * from an even larger relation than we used here.

As an aside, Queries 3 and 6 exhibited an unusual pattern that is worth pointing out. In both cases, the SELECT TOP 1 and SELECT INTO modifications resulted in little if any improvement, but the SELECT COUNT(*) modifications resulted in drastic improvement, with both queries returning results in an average of just 0.04 seconds! Recall that these two original queries both returned a single result: a MAX and an AVG, respectively. Thus, it appears that Redshift was smart enough to recognize the fact that we were asking for the COUNT of a query that would return only a single result, so it apparently skipped the entire query execution process altogether and simply output a result of '1'. A review of the query plan for these queries confirms this fact:

```
XN Aggregate (cost=0.03..0.03 rows=1 width=0)
    ->  XN Network (cost=0.00..0.02 rows=1 width=0)
        Distribute Round Robin
        -> XN Subquery Scan derived_table1
            (cost=0.00..0.02 rows=1 width=0)
        -> XN Result
            (cost=0.00..0.01 rows=1 width=0)
```

As shown, the query plan indicates that the row count was identified at all levels; thus, the final query to return the count of 1 could be executed very quickly, without actually having to execute the full underlying query.

5. RQ2: PARALLEL LOADING

I. Background

When first introduced to Amazon Redshift, we were under the impression that data import could be sped up by loading in parallel if multiple nodes were used. Thus, we were surprised when load times did not significantly improve as we increased node count from 2 to 16 in our prior work. In fact, we did not really see any improvement at all with larger clusters!

Upon further investigation, we learned that in order to take advantage of parallel data loading in Redshift, the source data file needs to be split into more than one file, and for best performance Amazon recommends splitting a source file into a multiple of the number of node slices available (each node consists of a set number of slices) [3].

We decided to split a single table file into several pieces and experiment with ingesting it onto clusters of various sizes to answer the following question: **How do file partitions and node slice counts interact to affect Redshift load times?**

II. Experiment Setup

We used the AWS Command Line Interface (CLI) to copy the 10 GB TPC-H database 'lineitem' table from S3 to our local machine. The file was 7.78 GB in size, with 59,986,052 lines of data. We used the 'split' command in the bash shell to break the file into 32 pieces of 1,874,565 lines each, except for the last file which had only 1,874,537 lines. Most files came out to be ~243 MB in size; the smallest was 240.3 MB, and the largest was 243.4 MB. We then used the AWS CLI to upload the 32 split files back to S3. We spun up dc2.large clusters of 14 different sizes, with node counts of 1, 2, 3, 4, 6, 8, 10, 12, 14, 16, 18, 20, 26 and 32. Each of the dc2.large nodes contained two slices, so our node slice count was double the values listed above.

For each of the clusters, we created an empty table called 'lineitem' and issued a command through SQL Workbench/J to populate the table with data from the S3 bucket that contained the 'lineitem' table data in 32 separate files. We recorded the time it took to copy the data into the table; times were retrieved from the AWS Redshift Clusters Queries page, as opposed to the SQL Workbench/J status bar: we did not want our results to be affected by any ancillary processes that may be included in the SQL Workbench/J status bar "Execution time", as observed in our previous exercise.

We would have liked to perform multiple trials for each configuration; however, due to the expense of the clusters by node-hour, we opted to perform each load only once.

Based on what we had read about parallel file loading in Redshift, we were particularly interested to investigate load times compared to the number of "load rounds" we assumed would take place for each given number of node slices. For example, in our first trial, we loaded 32 files to only 2 node slices. We assumed that the two node slices would process a combined two files at a time, starting with the first two files, before moving on to the next two files, and so on until all 32 files had been loaded. Thus, we assumed it would take 16 rounds of loading two files at a time to import the entire data table. If Redshift worked the way we thought it did, we expected to see load times that correlated well with the number of load rounds for clusters of various sizes, given our 32 data files.

III. Results

The results of our data load timing experiments are shown in Figure 4 below. The orange bars represent the number of assumed load rounds for the given number of file partitions and node slices (axis on left). The blue bars represent the load times, and the gray dots represent the load times divided by the assumed number of load rounds (axes for both on the right).

We see generally decreasing load times, which is what we would expect for adding additional nodes. And, as anticipated, the load times appear to follow the assumed load rounds remarkably closely, even when the assumed load rounds don't change from cluster to cluster. For example, note that the timings roughly track the assumed load rounds even when the latter remains fixed at two from the sixth

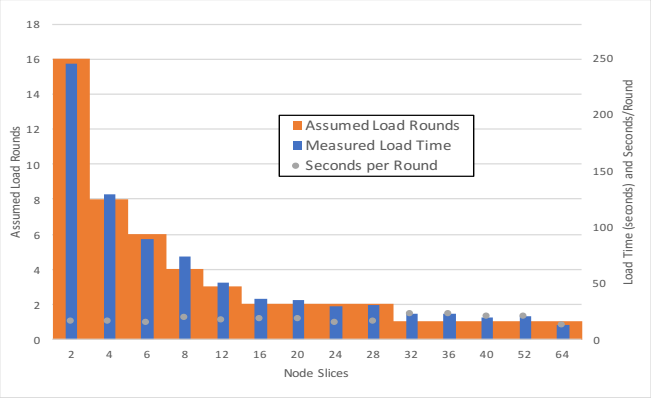

Figure 4: Rounds (assumed) and times to load a 32-piece file

through ninth clusters (i.e. 16-28 node slices), and when the rounds remain fixed at one from the 10th through the 12th clusters (i.e. 32-52 slices). The final data point appears to deviate from this trend slightly; we will address this below.

These findings validate our understanding of Redshift's parallel loading functionality. Specifically, it appears files are assigned to a specific node and, due to their similar size, are effectively loaded in rounds. The number of rounds seems to depend on the number of file partitions as well as the number of node slices. If the number of file partitions is a multiple of the number of node slices, then the files are evenly distributed across the cluster, and all node slices finish importing their assigned data partition at approximately the same time after a set number of load rounds. However, if the number of file partitions is not a multiple of the number of node slices, then the cluster proceeds as before, but some "leftover" files have to be evaluated at the end, resulting in an extra load round. For example, with 32 files and 16 node slices, all 16 node slices process two files each and the nodes all finish their second load at approximately the same time. With 20 node slices, the first 20 files are loaded in a single round by the 20 node slices; however, there are still 12 files that need to be loaded. The remaining files are presumably then assigned to 12 of the 20 node slices for a second round of loading. Thus, increasing the node slice count will not necessarily speed up the load time if the number of load rounds does not change.

As mentioned above, the final data point (64 node slices) appears to deviate from this logic slightly. We had assumed that increasing the node slice count from 32 to 64 would not speed up the load process, since all 32 files would still be loaded in a single round by only half of the available node slices. However, as shown in Figure 4 above, the load time for 64 node slices was 12.5 seconds, compared to an average of ~21 seconds for the prior four one-round load processes (i.e. 32-52 node slices). This inconsistency is also reflected in the seconds/round value, which is a significant outlier compared to the same measurement for all other node sizes in this experiment. Our assumption is that the increase in load speed with 64 node slices has something to do with the redundant backups that Redshift automatically creates in the background. However, more reading and experimentation would be needed to explore this concept further.

6. CONCLUSIONS

We explored the scalability of Amazon Redshift's MPP in our prior work by experimenting with runtimes of queries on different-sized databases and clusters. We showed that Redshift provides near-linear scale-up when increasing from 1 to 10 to 100 GB databases. We also showed that increasing cluster size does not always translate to faster performance; in fact, in some cases, adding nodes may actually slow the query execution process.

In Research Question 1, we showed that the "Execution time" reported in the SQL Workbench/J status bar may include ancillary processes such as loading the results to display on screen; such processes are overhead to the query execution job and should not be taken into account when evaluating the scalability of Redshift itself. This finding may invalidate some of the results from our prior work. We also showed that Redshift appears to have at least some form of built-in query optimization, as demonstrated by the query plan and the observed speed with which the SELECT COUNT(*) queries returned results for the subqueries that produced only one row (i.e. MAX or AVG).

Finally, we demonstrated the relationship between node slice count and query load times when importing data in parallel. We showed that increasing cluster size does not necessarily translate to faster load time; instead, node slice count should be a multiple of file partitions, as suggested by Amazon in their Redshift documentation [3].

7. ACKNOWLEDGMENTS

We thank Parmita Mehta for providing a copy of the TPC-H data used in this study.

8. REFERENCES

[1] Anurag Gupta, Deepak Agarwal, Derek Tan, Jakub Kulesza, Rahul Pathak, Stefano Stefani, and Vidhya Srinivasan. Amazon Redshift and the Case for Simpler Data Warehouses. In Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data (SIGMOD '15), 2015. Pages 1917-1923.

[2] *New – Encryption at Rest for Amazon Aurora*, https://aws.amazon.com/blogs/aws/new-encryption-at-rest-for-amazon-aurora/

[3] *Using the COPY Command to Load from Amazon S3*, docs.aws.amazon.com/redshift/latest/dg/t_loading-tables-from-s3.html