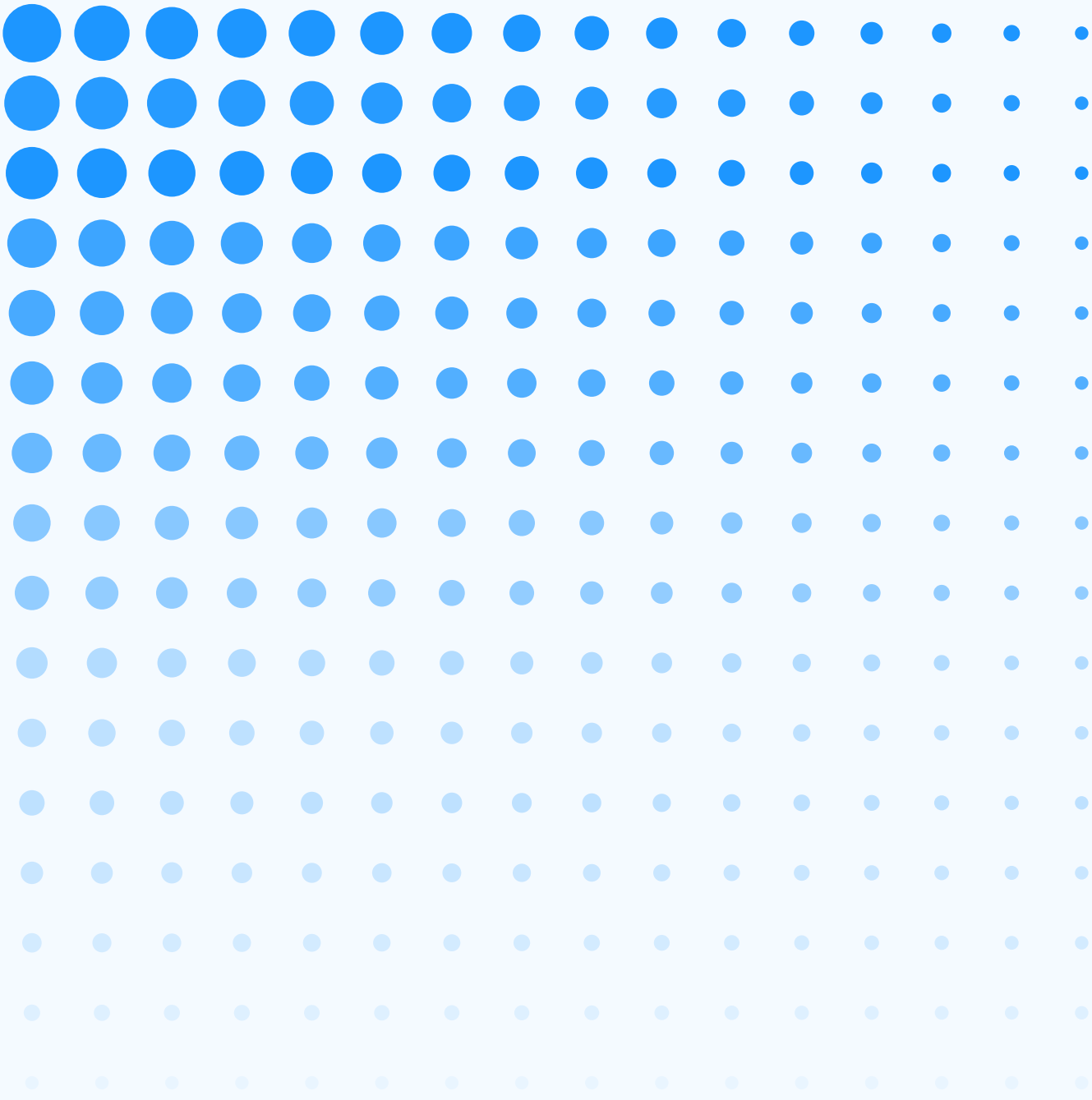
Top 14 Performance Tuning Techniques for Amazon Redshift



integrate.io

Table Of Contents

01

INTRODUCTION

Amazon Redshift is a data warehouse that makes it fast, simple and cost-effective to analyze petabytes of data across your data warehouse and data lake.

Redshift can deliver 10x the performance of other data warehouses by using a combination of machine learning, massively parallel processing (MPP), and columnar storage on SSD disks.

But even with all that power, it's possible that you'll see uneven query performance, or challenges in scaling workloads. Performance optimization for Amazon Redshift is a matter of doing some thoughtful up-front planning and ongoing monitoring as your data volume, users and cluster grow.

Running a Cluster That's Fast, Cheap and Easy to Scale

In this post, we're describing 14 best practices for performance tuning for Amazon Redshift. If you follow these practices, you'll have a cluster that is faster, cheaper and easier to scale than any other product on the markets.

For data teams in charge of managing an Amazon Redshift cluster, using these best practices will help them be successful in building complex data pipelines.

How We Use Amazon Redshift

We know this from experience. Here at Integrate.io, we use Amazon Redshift as part of our core platform. This blog post compiles learnings we've made over three years of operating several, very large Redshift clusters at high scale.

Integrate.io is an analytics platform that provides a single monitoring dashboard for data engineers to keep an eye on their mission-critical data flows.

Integrate.io uses Amazon Redshift for batch processing large volumes of data in near real-time. Our data pipeline processes over 20 billion rows per day. We serve data from Amazon Redshift to our application by moving it into RDS (via DBLINK) and Amazon Elasticsearch Service. And goes without saying that we're drinking our own Champaign – we use Integrate.io to monitor Integrate.io

02

CREATE CUSTOM WORKLOAD MANAGER (WLM) QUEUES

The Amazon Redshift Workload Manager (WLM) is critical to managing query performance.

Amazon Redshift run queries in a queueing model. The default WLM configuration has a single queue with five slots. 99% of the time this default configuration will not work for you and you will need to tweak it. Configuring the WLM for your workloads provides two main benefits:

1. Scaling workloads by giving them enough resources (e.g. concurrenc and memory)
2. Isolating and protecting your predictable workloads (i.e. batch operations) from your unpredictable workloads (i.e. ad hoc queries from reporting tools)

You can have up to 8 queues with a total of up to 50 slots. A query will run in a single slot, by default. Queries can be routed into queues using certain rules. Setting up your WLM the right way will eliminate queue wait times and disk-based queries.

To set-up your WLM for your workloads, we recommend following a four-step process:

1. Separate users
2. Define workloads
3. Group users into workloads
4. Select slot count & memory % per queue

We've written a detailed guide on tuning WLM that describes the four-steps. The approach will help you eliminate queues wait times and reduce disk-based queries. Both will slow your cluster down, so let's take a closer look.

Eliminate queue wait times by matching queue slot count to peak concurrency

If you've used Redshift for any period of time, you may have come across a situation where a query that used to run for two seconds, all of the sudden starts running much, much slower. The most common reason for this is queueing. The query was waiting in a queue because the # of slots in the cluster was too low for the number of concurrent of queries executing.

The default configuration allows you to run five concurrent queries in one queue. That means if five queries are executing, the sixth one will queue until a slot becomes available.

The goal is to ensure that queries are not waiting in the queue. This can be done by matching the slot count of the queue with the actual concurrency of the queries running in that queue.

You can eliminate queue wait times by:

1. Increasing the slot count for you queues
2. Reducing concurrency by distributing queries more evenly throughout the day.

Reduce disk-based queries by assigning enough memory to your queues

Increasing slot count to eliminate queuing can have an adverse side effect: Diskbased queries. "Disk-based" means that the query runs out of RAM, and begins using the hard drive. Queries go disk-based because the query memory exceeds the 'memory per slot' in that queue. The memory per slot is calculated as: *memory assigned to that queue / # of slots*

Since each queue is assigned a fixed percentage of a cluster's memory (a value you'll set when you configure your WLM queue), adding more slots will decrease the memory per slot.Disk-based queries cause two major problems:

1. Queries slow down because they need more I/O
2. Concurrency goes up which causes more queueing.

When the frequency of disk-based queries goes up, a chain reaction can occur. More I/O causes more CPU, which in turn make queries run slower, increasing concurrency overall. As a rule of thumb, maintain your queues so that on average fewer than 10% of queries go disk-based.

03

USE CHANGE DATA CAPTURE (CDC)

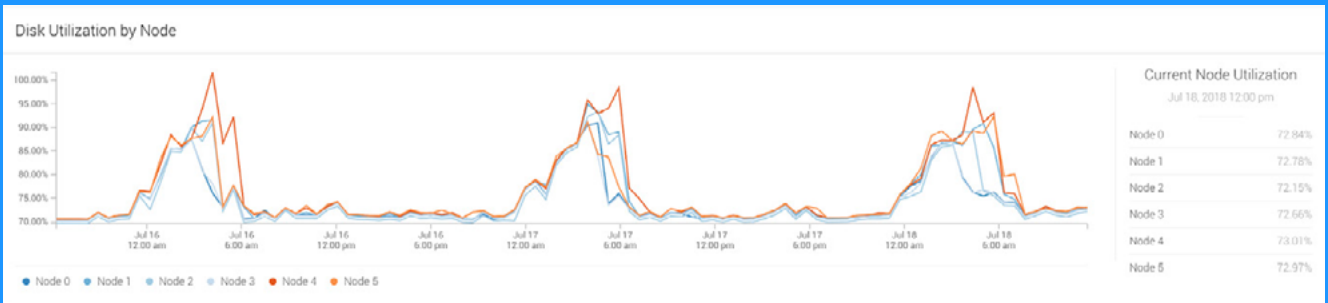
The Amazon Redshift COPY command takes advantage of the parallel architecture and is the recommended way of moving data into Redshift.

The COPY command is optimized, but the COPY operation is still expensive. The best practice is to only copy in rows that are needed.

The goal is to minimize the number of rows ingested. The best way to do this is to ensure that your ETL tools are only COPYing in data that has changed since the last time.

Otherwise, you will have two issues:

- Frequent spikes in disk utilization which will require to keep more free capacity
- Deleting redundant data (deduplication) uses I/O and increases the need to run VACUUM operations.



Spikes in Disk Utilization

CHANGE DATA CAPTURE

Step	Command
Find the next page by querying the destination table	`most_recent_record = SELECT max(time) FROM $destination_table`
Move to S3	`S3.put($file)` `SELECT * FROM $source_table WHERE time > $most_recent_record`
Copy into Redshift (or UPSERT)	`COPY $destination_table ($file)` **`compupdate off statupdate off`**

Here is an example of a CDC operation

04

USE COLUMN ENCODING

Adding compression to large, uncompressed columns will have a big impact on cluster performance.

Compression accomplishes two things:

1. **Reduce storage utilization.** Because file compression reduces the size footprint of data, you'll use less of the disk on your cluster nodes.
2. **Improve query performance.** Because there is less data to scan or join on, I/O usage is limited which increases query speeds.

We recommend using the Zstandard (ZSTD) encoding algorithm. This relatively new algorithm provides a high compression ratio and works across all Amazon Redshift data types. ZSTD is especially good with VARCHAR and CHAR fields that have a mixture of long and short strings. Also, unlike some of the other algorithms, ZSTD is unlikely to increase storage utilization.

Here is a real-world example of applying ZSTD to three Amazon Redshift logging tables. The average storage reduction is over 50%!

	Table	Column	Encoding	Est_reduction_pct
1	stv_tbl_perm	slice	zstd	69.67
2	stv_tbl_perm	id	zstd	94.50
3	stv_tbl_perm	name	zstd	52.82
4	stv_tbl_perm	rows	zstd	29.61
5	stv_tbl_perm	sorted_rows	zstd	49.58
6	stv_tbl_perm	temp	zstd	63.69
7	stv_tbl_perm	db_id	zstd	58.96
8	stv_tbl_perm	insert_pristine	zstd	65.89
9	stv_tbl_perm	delete_pristine	zstd	65.57
10	stv_tbl_perm	backup	zstd	69.99
11	stv_tbl_perm	scantime	zstd	99.83

	Table	Column	Encoding	Est_reduction_pct
1	svl_query_report	userid	zstd	67.42
2	svl_query_report	query	zstd	90.03
3	svl_query_report	slice	zstd	55.80
4	svl_query_report	segment	zstd	52.12
5	svl_query_report	step	zstd	56.22
6	svl_query_report	start_time	zstd	82.93
7	svl_query_report	end_time	zstd	32.75
8	svl_query_report	elapsed_time	zstd	29.13
9	svl_query_report	rows	zstd	44.07
10	svl_query_report	bytes	zstd	40.23
11	svl_query_report	label	zstd	34.23
12	svl_query_report	is_diskbased	zstd	61.76
13	svl_query_report	workmem	zstd	49.35
14	svl_query_report	is_rrscan	zstd	51.09
15	svl_query_report	is_delayed_scan	zstd	91.61
16	svl_query_report	rows_pre_filter	zstd	44.12

	Table	Column	Encoding	Est_reduction_pct
1	stv_blocklist	slice	zstd	63.97
2	stv_blocklist	col	zstd	52.99
3	stv_blocklist	tbl	zstd	90.75
4	stv_blocklist	blocknum	delta	66.84
5	stv_blocklist	scantime	zstd	99.95

05

DON’T ANALYZE ON EVERY COPY

Real-time data is essential for brands within the travel industry.

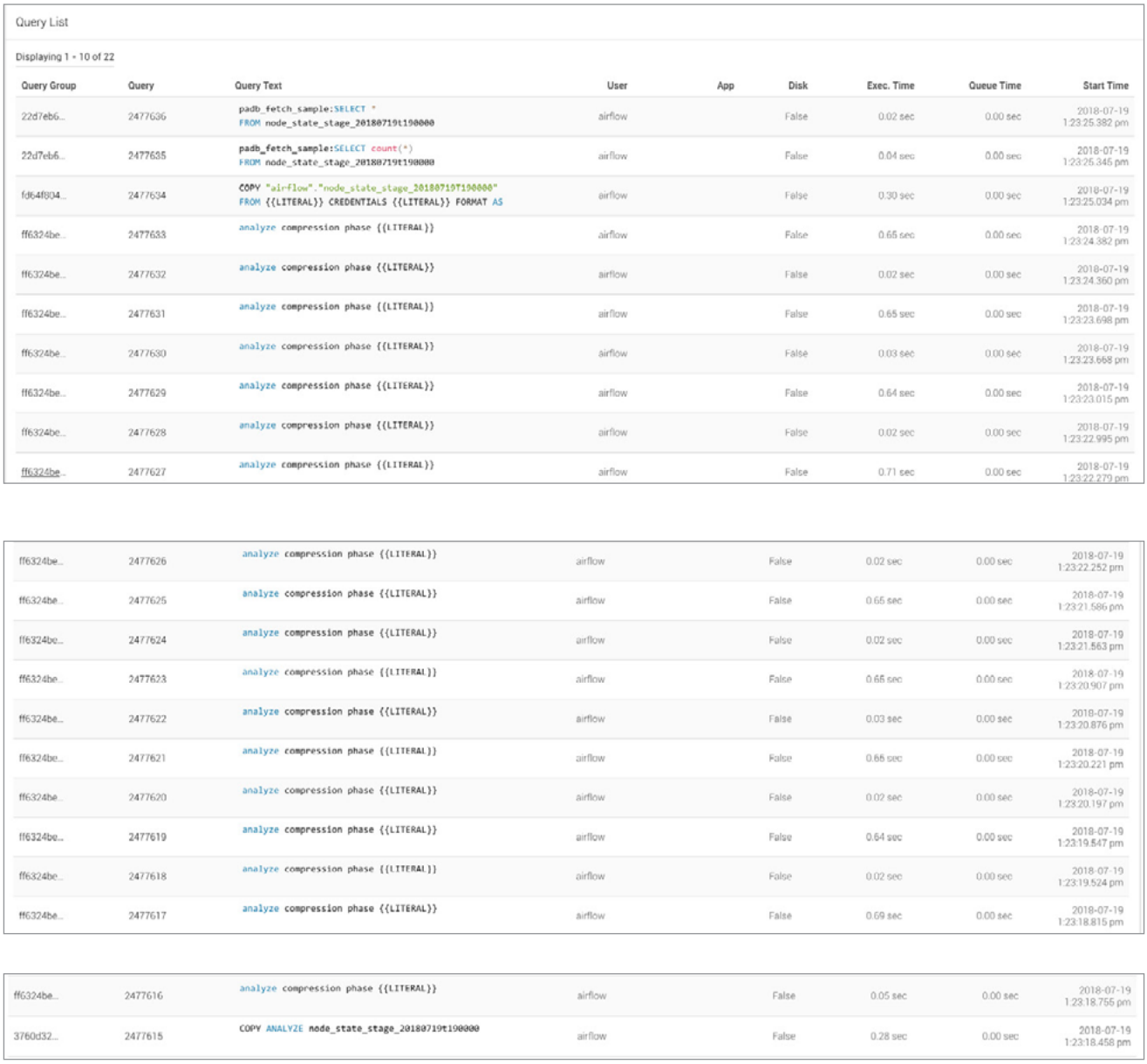
The Amazon Redshift COPY command loads data into a table. The default behavior of Redshift COPY command is to run two commands:

1. “COPY ANALYZE PHASE 1|2” and
2. “COPY ANALYZE $temp_table_name”

Amazon Redshift runs these commands to determine the correct encoding for the data being copied. This may be useful when a table is empty. But in the following two cases the extra queries are useless and thus should be eliminated:

1. When COPYing into a temporary table (i.e. as part of an UPSERT)
2. When the table already has data in it. For an existing table, encoding cannot change. So even if the COPY command determines that a better encoding style exists, it’s impossible to modify the encoding of the table without doing a deep copy operation.

In the example, a single COPY command generates 18 ‘analyze compression’ commands and a single ‘copy analyze’ command.



Query List

Displaying 1 - 10 of 22

Query Group	Query	Query Text	User	App	Disk	Exec. Time	Queue Time	Start Time
22d7eb6...	2477636	padb_fetch_sample:SELECT * FROM node_state_stage_20180719t190000	airflow		False	0.02 sec	0.00 sec	2018-07-19 1:23:25.382 pm
22d7eb6...	2477635	padb_fetch_sample:SELECT count(*) FROM node_state_stage_20180719t190000	airflow		False	0.04 sec	0.00 sec	2018-07-19 1:23:25.345 pm
fd64f804...	2477634	COPY "airflow"."node_state_stage_20180719t190000" FROM {{LITERAL}} CREDENTIALS {{LITERAL}} FORMAT AS	airflow		False	0.30 sec	0.00 sec	2018-07-19 1:23:25.034 pm
ff6324be...	2477633	analyze compression phase {{LITERAL}}	airflow		False	0.65 sec	0.00 sec	2018-07-19 1:23:24.382 pm
ff6324be...	2477632	analyze compression phase {{LITERAL}}	airflow		False	0.02 sec	0.00 sec	2018-07-19 1:23:24.360 pm
ff6324be...	2477631	analyze compression phase {{LITERAL}}	airflow		False	0.65 sec	0.00 sec	2018-07-19 1:23:23.698 pm
ff6324be...	2477630	analyze compression phase {{LITERAL}}	airflow		False	0.03 sec	0.00 sec	2018-07-19 1:23:23.668 pm
ff6324be...	2477629	analyze compression phase {{LITERAL}}	airflow		False	0.64 sec	0.00 sec	2018-07-19 1:23:23.015 pm
ff6324be...	2477628	analyze compression phase {{LITERAL}}	airflow		False	0.02 sec	0.00 sec	2018-07-19 1:23:22.995 pm
ff6324be...	2477627	analyze compression phase {{LITERAL}}	airflow		False	0.71 sec	0.00 sec	2018-07-19 1:23:22.279 pm

ff6324be...	2477626	analyze compression phase {{LITERAL}}	airflow		False	0.02 sec	0.00 sec	2018-07-19 1:23:22.252 pm
ff6324be...	2477625	analyze compression phase {{LITERAL}}	airflow		False	0.65 sec	0.00 sec	2018-07-19 1:23:21.586 pm
ff6324be...	2477624	analyze compression phase {{LITERAL}}	airflow		False	0.02 sec	0.00 sec	2018-07-19 1:23:21.563 pm
ff6324be...	2477623	analyze compression phase {{LITERAL}}	airflow		False	0.65 sec	0.00 sec	2018-07-19 1:23:20.907 pm
ff6324be...	2477622	analyze compression phase {{LITERAL}}	airflow		False	0.03 sec	0.00 sec	2018-07-19 1:23:20.876 pm
ff6324be...	2477621	analyze compression phase {{LITERAL}}	airflow		False	0.65 sec	0.00 sec	2018-07-19 1:23:20.221 pm
ff6324be...	2477620	analyze compression phase {{LITERAL}}	airflow		False	0.02 sec	0.00 sec	2018-07-19 1:23:20.197 pm
ff6324be...	2477619	analyze compression phase {{LITERAL}}	airflow		False	0.64 sec	0.00 sec	2018-07-19 1:23:19.547 pm
ff6324be...	2477618	analyze compression phase {{LITERAL}}	airflow		False	0.02 sec	0.00 sec	2018-07-19 1:23:19.524 pm
ff6324be...	2477617	analyze compression phase {{LITERAL}}	airflow		False	0.69 sec	0.00 sec	2018-07-19 1:23:18.815 pm

ff6324be...	2477616	analyze compression phase {{LITERAL}}	airflow		False	0.05 sec	0.00 sec	2018-07-19 1:23:18.755 pm
3760d32...	2477615	COPY ANALYZE node_state_stage_20180719t190000	airflow		False	0.28 sec	0.00 sec	2018-07-19 1:23:18.458 pm

Extra queries can create performance issues for other queries running on Amazon Redshift. They increase concurrency and hence may saturate the number of slots in a WLM queue, causing other queries to have queue wait times.

The solution is to adjust the COPY command parameters to add “COMPUPDATE OFF” and “STATUPDATE OFF”. These parameters will disable these features during “UPSERT”s.

Here is an example of a “COPY” command with these options set

```
-- Load data into the staging table

COPY users_staging (id, name, city)

FROM ‘s3://.....’

CREDENTIALS ‘aws_access_key_id=xxxxxxx;aws_secret_access_key=xxxxxxx’

COMPUPDATE OFF STATUPDATE OFF;
```

06

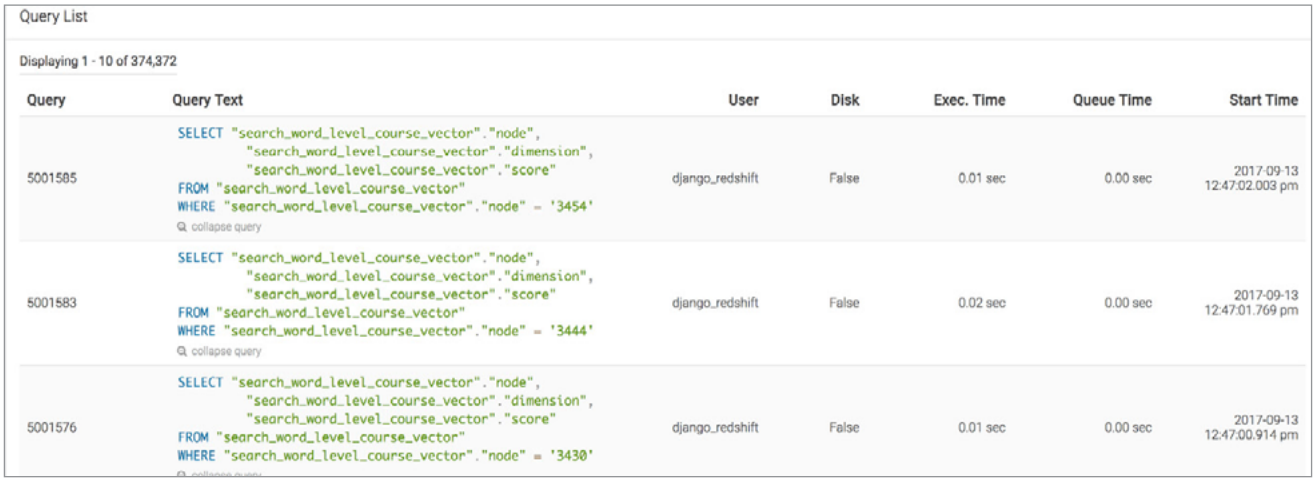
DON’T USE REDSHIFT AS AN OLTP DATABASE

It is common to connect an application framework like Django to Amazon Redshift.

This is useful when using Redshift data in your application, i.e. in an OLTP scenario. Since Amazon Redshift is an OLAP database, it may not handle these queries well.

The challenge of using Redshift as an OLTP database is that queries can lack the low-latency that would exist on a traditional RDBMS and transactional queries. Unlike OLTP databases, OLAP databases do not use an index. This is a result of the column-oriented data storage design of Amazon Redshift which makes the trade-off to perform better for big data analytical workloads.

Consider this example from a live production cluster. The user *'django_redshift'* is querying the table *'search_word_level_course_vector"*, a table with 443,744 rows.



Query List

Displaying 1 - 10 of 374,372

Query	Query Text	User	Disk	Exec. Time	Queue Time	Start Time
5001585	SELECT "search_word_level_course_vector"."node", "search_word_level_course_vector"."dimension", "search_word_level_course_vector"."score" FROM "search_word_level_course_vector" WHERE "search_word_level_course_vector"."node" = '3454' collapse query	django_redshift	False	0.01 sec	0.00 sec	2017-09-13 12:47:02.003 pm
5001583	SELECT "search_word_level_course_vector"."node", "search_word_level_course_vector"."dimension", "search_word_level_course_vector"."score" FROM "search_word_level_course_vector" WHERE "search_word_level_course_vector"."node" = '3444' collapse query	django_redshift	False	0.02 sec	0.00 sec	2017-09-13 12:47:01.769 pm
5001576	SELECT "search_word_level_course_vector"."node", "search_word_level_course_vector"."dimension", "search_word_level_course_vector"."score" FROM "search_word_level_course_vector" WHERE "search_word_level_course_vector"."node" = '3430'	django_redshift	False	0.01 sec	0.00 sec	2017-09-13 12:47:00.914 pm

The query ran 374,372 times. Each query returned a single row.

Query: 5001585

Start Time:	2017-09-13 5:47:02.003 am
End Time:	2017-09-13 5:47:02.014 am
Type:	select
Total Duration:	0.01 sec
Queue Time:	0.00 sec
Execution Time:	0.01 sec
User:	django_redshift
Label:	default
Aborted:	False
Is Diskbased?:	False
Database:	prod
Rows Scanned (Pre-Filter):	443,744
Memory:	0
Transaction ID:	39026256
Query Group:	6441d2757bf845a97f942fcfa2f46299

Query Text:

```
SELECT "search_word_level_course_vector"."node",
       "search_word_level_course_vector"."dimension",
       "search_word_level_course_vector"."score"
FROM "search_word_level_course_vector"
WHERE "search_word_level_course_vector"."node" = '3454'
```

The impact on the cluster is quite dramatic:

- 374,371 queries @ 0.02s per query equal 7,487 seconds, or 125 minutes of query time. The commit queue backs up with all these requests, impacting the execution time of all other queries running in the cluster

- The query volume drives up concurrency and may exceed the # of available WLM slots, which results in queue wait times for other queries running in that queue.

There two approaches to resolve the problem.

- **Re-write the queries** to select all 443,744 rows of the table, and then parse each row in application memory. Doing so would remove 374,371 queries from your Redshift database. Such a single query would take a few seconds, instead of 125 minutes.

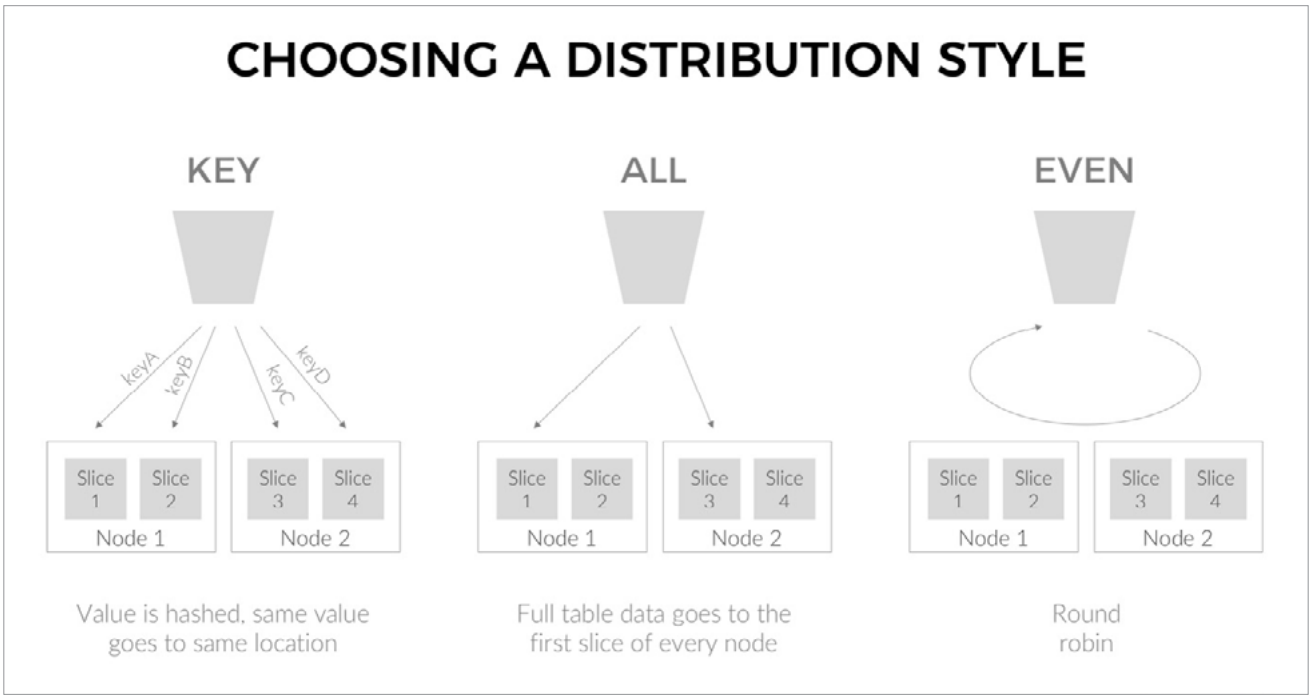
- **Use Amazon RDS and DBLINK** to use Redshift as an OLTP. It the post "Have your Postgres Cake and Eat it Too" we describe that approach in detail.

07

USE DISTKEYS ONLY WHEN NECESSARY

Distribution style is a table property which decides how to distribute rows for a given table across the nodes in your Amazon Redshift cluster.

Choosing the correct distribution style is important for query performance. The below diagram shows the three options.



There are two major consideration when choosing a distribution style:

1. Minimize data movement across nodes, which is expensive because of network I/O and disk I/O.
2. Distribute data evenly across your cluster to maximize query performance and minimize row skew. We will cover 'row skew' below.

EVEN-based Distribution

The default distribution style is 'EVEN'. All nodes contain an equal number of rows for a given table. The benefits of the 'EVEN' distribution style are:

- Table scans are fast, since all nodes have the same workload

- Disk utilization of nodes is the same, since there is no row skew. We will explain what 'row skew' is below.

However, 'EVEN' distribution is not optimal when joining two tables. Consider what happens when two tables are JOINed:

1. Select data for table 1
2. Select data for table 2
3. Move data to a single node (co-located)
4. Join data on that node and store results

... and the query execution continues from here.

With EVEN distribution, it's easy to see that step 3 requires the movement of data between nodes. This is not ideal because it requires network (broadcast) traffic and increases I/O utilization across the cluster. Both factors increase query latency.

KEY-based distribution

To solve this problem and make JOINs faster, Amazon Redshift offers a KEY-based distribution style. With KEY-based distribution, Amazon Redshift will ensure that for a given column across two tables, step 3 (move data to a single node) will not be necessary. This is accomplished by applying an algorithm when writing data to nodes. The algorithm ensures that rows with the same value in the 'DISTKEY' column end up on the same node.

Consider an example where the name of the JOIN column is *'customer_id'.*

1. The DISTKEY for table 1 must be *"customer_id"*
2. The DISTKEY for table 2 must be *"customer_id"*
3. Query 1 joins on table 1 and table 2 on *"customer_id"*

In this case, Query 1 will execute faster than the *case* when table 1 or table 2 use an EVEN-based distribution.

Downsides of KEY-based distribution

But what happens when I run another type of query against table 1? For example, a query that does not join on "customer_id" but on another column? Or does not do a JOIN at all? Queries which do not JOIN on these columns may run much slower.

There are two main downsides of using KEY based distribution.

1. Uneven node disk utilization. Row skew happens when you use KEY based distribution for a table, and the values in the DISTEY column are not evenly distributed. The result is that a node ends up having more rows for that table.
2. Slower queries. With different row counts, all other queries like a straight SELECT that touch that table will be a bit slower. Because one node has more data than the next, the query execution must wait for the "slowest" node" (i.e. the one with the most rows to send up its data to the leader node.

When to use KEY-based distribution

KEY-based distribution is great if and only if you have a major query that you want to optimize. In all other cases, use an EVEN-based distribution. Using EVEN distribution will:

- eliminate row skew
- ensure SELECTs of that table are optimized

08

MAINTAIN ACCURATE TABLE STATISTICS

Amazon Redshift builds a custom query execution plan for every query. For a given query plan, an amount of memory is allocated.

Here are some of the benefits to understanding the data:

The memory allocation is determined by estimating the amount of memory needed to store intermediate query results (as in a JOIN or aggregation).

The query plan allocates a certain amount of memory to each query, by estimating the amount of memory needed to store intermediate results (e.g. for a JOIN or aggregation).

It is important for a query to have sufficient memory to not spill to disk (go “disk-based”). Allocating too much memory is not desirable either however. Queries do not share memory. Allocating more memory than needed wastes memory since it is unavailable to other queries.

An important point is that the system is not adaptive. If the plan was wrong and the query needs more (or less) memory than was allocated – the execution engine will not go back and adjust the memory allocation after the query has already started executing.

What could cause the plan to be wrong? A major factor is the number of rows in a table.

The ANALYZE command will ensure that the planner has an accurate, up-todate view of the row counts for tables. Let's look at an example of what happens if the statistics are wrong.

EXAMPLE 1 – Table has more rows than the planner thinks it has

In this example, the planner will allocate too little memory to the query. Once the query starts running, it will encounter that it requires more memory than it was allocated. The query will go disk-based and thus will run slower than otherwise.

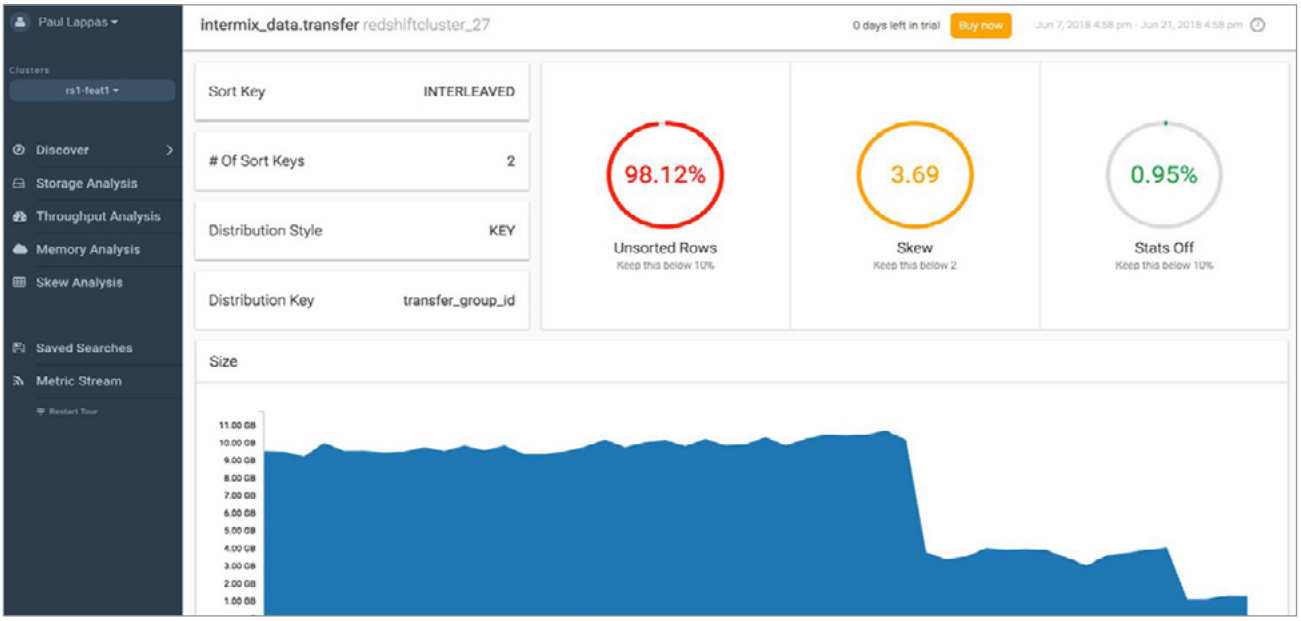
This could have been avoided by running the query in a slot with enough memory. It would not have gone disk-based.

1. The query will not go disk-based, but it used up too much memory. That may cause other queries to go disk-based.
2. The query was allocated more memory than was available in the slot it ran in, and the query goes disk-based. This could have been avoided with up-to-date statistics

Running ANALYZE

Amazon Redshift provides a statistics called "stats off" to help determine when to run the ANALYZE command on a table. The "stats off" metric is the positive % difference between the actual number of rows and the number of rows seen by the planner.

As a best practice, we recommend running ANALYZE on any tables with a "stats off" percentage greater than 10%.



09

USE ROUTINE VACUUMING TO RECLAIM UNUSED SPACE

Amazon Redshift databases require periodic maintenance known as vacuuming. Amazon Redshift is based on PostgreSQL, but unlike PostgreSQL, Redshift doesn't offer autovacuum. So when a row is deleted from a table in Amazon Redshift, the disk space used by that row is not immediately recovered. A special command is necessary to recover the disk space for use by other tables.

In Amazon Redshift, the "VACUUM FULL" operation will accomplish two things:

1. Sort tables (for tables that have a SORTKEY)
2. Reclaim space from rows that were flagged for deletion (as from a DELETE or UPDATE operation)

In most cases, it's not desirable to do both things at the same time. The requirements for sorting a table are very different from reclaiming space. Sorting may use a lot of resources and time.

Running ANALYZE

We recommend sepaerating the VACUUM DELETE ONLY operation from the SORT operation. The recommendation is to run VACUUM DELETE ONLY

1. after every ETL operation which UPDATEs or DELETEs from a table
2. nightly on all tables with "stats off" greater than 10

10

WRITE SMARTER QUERIES

Amazon Redshift is a distributed, shared-nothing database that scales horizontally across multiple nodes.

Query execution time is very tightly correlated with:

- the # of rows and data a query processes.
- the amount of data moving between nodes.

Below is an example of a poorly written query, and two optimizations to make it run faster.

```
1 with
2
3 table1_cte as
4
5 (
6
7 select * from table1
8
9 ),
10
11 table2_cte as
12
13 (
14
15 select * from table2
16
17 ),
18
19 select
20
21 *
22
23 from table1_cte as a
24
25 JOIN table2_cte as b
26
27 ON a.id = b.id
```

Optimization #1: Limit Rows Processed by using a WHERE clause

Queries can run faster by minimizing the amount of data moving between nodes.

In practice, this means being careful when writing multi-stage queries where the results of one stage feeds into the next.

In the case of our example query, modifying your ‘WHERE’ clauses to only select rows needed will minimize the amount of data that needs to be moved around and speed up the query.

```
with

table1_cte as

(

select * from table1 where
 created_at>'{{l_bound}}' and
 created_at <'{{u_bound}}'

),

 table2_cte as

 (

 select * from table1 where
 created_at >'{{l_bound}}' and
 created_at <'{{u_bound}}'

 ),

 select

 *

 from table1_cte as a

 JOIN table2_cte as b

 ON a.id = b.id
```

Optimization #2: Limit Columns Scanned

Amazon Redshift is a columnar-oriented database. As a result, scanning a table doesn't read each row in its entirety. Instead, individual columns can be scanned without needing to read other columns. This means that you should be careful to only select columns that you will use for your query. Try avoiding using a

*'SELECT *'*

operation in all cases.

Using these two optimizations when writing queries will have dramatic positive effects on your query speeds.

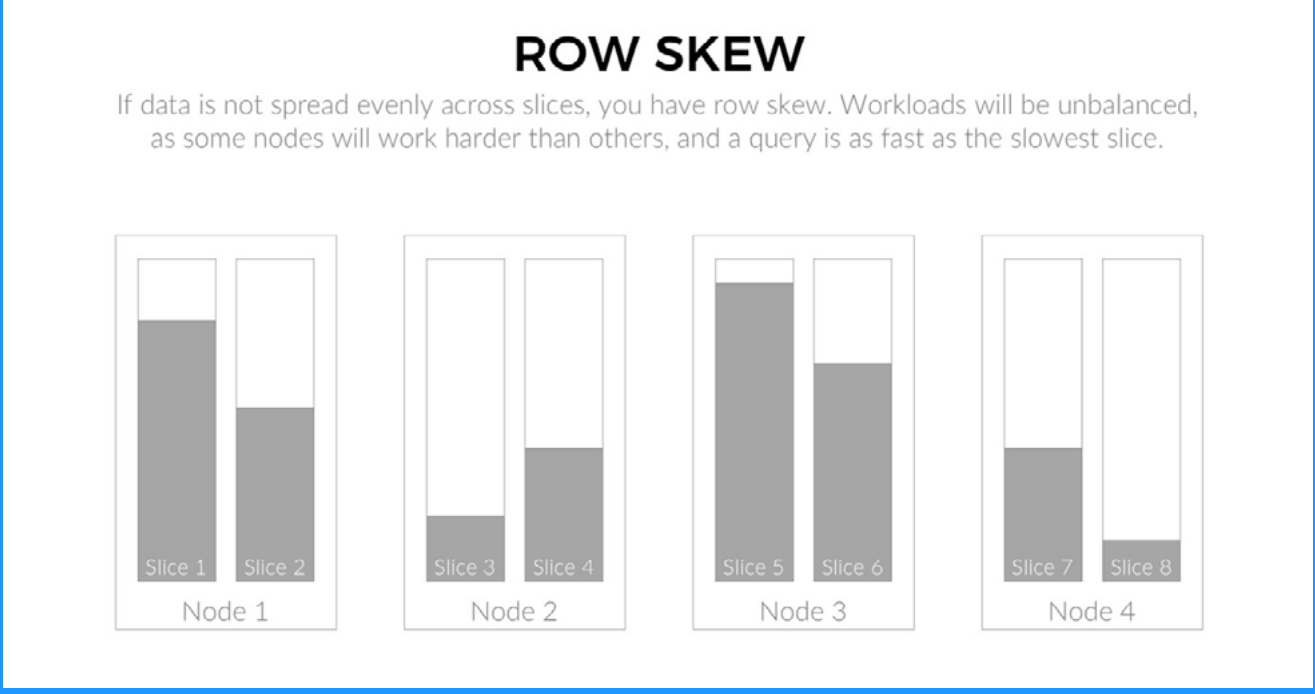
```
with

table1_cte as

(

select id,name,address from table1
 where start_time>'{{l_bound}}' and
 start_time<'{{u_bound}}'

),

 table2_cte as

 (

 select id,name,address from table1
 where start_time>'{{l_bound}}' and
 start_time<'{{u_bound}}'

 ),

 select

 a.name,b.address

 from table1_cte as a

 JOIN table2_cte as b

 ON a.id = b.id
```

11

AVOID ROW SKEW

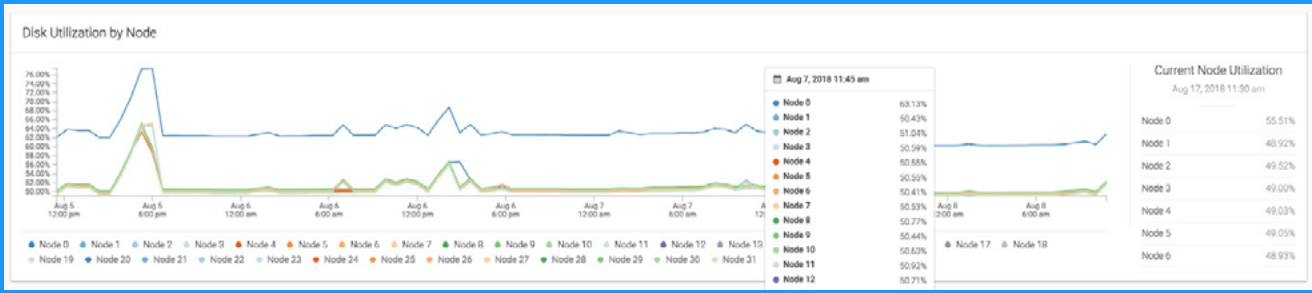
Row Skew results when a table uses KEY based distribution, and the values in the DISTKEY column are not evenly distributed. The row skew metrics is a positive integer ranging from 1 to the number of rows in the table. Row skew is the ratio of:

- number of rows on the node containing the most number of rows for the table
- number of rows on the node containing the least number of rows for the table



High row skew results in uneven node disk utilization (cost) and slower queries (performance).

The chart below shows a real-world example. With uneven disk utilization, a single node(s) ends up having more rows for that table. This can be a major (cost) problem if you need to add more nodes in your cluster just because a single node is skewed.



With high row skew, doing a straight SELECT on that table will be slower than otherwise. This is because one node has more data than the next, and the query execution must wait for the "slowest" node to send up its data to the leader.

There are two options to eliminate row skew:

1. selecting a DISTKEY that is random, or
2. change the distribution style to EVEN or ALL

The exception to tolerate row skew is if – and only if – you make a conscious decision to optimize a single query. See the section "Use DISTKEYs Only When Necessary" in this article for more information.

12

USE SHORT QUERY ACCELERATION (SQA)

Short Query Acceleration (SQA) will speed up the execution of short running queries.

It does this by selecting certain queries to jump the queue. This can be useful when your cluster runs a mixture of big and small queries. In this case, a small query that would otherwise queue up behind a longer query will execute first.

SQA is enabled by default on Amazon Redshift clusters. But using SQA without any other adjustments to your cluster is not a recipe for success. There are other levers to pull first. And then SQA becomes one part of your performance tuning strategy.

See our quick guide to using Short Query Acceleration and WLM for Amazon Redshift for faster queries.

13

COMPRESS DATA IN S3

The Amazon Redshift COPY command is the recommended way of moving data into Amazon Redshift.

The COPY command takes advantage of the parallel architecture in Amazon Redshift to move data. The COPY command can read files from various sources, including EMR, DynamoDB, and remote hosts via SSH.

Compressing files in S3 when loading large amounts of data will accomplish three goals:

1. Faster file upload to S3
2. Lower S3 storage utilization (cost)
3. Faster load process since uncompression un-compression can happen as files are read.

Long-running COPY commands will see the most improvement.

14

MANAGE VERY LONG TABLES

Amazon Redshift is very good for aggregations on very long tables (e.g. tables with > 5 billion rows). Some use cases call for storing raw data in Amazon Redshift, and then reducing the table and storing the results in subsequent, smaller tables later in the data pipeline.

This is a great use case in our opinion. However, managing very large tables presents two challenges:

1. Pruning (i.e. deleting historical data) can be very expensive.
2. Sorting the long table can be very expensive (or not possible)

This section discusses a few approaches to managing these issues for long tables.

Use UNION to make it easier to PRUNE very long tables

Pruning a long table requires running the DELETE operation. This needs to be done rather frequently to avoid the table filling up your disk. After every DELETE operation, you need to run the following three maintenance steps on the table:

1. Sort
2. Reclaim space on the table
3. Update statistics on the table

On a very long table, these operations can be very expensive. To avoid the three steps you can partition the very long table into smaller tables. Create multiple tables with the same schema, but with different table names. The rows in the table are then partitioned based on the chosen partition key. The job that INSERTs into these tables must be aware of the partitioning scheme.

To select from this table, create a view (with the original table name) and use the UNION directive to provide a consistent view to your application. This has the following benefits:

- The application doesn't need to care about the partitioning, since the VIEW presents the same table name
- Pruning is simply a matter of dropping the "oldest" table. No need to run VACUUM at all. Drop operations are very inexpensive and reclaim space immediately.

There is a downside to this approach. SELECTs on the table will go a bit slower, since the UNION operation won't be as fast as scanning a single table. But depending on your environment, it's a small trade-off worth it and a good solution to avoid the pain of maintaining a very long table.

COPY in sort order

We've learned that sorting is an expensive operation. If you use an UPSERT method to COPY new data into a table, you will need to sort that table.

UPSERT is a method of de-duplicating data when copying into Amazon Redshift. The UPSERT operation merges new records with existing records using primary keys. While some RDBMSs support a single "UPSERT" statement, Amazon Redshift does not support it. Instead, you should use a staging table for merging records.

Since UPSERT performs a DELETE, it may leave the table in an unsorted state. One approach to eliminate the need to sort the table is to COPY in sort order. This will prevent the need for you to ever sort the table.

There are a few caveats when considering using this method:

- Only works for COPYs (not regular inserts)
- Using a manifest is problematic because ordering of files isn't guaranteed
- The table can have only one sort key (interleaved style is not supported)
- The sort column should be NOT NULL and the table is 100 sorted (or empty)
- New rows are higher in sort order than existing rows, including rows marked for deletion.

15

USE AMAZON REDSHIFT SPECTRUM

Amazon Redshift launched with disruptive pricing.

Amazon Redshift launched with disruptive pricing. To compare the cost, we're looking at the price for storing 1TB of data for one year ($ / TB / Year). With a 3-year commitment for the ds2.8xlarge nodes, the price comes down to $934 / TB / Year. That price point is unheard of in the data warehousing world.

The average Amazon Redshift customers double their data every year. In fact, that is one of the reasons why it's important to focus on performance improvements – since managing performance becomes a bigger challenge as data volume grows.

At some point, the cost of storing all this data in Amazon Redshift become prohibitive. So Keeping a multi-year history of data "forever" can become expensive. Deleting data may not be an option, e.g. for regulatory reasons or multi-year comparisons.

The issue here is that Amazon Redshift prices based on the size of your cluster, i.e. compute and storage are coupled. You'll have to keep adding nodes for storage, even though you may not need the additional computing power of the additional vCPUs.

Because pricing is so cheap, a common initial behavior is to store all historical raw data in Redshift. But data volume is growing. You may also want to use the faster but more expensive dense compute nodes. Many companies don't want to make a capital commitment beyond a 1-year term.

So keeping a multi-year history of data “forever” can become expensive. Deleting data may not be an option, e.g. for regulatory reasons or multi-year comparisons.

The issue here is that Amazon Redshift prices based on the size of your cluster, i.e. it couples compute and storage. You’ll have to keep adding nodes for storage, even though you may not need the additional computing power of the additional vCPUs.

Enter Amazon Redshift Spectrum. With Redshift Spectrum, you can leave data as-is in your S3 data lake, and query it via Amazon Redshift. You can de-couple compute from storage. This approach makes sense when you have data that doesn’t require frequent access. Leave your “hot” data in Amazon Redshift, and your “cold” data in S3.

The impact on cost can be substantial. The price for S3 Standard Storage is $281 / TB / Year. And so with Redshift Spectrum, you get the best of both worlds. We call it “data tiering”. You get to keep all your historical data, along with the performance of Amazon Redshift. With Redshift Spectrum you can benefit from the cost savings of using S3.

In “Amazon Redshift Spectrum: Diving into the Data Lake!”, we’re taking an even closer look at using Redshift as part of a data lake architecture, including the use of Amazon Athena and AWS Glue.