Amazon Redshift Serverless

Max Ganz II @ Redshift Research Project

4th October 2023

Abstract

Redshift Serverless is *not* serverless. A workgroup is a normal, ordinary Redshift cluster. All workgroups are initially created as a 16 node cluster with 8 slices per node, which is the default 128 RPU workgroup, and then elastic resized to the size specified by the user. This is why the original RPU range is 32 to 512 in units of 8 and the default is 128 RPU; the default is the mid-point of a 4x elastic resize range, and a single node, the smallest possible change in cluster size, is 8 RPU/slices. 1 RPU is 1 slice. With elastic rather than classic resize, the greater the change in either direction from the size of the original cluster, the more inefficiency is introduced into the cluster. As a cluster becomes increasingly larger, it becomes increasingly computationally inefficient (the largest workgroup has 128 normal data slices, but 384 of the lesser compute slices), and increasingly under-utilizes disk parallelism. As a cluster becomes increasingly smaller, it becomes increasingly computationally inefficient (each data slice must process multiple slices' worth of data), and incurs increasingly more disk use overhead with tables. The more recently introduced smaller workgroups, 8 to 24 RPU (inclusive both ends) use a 4 slice node and have two nodes for every 8 RPU. In this case, the 8 RPU workgroup is initially a 16 node cluster with 8 slices per node, which is resized to a 2 node cluster with 4 slices per node - a staggering 16x elastic resize; the largest resize permitted to normal users is 4x; an 8 RPU workgroup, with small tables, uses 256mb per column rather than the 16mb per column of a native two node cluster. Workgroups have a fixed number of RPU and require a resize to change this; workgroups do not dynamically auto-scale RPUs. I was unable to prove it, because AutoWLM is in the way, but I am categorically of the view that the claims made for Serverless for dynamic auto-scaling are made on the basis of the well-known and long-established mechanisms of AutoWLM and Concurrency Scaling Clusters ("CSC"). Finally, it is possible to confidently extrapolate from the `ra3.4xlarge` and `ra3.16xlarge` node types a price as they would be in a provisioned cluster for the 8 slice node type, of 6.52 USD per hour. Both Provisioned and Serverless clusters charge per node-second, but Serverless goes to zero cost with zero use. With the default Serverless workgroup of 128 RPU/16 nodes (avoiding the need to account for the inefficiencies introduced by elastic resize), one hour of constant use (avoiding the need to account for the Serverless minimum query charge of 60 seconds of run-time), without CSC (avoiding the question of how AutoWLM will behave), costs 46.08 USD. A Provisioned cluster composed of the same nodes costs 104.32 USD; about twice as much. Here we have to take into consideration the inefficiencies introduced by elastic resize, which become more severe the more the cluster deviates from 16 NRPU, that Serverless uses AutoWLM, with all its drawbacks, and which is a black box controlling the use of CSC, with each CSC being billed at the price of the workgroup, and the 60 second minimum charge. All Serverless costs (including the charge for Redshift Spectrum S3 access) have been rolled into a single AWS charge for Serverless, so it is not possible to know what is costing money. It would have been much better if AWS had simply introduced zero-zero billing on Provisioned clusters. This would avoid many of the weaknesses and drawbacks of Serverless Redshift, which wholly unnecessarily devalue the Serverless product, as well as avoiding the duplicity, considerable

added complexity, end-user confusion, cost in developer time and induced cluster inefficiency involved in the pretence that Serverless is serverless.

Contents

Introduction

AWS previewed the product "Amazon Redshift Serverless" in 2021-11, which went to GA in 2022-07.

AWS provide a product comparison page, and at 2021 re:Invent, Adam Selipsky, AWS CEO, introduced Redshift Serverless as one of the four serverless services offered by AWS.

This document first introduces Redshift Serverless, explaining its concepts and how it is operated, then investigates Serverless, and looks in particular to try to discern the underlying architecture, so the correct use and actual value of Serverless can be known from and ideally from first principles.

Introduction to Redshift Serverless

Every AWS account has, on a per-region basis, a single Redshift Serverless system.

Redshift Serverless operates in terms of two concepts, which AWS have termed *workgroups* and *namespaces*.

A workgroup represents computing resource, a namespace represents data.

Workgroups

You can have as many workgroups as you like (up to the account limit, which begins at 25 but can be raised). A workgroup is defined by its name and a single parameter, *Redshift Processing Units* (RPU from here on in), and a workgroup can have from 8 to 512 RPU, but they increment in units of 8, so it's really 1 to 64 units, which I will call Normalized Redshift Processing Units (NRPU), and usually use instead of RPU as normalized units are more natural and so easier to use and grasp, and also because I'm not in a marketing department and so have no intrinsic preference for large numbers.

An RPU is *not* defined in any way, shape or form, excepting the and the only statement made about them, in the docs, is that;

> *One RPU provides 16 GB of memory.*

Which means that 1 NRPU is 128 GB of RAM. The same page also gives the general impression that "more RPU is better".

> *Setting higher base capacity improves query performance, especially for data processing jobs that consume a lot of resources.*

There is however no information about *how* more RPU is better. This leaves open the question, for example, of whether more RPU means more queries in parallel, or if individual queries go faster, or both.

Pricing is in fact as we shall see not fully defined by AWS, but AWS present pricing as being "per-RPU-hour", and Serverless is defined as a serverless system, so a natural implication is that NRPU are identical and linear, since you pay the same amount per-RPU no matter how many you have; this would then imply 4 NRPU is four times as much computing power as 1 NRPU (whatever computing power is).

The minimum number of NRPU originally was 4, which if the 16 GB figure is correct, is 512 GB of RAM, and the default number of NRPU was and is 16, which is an extraordinary 2 TB of RAM. That's a remarkably large amount and noteworthy amount of memory for a default, and we will be coming back to this point later in the investigation.

Eights months after GA, the minimum was reduced to 1 NRPU (I suspect because the still high price of the previous minimum of 4 was discouraging uptake).

A workgroup provides an IP address and port, which are needed to connect to with your SQL client, but connecting also requires a user name and a database name, which are not provided by a workgroup; rather, these are provided by the namespace (as you will read in the next section, which describes namespaces).

To issue queries, you must create a workgroup and assign a namespace to it (more on this below), which provides you with the full set of IP, port, username and database, to which you

then connect in the usual way with your usual SQL client, and then issue queries.

Creating a workgroup takes some minutes, plus up to another minute or two after the Redshift UI reports the workgroup is created before you can actually connect.

The compute capacity of the workgroup, its number of RPUs, is defined when the workgroup is created.

The term used in the docs and UI for the number of RPU is *Base Capacity*, which implies that it is a minimum value. This is *not* the case. A workgroup when created has the number of RPU you specify when you create it; no more and no less, and the number of RPU does not change by itself. If you want to change the number of RPU, you must resize the workgroup, which takes several minutes, during which time the workgroup goes off-line.

When you're finished with a workgroup, you delete it, which also takes some minutes.

Workgroups execute queries wholly independently of other workgroups, and they are created, resized and deleted, wholly independently of other workgroups.

Comparing workgroups to say, AWS Lambda, we see that with Lambda, when you issue a lambda function, and specify the resource to be allocated to that function and fire it off. That's it.

With Redshift Serverless, you must create a workgroup, some minutes, which defines the compute capacity of that workgroup; all queries issued to that workgroup receive that compute capacity, and when you're done, you delete the workgroup, some more minutes.

You could then imagine creating a range of workgroups, so you have a range of compute capacities, and issuing queries to the workgroup with the appropriate compute capacity for the query - however, this doesn't work, because - as we will see - only one workgroup at a time can operate on a given namespace, a namespace being basically being a snapshot (but constrained such that it can be associated with only a single workgroup at a time).

It must be said at this point that Serverless does not look very serverless.

However, a property typically associated with serverless is dynamic auto-scaling, in some shape or form, and there are a few allusions in the official docs to auto-scaling - but I must say allusions because they are nothing strong - and two more concrete, but ambiguous, statements, which we find in a page about Serverless billing.

First, the allusions;

On the *Amazon Redshift Serverless* home page, we see this and only this;

> *Auto-Scaling*
>
> *Scale resources seamlessly and maintain consistent performance for thousands of concurrent users as workload demands change or spike with traffic.*

On the *Billing for Amazon Redshift* page, we see this and only this;

> *As the number of queries increase, Amazon Redshift Serverless scales automatically to provide consistent performance.*

Next, we can see the following claim, on *What is Amazon Redshift Serverless?*;

> *Amazon Redshift Serverless automatically provisions data warehouse capacity and intelligently scales the underlying resources. Amazon Redshift Serverless adjusts ca-*

> *pacity in seconds to deliver consistently high performance and simplified operations for even the most demanding and volatile workloads.*

Now for the two more concrete statements, which we find in *Billing usage notes*;

> *Scaling - The Amazon Redshift Serverless instance may initiate scaling for handling periods of higher load, in order to maintain consistent performance. Your Amazon Redshift Serverless billing includes both base compute and scaled capacity at the same RPU rate.*

This is to my eye ambiguous - the final sentence. Does this mean you pay the same rate whether the workgroup is scaled or not, or does it mean you pay the same rate for scale capacity as you do for the workgroup?

> *Scaling down - Amazon Redshift Serverless scales up from its base RPU capacity to handle periods of higher load. It some cases, RPU capacity can remain at a higher setting for a period after query load falls. We recommend that you set maximum RPU hours in the console to guard against unexpected cost.*

This then implies it must be the second reading; you pay for scaled capacity at the same rate as the workgroup.

So, what I've quoted above is *it*. Everything I've found about scaling. No details, no characterization, no limits, no performance indications, a single implication about pricing. What you've read there is all of the information AWS have provided.

AWS, to my eye, have with Redshift (I have no knowledge of other products) a long and studied history of obfuscation and profoundly inflated, improper and misleading claims, and so to see such claims, with no backup, has for me alarm bells going off like the Ectomobile on its first run to the NYC library.

In any event, what I can say at this point is that however dynamic auto-scaling does work, it is *not* via the mechanism of RPUs, as when a workgroup is created, the number of RPUs is specified and can only be changed manually, by resizing the cluster, which takes several minutes and takes the cluster off-line.

I will visit this matter and these claims in the Discussion of the test script results.

Namespaces

You can have as many namespaces as you like (up to the account limit, which begins at 25 but can be raised). A namespace is defined by its name and when created begins almost empty; it contains a single initialized database with the rdsdb and the admin user, so it contains the system tables, all the usual functions (thousands of them), and so on - in short, exactly the state you find in a provisioned Redshift cluster when it is brought up for the first time.

When you create a workgroup, you need to specify a single namespace, and that workgroup will operate upon the data in that namespace.

A single workgroup works on a single namespace at a time.

Only one workgroup can work on a given namespace at a time.

A namespace however does not *require* that it is loaded into a workgroup. A namespace can be unused. However, a workgroup to be started *must* have a namespace specified.

A namespace can contain as many databases as Redshift Serverless allows (which is 100, and the docs state this is a hard limit), and as many tables, schemas, user and so on, as are permitted by Redshift Serverless (where the limits which are documented are identical to those of provisioned Redshift, except for the number of databases, which is 100 rather than 60).

It is from the namespace you obtain the database name and user name you need when connecting with your SQL client to the workgroup which is loaded with that namespace.

Namespaces are created almost instantly, but they take several minutes to delete.

General Operation

Serverless then is operated by creating a workgroup, where creating a workgroup requires specifying a namespace for the workgroup, and the workgroup will operate on and only on that namespace, and one and only one workgroup can operate at any given time on a given namespace.

A namespace does not have to be in use.

Workgroups are independently sized, and are sized in and only in terms of the number of RPU they have, where each workgroup has its own IP address, by which it is connected to by SQL clients.

Creating, re-sizing and deleting a workgroup all take some minutes. A workgroup goes off-line during a resize.

Creating a namespace is just about instant, but deleting takes some minutes (the duration may relate to how much data is in the namespace - I've not investigated this; all my namespaces have had almost no data in).

Pricing

All prices are for `us-east-1`.

With provisioned Redshift, you pay per-node-second, which is to say, you pay for and only for the number of nodes you have. The number of running queries irrelevant.

With Redshift Serverless billing is *not* like other serverless systems. If we look at Athena, billing is per query. If we look at Lambda, it is per Lambda function. With Redshift Serverless, however, and unexpectedly, billing is *not* per-query. Billing is in fact per-workgroup-second, where the price depends on the number of RPU in the workgroup, seemingly regardless of the number of running queries.

This leaves open the question of how many queries a workgroup can run, and also raises then the question of scaling - both of which I cannot address here as the results of the investigation are needed to explain the answers to these questions.

However, if no queries are running, you pay nothing; Serverless does offer zero-zero billing.

Where Serverless stores data in RMS, there is also the RMS price of 2.4 cents per gigabyte per month, or to put it in more accessible terms, 24.576 USD per terabyte per month. This is as we shall see almost negligibly cheap compared to the cost of running queries - one hour of 1 NRPU with 10 queries is one month of 1 terabyte of store.

There is also a 60 second minimum charge. Any query, or set of queries running in parallel, where the single queries runs for less than sixty seconds, or the set of queries running in parallel run for less than 60 seconds, is billed as if it or they ran for 60 seconds.

On a busy workgroup, queries will be running all the time, so the minimum charge will disappear - the workgroup will always be in use. For an idle or almost idle workgroup, the minimum is going to be costly if many short single queries are run, which on the face of it seems an unlikely scenario.

The smallest workgroup, 1 NRPU, used continuously for one hour costs 2.88 USD.

It's not clear if there is a limit to how many queries can run in parallel, or what happens if that limit is exceeded.

It is not clear how much compute 1 NRPU is, or the character of that compute - do more NRPU mean more queries running at the same speed, or does it mean the same number of queries but they run faster? the docs say nothing.

The smallest possible provisioned cluster, two nodes of `dc2.large` (never use single node clusters), costs 0.50 USD per hour, with a larger node type, two nodes of `ra3.xlplus`, coming in at 2.172 USD per hour.

On a normal cluster you'd be running about 10 queries concurrently.

If Serverless workgroups can also run about 10 queries concurrently, the smallest workgroup costs as we see something more than the smallest `ra3` cluster.

We see multiple but entirely vague claims from AWS about auto-scaling - no details at all. It's just about implied that pricing rises as scaling kicks in, that scaling changes the number of RPU (which it categorically does not - you have to manually resize a cluster to do this), and that the pricing for scaling is the same as the price of the main workgroup - but what does that *actually mean?*

For example, if I have a workgroup running ten queries, will it have scaled or not? if so, by how much? how "much" of a workgroup is the scaling, if it occurred, so I can figure out the price of that scaling?

There is *no* information about this, and the number of RPU in a workgroup *is* fixed, so this matter is shrouded in mystery, which is great for a thriller novel but highly inappropriate for a commercial database offering.

I'm not going to write more than this here, because later on, after we examine the results of the test script, we will understand auto-scaling and so its pricing, and these matters will become clear.

System Tables

The large majority of system tables have in Serverless been made inaccessible to both admin and normal users.

The docs, as ever, contain fragmentary information only.

I originally wrote at length about the failings of the Serverless docs, the length being necessary to cover all the issues relating to the system tables, but - flogging a dead horse, you know? so I've removed all of that. Suffice to say, the docs are to information what the South African National Electricity Company is to a stable power supply.

What I have found is that the `PG`, most of the `SYS` and most of the `SVV` system tables are accessible, *but* AWS have to my eye specifically acted to make inaccessible those system tables which provide information on *what* Serverless is doing internally, or *how*.

The system tables which remain accessible provide information about *data* - about tables, users, groups, etc - but no information about *function* - about queries and load and run-times and so on - so they're no use to try to understand what's going on under the hood.

As such we see the `STL`, `STV`, `SVCS` and `SVL` tables and views all have been made inaccessible (although I have not tested every view to check, as there are thousands of them, and so this would be exorbitantly expensive, where queries have a 60 second minimum charge), as have those `SVV` views which provide information about what Serverless is doing internally (I did test each `SVV` view individually, as there's only about 60 or so of them).

The `PG` tables and views come from Postgres and are pretty much all on the leader node, and contain information about data - tables, users, groups, etc.

The `SYS` tables are recent, I think were always intended to be for Serverless, and so most are accessible because they were originally created in a sanitized form (lacking "what and how" information).

Furthermore, and quite remarkably, on Serverless, AWS have taken the step of censoring the SQL of *all* the non-`PG` system table views which remain - so all the `SVV` views, and all the `SYS` views.

To do this, the SQL function which returns the SQL of a view has been modified such that for the censored views, this function returns not the SQL of the view but the following;

> *Error: describing system view <[view_name]> is disabled.*

(In provisioned Redshift, this censorship exists only for the `SYS` views. The source of the `SVV` views, these views having existed for many years, can still be seen. I also must note this output breaks the semantics of the function to display the SQL of a view, since we now have a plain text error message where there should be and only be DDL - this is however by no means the first time I've seen this type of interface corruption from the devs.)

I am firmly of the view AWS are specifically acting to prevent Serverless users knowing what's going on under the hood.

We could think to explain this by saying AWS are looking to automate Serverless and so relieve users from needing to know about such information; however, as we shall see, a different explanation will after investigation present itself.

Tests

The tests are almost all scripted, but one test must be performed by hand using `psql`, as the Python module `psycopg2`, which is used by the test script to communicate with Redshift, is confused by dates prior to 1 AD.

Workgroups when they are created are configured by specifying a value for Redshift Processing Units (RPU), which ranges from 8 to 512, in units of 8. I have normalized this, to the range 1 to 64, and called this Normalized Redshift Processing Units (NRPU).

The automated tests are split into four sets of tests; two sets of slow tests for Serverless, one set of fast tests for Serverless, and one set which runs on a provisioned cluster.

(There are two sets of slow tests, rather than one, because I found Serverless and `boto3` with Serverless to be unreliable, which amongst other consequences forced me to eschew concurrent bring-up/shut-down of multiple workgroups and namespaces and instead serialize bring-up and shut-down of workgroups and namespaces, which made the test script exceedingly slow - at one point, nine hours for a single run through, and this with constant intermittent and unexpected failures - breaking the slow tests into two allowed me during development to move isolate unfinished and/or still unreliable tests from those which were finished or had been made reliable.)

First Set of Slow Tests

1. The first tests time workgroup resize. A 16 NRPU workgroup, the default size, is created, and then resized, and then deleted. A new namespace is created for each test, and deleted after the resize is complete. This is repeated seven times, with the resize being to 1, 2, 3, 4, 8, 32 and 64 NRPU.

   (Sizes less than 4 are expected to be different in their nature to sizes 4 and above, so all three sizes less than 4 are being examined.)

   The time taken to create (`boto3` reports `available`), and then to actually successfully connect to, and the time taken to delete, each workgroup, is recorded.

   The time taken to create and delete each namespace is also recorded.

2. A 1 NRPU workgroup is created, a test table is created, and then the workgroup is disconnected from and left idle for 61 minutes (I determined this duration by prior experimentation). The workgroup is then connected to, and issued a query on the test able, and then issued the same query a second time. The time taken to connect, and the time taken to execute each query, is recorded.

Second Set of Slow Tests

1. A series of new namespace-workgroup pairs are created, the sizes being 1, 2, 3, 4, 8, 16, 32 and 64 NRPU.

   The time taken to create (`boto3` reports `available`), and then to actually successfully connect to, and the time taken to delete, each workgroup, is recorded.

   The time taken to create and delete each namespace is also recorded.

   Once a workgroup can be connected to, a test table named `table_slices` is created, with key distribution and two `raw` encoding `int8` columns, where the first column is an

identity column, starting at 1 and incrementing by 1 and which is also the distribution key. 1024 rows are written to the table, which being the single insert after table create will automatically be sorted, and then the following query is issued;

1. `select size from svv_table_info where "table" = 'table_slices';`

After this query, I insert to the table using a self-join, with a `LIMIT` of 250,000 rows, and issue the following query;

2. `select distinct slice_num() as slice from table_slices order by slice;`

Finally, the time taken to create and delete each namespace is also recorded.

Docs links;

1. https://docs.aws.amazon.com/redshift/latest/dg/r_SVV_TABLE_INFO.html
2. https://www.redshiftresearchproject.org/system_table_tracker/1.0.55524/pg_catalog.svv_table_info.html
3. https://docs.aws.amazon.com/redshift/latest/dg/r_SLICE_NUM.html

Fast Tests

1. A single 1 NRPU workgroup is brought up, along with a new namespace.
    1. The `EXPLAIN` query plans are generated for `select *` from every `SYS` system table views. We use these plans to find out which actual tables are being used by these views (we cannot examine the SQL of these views, as it is censored by AWS). We will then examine the columns in these tables, so see what we find.
    2. Every `SVV` system table is tested to see if it is accessible.

Provisioned Tests

1. A four node `ra3.xlplus` provisioned cluster is created.

    Once the cluster can be connected to, the following query is issued, together info about nodes and slices;

    1. `select node, slice, localslice, case when type = 'C' then 'compute' when type = 'D' then 'data' end as type from stv_slices order by node, slice;`

    Then, exactly as performed in the slow workgroup tests, a test table named `table_slices` is created, with key distribution and two `raw` encoding `int8` columns, where the first column is an identity column, starting at 1 and incrementing by 1 and which is also the distribution key. 1024 rows are written to the table, which being the single insert after table create will automatically be sorted, and then the following query is issued;

    2. `select size from svv_table_info where "table" = 'table_slices';`

    After this query, I insert to the table using a self-join, with a `LIMIT` of 250,000 rows, and issue the following query;

    3. `select distinct slice_num() as slice from table_slices order by slice;`

    The cluster is then resized to two nodes, and the first and third queries are re-issued (the second query is not re-issued, because adding 250k rows changes the result, but since we

repeat the whole test, but starting with two nodes and then resizing to four, we get to find the result of the second query on two nodes anyway).

The above tests are repeated, but starting with a two node cluster, which is then resized to a four node cluster.

Manual Tests

1. Create a 2 node `dc2.large` cluster.

   In normal Redshift clusters, leader nodes and worker nodes print dates prior to 1 AD differently.

   This query, `select '0100-01-01 BC'::date`, runs on the leader node and prints an AD/BC format date.

   This query, `select '0100-01-01 BC'::date from table_1;`, runs on the worker nodes and prints an ISO 8601 format date. (The table `table_1`, with a single for for convenience, must first be created, like so, `create table table_1 as select 1;`.

   The manual tests then are to bring up a 2 node `dc2.large` cluster, and run these queries, recording their output, and then to repeat, but on a 1 NRPU workgroup.

Results

First Set of Slow Tests

Resize Times (in seconds)

Resize in all cases is from 16 NRPU to the target size in the NRPU column.

NRPU	Available	Connect
64	221.70308017730713	1.1366949081420898
32	290.20202445983887	1.7749905586242676
8	307.84038615226746	2.2486279010772705
4	237.87361240386963	2.4608988761901855
3	223.4217014312744	1.3041656017303467
2	214.12547516822815	2.4654417037963867
1	261.23654770851135	1.3217377662658691

Additional Workgroup Times for 16 NRPU (in seconds)

For the seven 16 NRPU workgroups created for the resize tests.

NRPU	Available	Connect
16	142.00040555000305	42.092533111572266
16	158.3465723991394	8.1436448097229
16	122.82265400886536	40.22297286987305
16	124.32334208488464	75.41635346412659
16	127.80255341529846	104.72378063201904
16	127.83186173439026	93.99888372421265
16	127.59397530555725	5.925532102584839

Additional Namespace Times for 16 NRPU (in seconds)

For the seven 16 NRPU workgroups created for the resize tests.

NRPU	Create	Delete
16	0.5747129917144775	195.5482771396637
16	0.2790834903717041	537.4131038188934
16	0.37318849563598633	267.475439786911
16	0.30323123931884766	264.1939504146576
16	0.41089797019958496	149.59405493736267
16	0.2731807231903076	146.12539196014404
16	0.409212589263916	302.38894510269165

Connection and Query Times for Idle Workgroup

Wait duration (in minutes) = 61.000004458427426

Event	Duration (seconds)
Connection	2.052560806274414
First query	27.26804208755493
Second query	0.12465095520019531

Debug output from test script, when attempting to delete namespace workgroup immediately after idle workgroup test;

Output
The Cluster Idle Test… delete_workgroup(wg-u5-1) Failed to delete workgroup, attempt 1, error An error occurred (ConflictException) when calling the DeleteWorkgroup operation: There is an operation running on the workgroup. Try to delete it later. Failed to delete workgroup, attempt 2, error An error occurred (ConflictException) when calling the DeleteWorkgroup operation: There is an operation running on the workgroup. Try to delete it later. Failed to delete workgroup, attempt 3, error An error occurred (ConflictException) when calling the DeleteWorkgroup operation: There is an operation running on the workgroup. Try to delete it later. Failed to delete workgroup, attempt 4, error An error occurred (ConflictException) when calling the DeleteWorkgroup operation: There is an operation running on the workgroup. Try to delete it later.

Second Set of Slow Tests

Workgroup Times (in seconds)

NRPU	Available	Connect	Shutdown
64	162.94810509681702	36.03853249549866	149.57489109039307
32	158.84450340270996	3.484558582305908	277.4090383052826
16	122.33415484428406	23.383538246154785	164.1367542743683
8	121.4271936416626	69.81401205062866	184.42388319969177
4	122.49102735519409	79.7402982711792	238.34082007408142
3	153.5724790096283	78.02273607254028	284.17035365104675
2	120.54271459579468	15.95941710472107	57.93810534477234
1	116.43005561828613	22.12782907485962	314.46830320358276

Namespace Times (in seconds)

NRPU	Create	Delete
64	2.6436049938201904	139.958984375
32	0.2426462173461914	154.62592577934265
16	0.3628230094909668	197.90971612930298

NRPU	Create	Delete
8	0.40954065322875977	130.91410112380981
4	0.38789892196655273	663.5637822151184
3	0.4128589630126953	225.63704991340637
2	0.28119683265686035	469.62274193763733
1	0.27158045768737793	190.16908884048462

Slices

NRPU	# slices	min	max	slice numbers
64	96	2	503	2, 4, 8, 10, 12, 14, 16, 18, 20, 21, 22, 24, 31, 34, 38, 40, 42, 47, 48, 50, 52, 54, 56, 62, 64, 66, 68, 71, 72, 74, 76, 78, 80, 82, 84, 88, 90, 92, 94, 96, 97, 99, 104, 107, 112, 115, 117, 118, 120, 124, 126, 128, 148, 151, 161, 173, 176, 203, 206, 210, 212, 215, 223, 225, 229, 236, 237, 260, 261, 266, 274, 286, 302, 304, 308, 312, 314, 326, 341, 345, 375, 390, 411, 428, 429, 434, 435, 448, 452, 455, 458, 460, 464, 475, 497, 503
32	106	0	254	0, 2, 5, 7, 8, 10, 12, 13, 15, 16, 19, 20, 22, 27, 30, 32, 40, 44, 46, 48, 49, 50, 52, 53, 56, 58, 60, 61, 64, 65, 66, 69, 70, 72, 73, 78, 80, 84, 85, 88, 93, 97, 98, 100, 101, 105, 109, 113, 114, 116, 118, 120, 121, 124, 125, 126, 128, 132, 134, 138, 139, 141, 146, 150, 154, 155, 156, 157, 158, 160, 161, 165, 166, 167, 168, 169, 175, 177, 194, 196, 199, 202, 205, 206, 207, 209, 210, 213, 215, 216, 218, 222, 223, 226, 230, 232, 234, 236, 237, 238, 240, 243, 244, 247, 248, 254
16	95	0	127	0, 1, 3, 5, 6, 8, 9, 10, 11, 12, 13, 14, 16, 17, 20, 21, 22, 26, 27, 28, 29, 30, 31, 33, 35, 36, 37, 38, 39, 40, 42, 43, 46, 47, 50, 51, 52, 54, 55, 58, 59, 61, 62, 63, 64, 66, 67, 68, 69, 71, 72, 73, 74, 75, 76, 77, 79, 80, 82, 83, 84, 85, 86, 87, 88, 91, 93, 94, 95, 96, 97, 99, 100, 102, 103, 104, 105, 106, 107, 108, 109, 110, 111, 113, 114, 115, 116, 117, 118, 119, 121, 122, 125, 126, 127
8	58	0	119	0, 1, 2, 5, 6, 7, 16, 17, 18, 19, 20, 21, 22, 23, 32, 33, 34, 35, 38, 39, 48, 49, 50, 51, 52, 53, 54, 55, 64, 65, 66, 67, 68, 69, 70, 71, 80, 81, 82, 83, 84, 85, 86, 96, 97, 98, 99, 100, 101, 102, 103, 112, 113, 115, 116, 117, 118, 119

NRPU	# slices	min	max	slice numbers
4	28	0	101	0, 1, 2, 4, 5, 6, 7, 32, 33, 34, 35, 36, 37, 38, 39, 64, 65, 66, 67, 68, 69, 70, 71, 96, 97, 99, 100, 101
3	24	0	110	0, 1, 2, 3, 22, 23, 24, 25, 44, 45, 46, 47, 65, 66, 67, 68, 86, 87, 88, 89, 107, 108, 109, 110
2	16	0	99	0, 1, 2, 3, 32, 33, 34, 35, 64, 65, 66, 67, 96, 97, 98, 99
1	8	0	67	0, 1, 2, 3, 64, 65, 66, 67

Blocks

There are 5 columns and 2 segments (sorted and unsorted), the test data generates one block per column, per segment, per slice, so the number of data slices is the number of blocks divided by 10 (i.e. 5*2).

NRPU	# blocks	∴ # data slices
64	1280	128
32	1280	128
16	1280	128
8	1280	128
4	1280	128
3	1280	128
2	1280	128
1	1280	128

Fast Tests

Accessible SVV System Tables

Name	Accessible
svv_all_columns	True
svv_all_schemas	True
svv_all_tables	True
svv_alter_table_recommendations	True
svv_attached_masking_policy	True
svv_column_privileges	True
svv_columns	True
svv_database_privileges	True
svv_datashare_consumers	True
svv_datashare_objects	True
svv_datashare_privileges	True
svv_datashares	True
svv_default_privileges	True
svv_diskusage	False

Name	Accessible
svv_external_columns	True
svv_external_databases	True
svv_external_partitions	True
svv_external_schemas	True
svv_external_tables	True
svv_function_privileges	True
svv_geography_columns	False
svv_geometry_columns	False
svv_iam_privileges	False
svv_identity_providers	True
svv_integration	True
svv_interleaved_columns	True
svv_language_privileges	True
svv_masking_policy	True
svv_ml_model_info	True
svv_ml_model_privileges	True
svv_mv_dependency	True
svv_mv_info	True
svv_query_inflight	False
svv_query_state	False
svv_redshift_columns	True
svv_redshift_databases	True
svv_redshift_functions	True
svv_redshift_schemas	True
svv_redshift_tables	True
svv_relation_privileges	True
svv_restore_table_state	False
svv_rls_applied_policy	False
svv_rls_attached_policy	True
svv_rls_policy	True
svv_rls_relation	True
svv_role_grants	True
svv_roles	True
svv_schema_privileges	True
svv_schema_quota_state	False
svv_sem_usage	False
svv_sem_usage_leader	False
svv_system_privileges	True
svv_table_info	True
svv_tables	True
svv_transactions	True
svv_user_grants	True
svv_user_info	True
svv_vacuum_progress	False
svv_vacuum_summary	False

Dependencies for SYS System Tables

System Table	Depends Upon
sys_connection_log	stll_connection_log
sys_copy_job	pg_auto_copy_job
sys_datashare_change_log	stll_datashare_changes_producer, stll_datashare_changes_consumer
sys_datashare_usage_consumer	stll_datashare_request_consumer
sys_datashare_usage_producer	stll_datashare_request_producer
sys_external_query_detail	stll_external_query_detail_base
sys_integration_activity	stll_integration_log
sys_integration_table_state	stv_integration_table_state
sys_load_detail	stll_load_commits, stll_user_query_map
sys_load_error_detail	stll_user_load_error_detail, stll_load_history_base, stv_user_query_state
sys_load_history	stll_load_history_base, stv_user_query_state
sys_query_detail	stll_query_detail, stv_exec_state, stv_user_query_map
sys_query_history	stll_user_query_history, stv_user_query_state
sys_query_text	stll_user_query_text
sys_serverless_usage	stll_arcadia_billing_log, stll_arcadia_billing_ris_reports, stll_cluster_blocks_stats, stll_xregion_metering
sys_spectrum_scan_error	stll_spectrum_scan_error, stll_query_detail
sys_stream_scan_errors	stll_kinesis_scan_errors, stll_kafka_scan_errors
sys_stream_scan_states	stll_kinesis_scan_states, stll_kafka_scan_states
sys_unload_detail	stll_unload_log, stll_user_query_map
sys_unload_history	stll_unload_history_base, stv_user_query_state

Provisioned Tests

Four Node Original Cluster

Nodes/Slices/Type

Node	Slice	Type
0	0	data
0	1	data
1	2	data
1	3	data

Node	Slice	Type
2	4	data
2	5	data
3	6	data
3	7	data

Slices Participating in Test Query

Slices
0, 1, 2, 3, 4, 5, 6, 7

Block Size of Test Table

Blocks
80

Resized Four->Two Node Cluster

Nodes/Slices/Type

Node	Slice	Type
0	0	data
0	1	data
0	6	data
0	7	data
1	2	data
1	3	data
1	4	data
1	5	data

Slices Participating in Test Query

Slices
0, 1, 2, 3

Two Node Original Cluster

Nodes/Slices/Type

Node	Slice	Type
0	0	data

Node	Slice	Type
0	1	data
1	2	data
1	3	data

Slices Participating in Test Query

Slices
0, 1, 2, 3

Block Size of Test Table

Blocks
40

Resized Two->Four Node Cluster

Nodes/Slices/Type

Node	Slice	Type
0	0	data
0	4	compute
1	2	data
1	5	compute
2	1	data
2	6	compute
3	3	data
3	7	compute

Slices Participating in Test Query

Slices
1, 3, 4, 5

Manual Tests

Provisioned

1. A two node `dc2.large` provisioned cluster was created.
2. The query `select '0100-01-01 BC'::date;` was issued.
3. The result was;

```
     date
---------------
 0100-01-01 BC
```

3. The query `create table table_1 as select 1;` is issued, to create a Redshift table.
4. The query `select '0100-01-01 BC'::date from table_1;` was issued.
5. The result was;

```
    date
------------
 -099-01-01
```

Serverless

1. A 1 NRPU Serverless workgroup was created.
2. The query `select '0100-01-01 BC'::date;` was issued.
3. The result was;

```
     date
---------------
 0100-01-01 BC
```

3. The query `create table table_1 as select 1;` is issued, to create a Redshift table.
4. The query `select '0100-01-01 BC'::date from table_1;` was issued.
5. The result was;

```
    date
------------
 -099-01-01
```

Discussion

Primary Evidence

Workgroups

Workgroups take some, and sometimes several, minutes to create and to delete, as far as is indicated by the status returned by `boto3`, but then also usually take a minute or so for a connection to actually be possible, which rather belies the `available` state - if I can't connect, the workgroup isn't very available. As such, when creating a workgroup, you cannot rely on the `available` status code returned by `boto3` - if you see that status, and then try to connect, normally your connection will fail. You need to loop on the connect attempt, until it succeeds.

The times in the test run given below are unusually lacking in shutdown duration outliers. Usually there has been a shutdown which is 600 or more seconds.

NRPU	Available	Connect	Shutdown
64	162.94810509681702	36.03853249549866	149.57489109039307
32	158.84450340270996	3.484558582305908	277.4090383052826
16	122.33415484428406	23.383538246154785	164.1367542743683
8	121.4271936416626	69.81401205062866	184.42388319969177
4	122.49102735519409	79.7402982711792	238.34082007408142
3	153.5724790096283	78.02273607254028	284.17035365104675
2	120.54271459579468	15.95941710472107	57.93810534477234
1	116.43005561828613	22.12782907485962	314.46830320358276

Workgroups likewise take some, and sometimes several, minutes to resize.

The time taken to create a workgroup is pretty consistent, but the time before a connection can be made, and the time taken to delete a workgroup, are quite variable.

The following table shows the startup and connect times for the seven 16 NRPU workgroups brought up for the resize tests. The shutdown time is not shown, because the workgroup has by then been resized, which makes those workgroups different to those in the table above (which have not been resized).

NRPU	Available	Connect
16	142.00040555000305	42.092533111572266
16	158.3465723991394	8.1436448097229
16	122.82265400886536	40.22297286987305
16	124.32334208488464	75.41635346412659
16	127.80255341529846	104.72378063201904
16	127.83186173439026	93.99888372421265
16	127.59397530555725	5.925532102584839

We can see here delay after `boto3` reports `available` and being able to actually connect, is quite variable.

When a workgroup is created, the computing power assigned to the workgroup is specified, in terms of Redshift Processing Units (RPUs), and this can only be changed by a workgroup resize, which takes the workgroup off-line.

Be absolutely clear - RPUs are *not* dynamic.

Resize, like creation, takes some minutes (the times below are in seconds), but the time taken to connect after a resize is very short.

NRPU	Available	Connect
64	221.70308017730713	1.1366949081420898
32	290.20202445983887	1.7749905586242676
8	307.84038615226746	2.2486279010772705
4	237.87361240386963	2.4608988761901855
3	223.4217014312744	1.3041656017303467
2	214.12547516822815	2.4654417037963867
1	261.23654770851135	1.3217377662658691

In this table, the original workgroup is 16 NRPU, and it is then resized to the size given in the `NRPU` column.

Resize times are pretty consistent, and connect times look to be almost immediate - but on fact, as we see later, there is a connection proxy in front of the cluster, and I think what's happening here is that the connection is so fast because it is to the proxy, where in fact the cluster isn't yet ready for queries, and if I had issued a query, it would have been tens of seconds (the usual durations for a connection to a workgroup being brought up from scratch) until the query ran.

Workgroups are wholly independent of each other; each workgroup has its own IP address, which users connect to and into which they issue queries, and each workgroup executes queries its own resources only.

Now, in a serverless system, we would - like Lambda - specify the compute required for a query and issue the query. We would have no knowledge of servers, only of the specific query we issued.

What we actually have, with Redshift Serverless, is a much coarser granularity; we specify the compute when we provision a workgroup, and queries issued to that workgroup get the compute we specified for that workgroup, and all queries into that workgroup must then in some way competing for the fixed compute in that workgroup, unless we imagine auto-scaling, which AWS vaguely allude to but otherwise provide no information.

Additionally, a namespace can be operated on by and only by a single workgroup at a time; so the compute available for a given namespace can be varied only by resizing the workgroup to which a namespace is currently assigned, or by shutting down the existing workgroup, and then assigning the namespace to another workgroup.

So; with Redshift Serverless, we create a workgroup, which takes some time, and at creation we specify its compute capability, which can only be changed by a resize, which also takes some time and takes the workgroup off-line, and when the workgroup is created we assign it a namespace (the data the workgroup works on), where only one workgroup can be assigned a given namespace at a time, and we issue queries into the workgroup, and that workgroup handles those queries using its own resources only, in the manner it sees fit. Deleting the workgroup also takes some time.

I am reminded of the saying - if it looks like a duck, swims like a duck, and quacks like a duck, then it probably is a duck; this looks awfully like a server.

Now on to the smoking weaponry.

Leader Node and Worker Node

This test result is a (but not *the*) smoking gun.

With provisioned Redshift, there are in a cluster a single leader node, and between 2 and 128 worker nodes (the maximum number varies by node type).

Users when they connect to a cluster are in fact connecting to the leader node, and as such users issue queries to the leader node, which orchestrates the work required to complete the queries, which typically involves distributing work to, and directing the work done, by the worker nodes.

The worker nodes return the results of their individual work to the leader node, which then performs any final processing, and then finally the leader node returns the results to the user.

The leader node began as Postgres 8 and retains what looks like all of the original functionality, plus now all the new functionality needed to operate worker nodes.

The worker nodes look to be a wholly different code base, not derived from Postgres, and quite minimal in their functionality, and although of course adhering to the SQL specification.

As such, there are implementation and behavioural differences between the leader node and the worker nodes.

One of these differences is a simple, easy and reliable way to differentiate between the leader node and worker nodes; leader nodes and worker nodes emit different output formats for dates prior to 1 AD.

This is because the leader node is as Postgres does using the AD/BC format, where there is no year 0 and dates are written with a positive number for years but a `BC` indicator for years prior to `1 AD`), e.g. `4000-01-01 BC`, but the worker nodes are instead using the ISO 8601 standard, where there *is* a year 0, and dates are written where years can be a negative number and there is no `BC` indicator, e.g. `-3999-01-01`.

When issuing a query to a provisioned Redshift cluster, it will by default run on and only on the leader node. A query runs on worker nodes when and only when the query uses functionality on the worker nodes, such as reading tables created by users.

As such, the following query as it uses no functionality on the worker nodes runs on and only on the leader node, the output here being from a two node `dc2.large` cluster, which we see producing the AD/BC format output expected of the leader node.

```
dev=# select '0100-01-01 BC'::date;
     date
---------------
 0100-01-01 BC
(1 row)
```

The next query uses a Redshift table created by a user, which as with all user created tables is on the worker nodes, and so this query runs on the worker nodes, and we see the output is now in the ISO 8601 format;

```
dev=# create table table_1 as select 1;
SELECT
dev=# select '0100-01-01 BC'::date from table_1;
    date
------------
 -099-01-01
(1 row)
```

Now here's *a* smoking gun - these two queries, issued on a Serverless workgroup, produce the same results as when issued on provisioned Redshift.

The following two queries were run, manually, on a 1 NRPU workgroup.

```
dev=# select '0100-01-01 BC'::date;
     date
---------------
 0100-01-01 BC
(1 row)
```

```
dev=# create table table_1 as select 1;
SELECT
dev=# select '0100-01-01 BC'::date from table_1;
    date
------------
 -099-01-01
(1 row)
```

Why on earth would a serverless Redshift behave in this way?

We could of course imagine some explanation for this, where the design of Redshift Serverless is such that each workgroup still has a leader node and worker nodes, and queries are routed between them in exactly the same way as a provisioned Redshift cluster, and also as we have already seen, that workgroups take time to create, have fixed compute resources, take time to resize and delete, work on a single namespace at a time only, and decide for themselves how to handle concurrent queries issued to them, and do so only with their own compute resource - *but* that somehow, despite all this, there is something, somewhere, which we have yet to detect, which makes workgroups serverless.

Or we can, much more simply, begin to imagine that a workgroup is an ordinary, normal Redshift cluster.

This is as I say, *a* smoking gun. It is not enough by itself, but it will not need to be.

Slices

Now we come to the smoking howitzer.

Introductory Concepts To begin with, I need to explain a certain number of fundamental Redshift concepts, so the test results can be explained and understood.

In Provisioned Redshift, a cluster is composed of nodes, where a node is a virtual machine - an EC2 instance (although I think of specifications not available to normal users).

A provisioned cluster contains one and only one *leader node*, and two or more *worker nodes* (actually, you can have a one node cluster, but don't - they're strange things and should not be used).

The leader node runs the show, and the worker nodes do most of the heavy lifting.

The leader node receives all queries, returns all query results, and organizes and directs the work of the worker nodes.

The leader node began as Postgres 8 and retains what looks like all of the original functionality, plus now all the new functionality needed to operate worker nodes.

The worker nodes look to be a wholly different code base, not derived from Postgres, and quite minimal in their functionality, and although of course adhering to the SQL specification.

As such, there are implementation and behavioural differences between the leader node and the worker nodes.

When issuing a query to a provisioned Redshift cluster, it will by default run on and only on the leader node. A query runs on worker nodes when and only when the query uses functionality on the worker nodes, such as reading tables created by users.

When worker nodes are required (for example, the query uses data from tables created by users, for those tables are stored on worker nodes) the query is converted to multiple C++ files, each of which is compiled, each of which produces an executable, and those executables are distributed to each worker node, and each node runs multiple copies of the full set of executables - each running copy of the full set of executables is known as a *slice* - and each slice owns a portion of each table and performs its work on its portion of each table.

Different worker node types have different numbers of slices.

A cluster is configured by its admin to run a maximum number of queries concurrently - ten is usually a good choice. (To give perspective, the absolute maximum is 50, which would be a terribly bad idea, as each query would have far too little memory and you would have staggeringly large amounts of swapping to disk.)

So, bringing this together, if we have a 4 node cluster which is configured to allow 10 concurrent queries, and each query has say 5 executables, and there are 4 slices on a node, a node will be running 10 * 5 * 4 = 200 processes per node, so 800 in total, at any one time.

Importantly, as we shall see, so keep it in mind, slices are sequentially numbered, from 0 to the total number of slices; a cluster with 4 nodes of 2 slices has slices 0 to 7.

Now, originally, only the slice which stored a row could access that row. These days, any slice on a node can access the data stored by other slices.

This new found flexibility on the face of it seems at times to be improperly used. It's not uncommon to see a slice process no rows at all, while another slice processes twice as many, which to my eye looks like a mistake. In any event, I will say, Redshift under the hood has a lot of variability, once you start looking in detail at what's going on. Normally users have no idea or awareness of this; they issue queries, results come out. That the query might have executed twice as quickly is not something which is apparent.

Variability in slice use is as we shall see important, so keep that in the back of your mind, too.

Now, in Serverless Redshift, it would be interesting to see if nodes and slices are still in use, and if so, how they are being used.

The normal way in which I would look to see what's going on under the hood is to examine the system tables - a very large group of tables and views, in the `pg_catalog` schema, which contain information about the workgroup or cluster.

AWS have however in Serverless removed the large majority of system tables, censored the SQL of the views which do remain (so you cannot view the SQL of the view), and have specifically removed from the views all columns which carry information about nodes and slices.

You might think that this has been done because nodes and slices no longer exist, but these columns in fact still exist in the underlying tables the views are using; they've been removed from the *views* only.

(Both admin and normal users do not have permission to access rows in the underlying tables, but the columns in those tables can be seen from the system tables - that's a Postgres behaviour, and I think is impossible to prevent.)

So, given the censoring of the system tables, I need to find side-channels by which to gather such information as I can.

Fortunately, AWS appear to have overlooked a function in Redshift, `[slice_num()]`.

The docs assert this function returns the slice number a row is held upon. I think this description was probably always wrong, and in fact the function always returned the slice number of the slice which *processes* a row, but back in the day, the only slice which could process a row was the slice which held the row on disk. These days that's no longer true, so the slice number can vary over different runs of the same query, as slices can now read rows held by other slices.

As such `slice_num()` is an imprecise tool, but it does give us *some* sort of insight into slices - such as whether they even exist - and the mere existence of slices raises serious questions, because slices are so fundamental a concept integral to provisioned Redshift; slices run on nodes, nodes form a cluster.

Moving on from explaining about slices, we come to the next three introductory concepts, which are closely related and so presented together; elastic resize, compute slices, and data slices.

Historically, provisioned Redshift of course allowed the cluster to be resized and to do so offered what is now known as classic resize. With this resize, a new cluster is created with the new number of nodes, each naturally with the number of slices indicates in its specification, and every row from the existing cluster is copied over to, and redistributed upon, the new cluster. This takes forever, but you then have a new cluster with none of the (many) drawbacks of elastic resize.

Where classic resize takes forever, elastic resize was introduced.

Elastic resize changes the number of nodes, but does not change the number of slices.

So, nodes are added to, or removed from, the existing cluster, and the existing slices are redistributed *en bloc* over the new set of nodes, and this is very fast - it's just a bulk copy operation of data from S3, not a row-by-row redistribution of data.

(You can then end up with different numbers of slices on each node, which is not ideal, because nodes with more slices complete queries more slowly, and a query runs as slowly as the slowest

slice - but there's some nuance here about the exact mechanism by which performance suffers, which I'll get to shortly.)

With classic resize, you can move from any number of nodes, to any other number of nodes (assuming you have enough storage for your data, of course).

With elastic resize, there are limit on how far you can move from your original number of nodes, because elastic resize does not change the number of slices - it only redistributes the slices you have; so obviously you cannot have less than one slice per node, nor are you allowed to have too many slices per node.

A node however no matter how many slices it holds always runs and only runs the number of slices which are given in its hardware specification.

So if you have a node which by its specification has 4 slices, and due to an elastic resize it now holds 8, in fact only 4 slices will actually run (remembering here that a slice is a full set of the executables for a query).

I've not looked closely, but what must be happening is that each slice is reading the data which belongs to *two* slices, which halves performance, which is what you'd generally expect given you now have half as many nodes.

This makes good sense, because if you double the memory pressure on a node, by running twice as many slices, you are likely to cause a very great deal of swapping to disk and performance will fall off a cliff-edge.

Better to read two lots of data *without* swapping, than have twice as many slices and each reading one lot of data.

We can see now from this how having uneven numbers of slices on nodes causes a performance problem; one or more nodes in a cluster, those which have ended up with more slices than the other nodes, will find their slices need to read more slices worth of data, than the other nodes.

Queries run as quickly as the slowest slice, because a query cannot complete until all slices participating in the query have finished their part of the work for that query (and normally all slices participate in all queries), so any imbalance in node performance renders a cluster as slow as slowest node.

Also, there are efficiency losses in reading multiple lots of data; I've not investigated this, so this is my expectation based on my understanding of Redshift, but broadly speaking, all other things being equal, every doubling of the number of slices worth of data doubles the number of disk seeks.

Small nodes types are allowed at most a 2x resize (so to half the number, or double the number, of the original number of nodes), and large node types a 4x resize (so to a quarter of, or up to quadruple, the number of nodes of the original cluster).

So in the worse case, you're looking at 4x disk seeks.

By and large, although I've not investigated and with Redshift you know nothing until you've actually looked, I'm not particularly worried about this increase in the number of disk seeks - Redshift is intended for Big Data, and so uses min-max culling (aka "the Zone Map") and has no support for indexes. Correctly operated Redshift is not a seek-heavy database.

(This contrasts with Redshift when it is operated incorrectly, and you're using hash joins without enough memory, and so start swapping heavily, because that *is* seek heavy.)

This leads us to the final introductory concept, that there are these days two types of slice, what AWS call *compute slices* and *data slices*.

Until fairly recently, there was only one type of slice, which is the type now known as a data slice.

A data slice is the standard, full-fat slice. It holds data on disk, and it participates in all step types (a step is the atomic operation of work in Redshift, there's about forty of them - such as scanning a table, aggregating, reading from the network, etc).

After some time, and I think (I've not specifically checked, as I never use elastic resize) only on `ra3` node types, a second type of slice was introduced - what AWS call a *compute slice*, with the original slice now being termed a *data slice*.

A compute slice is a *filler*. It does *not* hold data on disk (but it can still read data held by data slices on the same node), and last I heard (and that's quite recently) participates in only some (I think only a few, but I've not investigated) types of step.

When a cluster is first made, all slices are data slices, and every node holds the number of slices as given in its specification.

When an elastic resize occurs and the cluster becomes smaller than the original size, those data slices are crammed onto the smaller number of nodes, but they all are still data slices, and so there are only data slices in the cluster.

However, when an elastic resize occurs and the number of nodes becomes larger than the original number of nodes in the cluster, then the number of slices on each node will be less than the specified number of slices for the node type.

In that situation, what happens then these days is that the missing slices are filled in by compute slices. Where compute slices have no data associated with them, the cluster can just spin them up at will - no need to move data about.

Compute slices are better than having no slice at all, but I think (not investigated, only reliable rumours, so no promises yet) they're a long way from being a real slice.

I've put a test, demonstrating data and compute slices on a provisioned cluster, in the test script, and we can here examine the output of that test.

First, we make a two node cluster, of `ra3.xplus`, which has two slices per node, and we examine the nodes, slices and slice types, and we get as expected two data slices per node.

Node	Slice	Type
0	0	data
0	1	data
1	2	data
1	3	data

Then we resize to four nodes. Now we see this (the four data slices are now distributed over four nodes, and the compute slices have filled the gaps);

Node	Slice	Type
0	0	data
0	4	compute
1	2	data
1	5	compute
2	1	data
2	6	compute
3	3	data
3	7	compute

But if we make a four node cluster in the first place, no resize, we get this (two data slices per node);

Node	Slice	Type
0	0	data
0	1	data
1	2	data
1	3	data
2	4	data
2	5	data
3	6	data
3	7	data

And just for completeness, if we resize a four node cluster to two nodes, we get this - same number of data slices, but now only two nodes;

Node	Slice	Type
0	0	data
0	1	data
0	6	data
0	7	data
1	2	data
1	3	data
1	4	data
1	5	data

We're now onto the final introductory concept, how data is stored on disk, and this is important, because as we will see, it provides a second and independent method of proof which corroborates the findings from `slice_num()`.

In Redshift, all disk I/O is in one megabyte blocks - which I will refer to simply as *blocks*.

You *cannot* write less than one block, no matter what you're doing. It's *always* one block or more, and I/O operations are conducted accordingly.

So for example when you write a single row, you're actually writing one block - with a single row in.

Additionally, remember that Redshift is column-store; so each column is, in effect, stored in its own file (not strictly how it's actually managed, but the exact implementation isn't needed here and what I'll write about it is accurate in terms of what's going on with blocks).

So when you write a single row, you're actually writing one block *per column*, because each column is a separate file, and all disk I/O is in blocks.

Finally, remember that Redshift is a sorted database, and so every table in fact consists of two *segments*, a sorted segment and an unsorted segment. When new rows are added, to go into the unsorted segment. The `VACUUM` command converts unsorted rows into sorted rows (and so converts unsorted blocks into sorted blocks).

Now, when a table is completely new and empty, it has no blocks on disk *at all*, but as soon as you write the first row, you get one block *per segment*, even though one of the segments is empty, and that's also one block for each segment *per column*, because the columns each have their own file.

You can sense already this is all going to become quite a lot of blocks, right? :-)

(In fact, the first addition of rows to a completely new, or freshly truncated, table goes into the sorted segment. This is because when rows are added, which is by `INSERT` or `COPY`, the rows added are sorted with respect to themselves, and then added to the table. If the table is wholly new or freshly truncated, those new rows become the sorted segment, and an empty unsorted segment is created. However, if a sorted segment already exists, then the new rows are added to the unsorted segment.)

So where does this leave us?

Well, let's imagine we have an 8 node cluster, with 4 slices per node, and a 20 column table, and in a single insert, we write one row per slice.

So that's 8 * 4 = 32 slices.

Each slice has 20 columns, so that's 32 slices * 20 columns = 640 files.

The table is brand new, so all the rows go into the sorted segment, and an empty unsorted segment is created.

So that's two segments per file, so that's 640 files * 2 segments = 1280 blocks.

So that's 1.28 *gigabytes* of disk, to store 32 rows.

Aside from just the simple amount of store used, we also have to think about the amount of disk I/O which is going on. Reading or writing those 32 rows used up something like a full third of a second of the entire disk I/O of a modern SSD - start doing lots of that on a busy cluster and see where it gets you.

Now, at first glance, this may seem completely crazy, but it *really* isn't.

You have to remember Redshift is a clustered, sorted, column-store database.

Redshift is designed for Big Data - it has taken all the design choices which are absolutely right and correct for working with Big Data - but these choices are death and ruin if you're working with small data.

What's widely unperceived is that with relational databases, Big Data and small data are *mutually exclusive*.

A Big Data relational database is not a sort of souped-up, super-sized version of a normal relational database; it's a *completely* different beast, and in particular it can efficiently handle Big Data and *only* Big Data.

Being able to handle Big Data does *not* mean you can handle small data.

If you try to use Redshift with small data, well, what happens is this : if your data is small enough, then the power of the hardware overwhelms the data and you can do anything you like and it all works just fine anyway.

Your queries are running literally a thousand times slower than they could, but it just doesn't matter, because they all run in a second or two or maybe a minute or two, or if you're doing lots of `INSERTS` maybe an hour or so (as opposed to running in, say, ten seconds).

In this scenario, you've loosing out because you don't need Redshift, so you could be using say Postgres, and Postgres has a lot, *lot* more functionality than Redshift - Redshift is a *modernized* database, not a *modern* database; and also because even with such tiny data, there are still ways with Redshift you can trip up over your own feet, such as query compilation times, which make BI application use problematic even with tiny data, or with disk use overheads, as discussed.

Redshift offers essentially three methods by which it allows data to be processed efficiently; clustering, sorting, and column-store. These methods all need to be operated correctly to be effective, and the correct operation of these methods imposes constraints and restrictions upon system and data design, which must be honoured, or these methods do not work.

As your data becomes larger, and where Redshift is being incorrectly operated and so you've not thought about any of the design considerations necessary to use clustering, sorting and column-store correctly, and so they are not in effect, you start to have enough data that the hardware no longer overwhelms the data, and now performance is becoming awful, and you are now well and truly dead in the water because you will need to redesign your entire system from the ground up, to use clustering, sorting and column-store correctly.

It's a big, nasty, deadly booby-trap and the killer problem here is that I have never seen AWS *ever* tell a client Redshift is inappropriate for their use case, or even that any of these issues exist. I don't think the TAMs actually know.

When your cluster starts to slow down the response I usually here - the only response available to them, given what I suspect is lack of knowledge, and that the entire system would need to be redeveloped to use clustering, sorting and column-store correctly, is that *you probably need to buy more nodes.*

This is also good for AWS's bottom line. Better than *use sorting correctly and halve your cluster size.*

A problem here is that AWS is lack a clustered, *unsorted*, column-store relational database - which is basically what Snowflake is, and to my eye is one of the reasons Snowflake is cleaning up (well, that and Snowflake has plenty of modern SQL functionality, excellent docs and actual, genuinely meaningful Support - I am of the view Redshift lacks all three).

Clustering is pretty easy to use correctly (but it costs money to add nodes, and not quite all functionality scales as you add more nodes - for example, you still have just the one leader node). Column-store also; there's not much devs and users need to know. Sorting is a living nightmare to get right, because it's extremely knowledge intensive, for all of the admin, devs *and* users who issue queries, and imposes no end of constraints on what you can do with your queries - there

are a ton of use cases which simply *can't* use sorting, and that's a shame, because sorting is free and *mind-bendingly* efficient - which means you need *staggeringly* less hardware to get the same performance level compared to an unsorted database.

So, given the knowledge requirements, what you find in practise is that the vast majority of clients are absolutely not in and will never be in a position to use sorting, but in fact they don't *need* sorting, because clustering and column-store would do just fine for the amount of data they have.

If you're using Redshift now, and you know you're using sorting, and you know your users are using sorting correctly, and you're running your own `VACUUM`s and you're aware of the producer-consumer scenario with `VACUUM` which is one of the ultimate capacity limits for a cluster, then you're good.

If you *don't* know you're using sorting, and you're not running your own `VACUUM`s, and either your admin, your devs or your users don't know what sorting means for the table design and queries - it makes no sense for you to be on Redshift.

(Note that everything you might be told by a TAM about auto-vacuum, and clusters managing themselves, and so on, infuriates me - it is in my view pure marketing material, and is in no way actual, effective functionality. In my experience, looking at clusters, and talking to other Redshift admin, and my own investigation into auto-vacuum, this stuff simply has no meaningful effect.)

So, that's been a long aside - which originally all started as an explanation of how data is stored on disk. Let's get back to that, because there's a little bit more to say.

We've seen then all disk I/O is in one megabyte blocks.

What we need, for this knowledge to be useful, is a way to know how many blocks are being used by a table.

This information is available normally though a system table, `STV_BLOCKLIST`, but that system table of course on Serverless is now inaccessible.

However, in the `SVV` system tables, there is a system table `SVV_TABLE_INFO`, which has one row per table and shows information about that table.

Normally I steer a mile wide of this system table view because the code is huge and incredibly complicated, I expect it to be buggy, the view has behaviours which are not helpful (such as not showing empty tables), the data it does show is a hodge-podge of the weird and useless, I utterly disagree with how it computes some of its values (in particular, `unsorted`, which is the percentage of how many rows are unsorted - this is being computed by taking the total number, across all slices, of unsorted rows and dividing it by the total number of rows, which is wrong - a query is as fast as the slowest slice, so this value needs to be computed *per-slice*, with the most unsorted slice being how unsorted the table is), and so on.

However, since almost all the system tables have been made inaccessible, we have to use what we can get, and in this view there is a column called `size`, and it is supposed to be the number of blocks used by the table.

That this column continues to exist in Serverless, given the censorship of the system tables, is extremely surprising.

I suspect it remains because if it were to be removed, the view code would need to be different for provisioned and Serverless, and that can't be done.

However, it might be perhaps it was just overlooked, or considered harmless.

I can on a provisioned cluster obtain the source code for this view (I can't on Serverless, it's censored, as described before) and looking at how the `size` column is computed I am not filled with joy.

In the end it comes down to values being read directly from a table which does not contain raw data as such (`STV_BLOCKLIST` has one row per block, and I trust it), but rather contains *computed* data, one row per table, which simply says how many blocks the table has.

So I have no idea how that data is being computed.

Well, it's still all there is, but as it turns out, it does seem to be producing exactly the values I would compute myself, so I think it is correct.

Where we know *a priori* how tables are working under the hood, at least in provisioned Redshift, we can predict for the tests exactly how many blocks should exist, and then we can check, using *size*, if that is in fact the number of blocks we find.

So, that's all the explanations. On to the good stuff.

Elastic Resize A workgroup specifies its compute in terms of Redshift Processing Units (RPUs), which originally ranged from 32 to 512, in units of 8.

I say originally, because after some time, 8, 16 and 24 RPU workgroups were introduced. I think this was done because a 32 RPU workgroup is already expensive (11.52 USD per query-hour), and this was felt to be hindering adoption.

I suspect these smaller workgroups are different to the original workgroups, and so would muddy the waters, and so to begin with I'm going to focus on the original 4 to 64 NRPU workgroups. I will return to the smaller workgroups once the original workgroups have been dealt with.

That form of numbering, 32 to 512 in steps of 8, is strange and curious, and as we shall see in fact carries hidden meaning.

If we normalize RPUs, using units of 1 rather than 8, it's really the range 4 to 64, which I call Normalized Redshift Processing Units (NRPU).

The official docs on the page *Compute capacity for Amazon Redshift Serverless*, provide the one and only one documented fact about RPUs, which states than 1 RPU has 16 GB of memory, which means 1 NRPU, where it is 8 RPU, has 128 GB of memory.

So, if the docs are correct, we're looking at workgroups ranging from 0.5 TB to 8 TB of memory.

Now, here's a thing - the *default* workgroup size is 16 NRPU - that's 2 TB of memory.

That's a *lot* for a default.

Why so large a default?

What immediately caught my eye is that the default, 16 NRPU, is exactly halfway, in terms of elastic resize, between the smallest and largest workgroups. It's four times larger than a 4 NRPU workgroup, and 4 times smaller than a 64 NRPU workgroup.

Spooky, huh?

(Looking at the official doc page *Overview of managing clusters in Amazon Redshift*, we find the maximum elastic resize permitted for `ra3.4xlarge` and `ra3.16xlarge` is indeed 4x. We also see on the page *Amazon Redshift clusters*, that `ra3.4xlarge` has 96GB of memory, while `ra3.16xlarge` has 384GB of memory, with 1 NRPU as we've seen, if the docs are correct, having 128 GB of memory, and so being in the right ball park.)

In the test script, I create a test table named `table_slices`, with key distribution and two `raw` encoding `int8` columns, where the first column is an identity column, starting at 1 and incrementing by 1 and which is also the distribution key.

1024 rows are written to the table, which being the single insert after table create will automatically be sorted, and then the following query is issued;

```
select size from svv_table_info where "table" = 'table_slices';
```

This tells me the number of blocks in use by the table.

Given the tiny number of rows (one block of `raw int8` holds about 260k rows), I am expecting one block per slice, per column, per segment.

(One important detail here is that every table has three hidden columns, which belong to Redshift, which are used for MVCC. So although I defined the table with two columns, in fact, it has five columns. Of the three hidden columns, one behaves normally, the other two behave oddly, here where I say behave I mean in terms of the number of blocks they consume; but where the number of rows is so small, their behaviour will not matter - they will all use up a single block per slice, per column per segment.)

After the first insert query with its 1024 rows, and the first test query which returns the number of blocks in use, I insert to the table using a self-join, with a `LIMIT` of 100,000 rows, the following query;

```
select distinct slice_num() as slice from table_slices order by slice;
```

I add the extra rows to get a decent number of rows on every slice, because Redshift is pretty variable when it comes to deciding what slices to use, and when a slice only has a few rows, it is often not used, with some other slice reading those rows. Getting a decent number of rows on each slice made them more likely to be used, which allows us to see them via `slice_num()`.

The query above, the `slice_num()` query, will tell me firstly if slices are in use at all, and if so, which slices participate in the test query, where the test query is simply reading the rows from the test table.

So, the first finding was it looks like there are indeed slices in a workgroup.

For example, here's the output of the query, the list of participating slices, for a 16 NRPU workgroup;

Slices
0, 1, 3, 5, 6, 8, 9, 10, 11, 12, 13, 14, 16, 17, 20, 21, 22, 26, 27, 28, 29, 30, 31, 33, 35, 36, 37, 38, 39, 40, 42, 43, 46, 47, 50, 51, 52, 54, 55, 58, 59, 61, 62, 63, 64, 66, 67, 68, 69, 71, 72, 73, 74, 75, 76, 77, 79, 80, 82, 83, 84, 85, 86, 87, 88, 91, 93, 94, 95, 96, 97, 99, 100, 102, 103, 104, 105, 106, 107, 108, 109, 110, 111, 113, 114, 115, 116, 117, 118, 119, 121, 122, 125, 126, 127

We see here about 128 slices, the range is from 0 to 127, and the numbers barring the occasional missing slice are contiguous. Redshift is always a bit variable in slice use, so the lack of a few slices (such as 2 and 53), is expected - it would be odd and remarkable if this did *not* happen.

Here's the same list of participating slices, but from an 8 NRPU workgroup;

Slices
0, 1, 2, 5, 6, 7, 16, 17, 18, 19, 20, 21, 22, 23, 32, 33, 34, 35, 38, 39, 48, 49, 50, 51, 52, 53, 54, 55, 64, 65, 66, 67, 68, 69, 70, 71, 80, 81, 82, 83, 84, 85, 86, 96, 97, 98, 99, 100, 101, 102, 103, 112, 113, 115, 116, 117, 118, 119

The number of NRPU has halved, and the number of slices has pretty closely halved.

However, the maximum slice number, 119, remains close to 128, and looking at the numbers we see distinct groups of numbers (0 to 7, 16 to 23, 32 to 39, and so on) - the slice numbers are pretty much contiguous in their groups, but the groups have gaps between them.

That's extremely important, as we shall shortly see, so keep it in the back of your mind.

Now, here's the table which presents the critical, central information in this investigation : for each workgroup (remember to begin with, I'm looking at 4 to 64 NRPU), we see the number of slices which participated in the query, and of those slices, the smallest and the largest slice number.

NRPU	# slices	min	max
64	96	2	503
32	106	0	254
16	95	0	127
8	58	0	119
4	28	0	101

Amazing, isn't it! Right now, I kid you not, I am writing with a French accent and twirling the ends of my little moustache.

To begin with, let's look at the number of slices participating in the test query (`# slices`).

First, we need to remember as mentioned before Redshift is fairly variable when it comes to slice recruitment, so some differences are always expected, and bearing this in mind, I would say for the 16, 32, and 64 NRPU workgroups the maximum possible number of recruited slices is 128 (but fell short where Redshift is unpredictable in its slice recruitment), and for 8 NRPU this number is 64, and 4 NRPU this number 32.

(Note I'm carefully saying "maximum possible number of recruited slices". There's going to be quite a bit of detail to come about slices, and that statement is all that can correctly be said at this point.)

As such, we see that the default workgroup size, 16 NRPU, can recruit up to 128 slices, and when we halve the number of NRPU, we halve the number of recruitable slices (and we see this twice, as we halve once to get to 8 NRPU, and halve again to get to 4 NRPU).

However, this is *not* the behaviour we see as we double the number of NRPU, when we go to 32 and then 64 NRPU. When we double the number of NRPU, the number of recruitable slices remains the same.

We'll be coming back to this shortly.

Turning now to the maximum slice number, remembering that slices are numbered from contiguously from 0, the maximum slice number gives us a pretty good idea of the total number of slices in the workgroup.

We can see that the 16 NRPU workgroup looks to have 128 slices (0 to 127), that when we then double the number of NRPU the number of slices doubles (32 NRPU has 256 slices, 64 NRPU has 512), but that if we instead halve the number of NRPU, the number of slices does *not* halve - the number goes down a bit but still basically looks like 128.

How exactly peculiar.

How very much like… *elastic resize*.

With elastic resize, when the cluster is enlarged, we have more nodes and more slices (and they're all compute slices), but when the cluster is reduced in size, the number of slices remains that of the original cluster - and this is exactly what we see here.

What's seen is exactly what would be expected if all workgroups begin as 16 NRPU, but are then elastic resized to the size the user selected when creating the workgroup.

This would also neatly explain why the default cluster size is so large, and why it is the mid-way point in a 4x elastic resize range.

So, what would be happening is that when the user creates a workgroup, it begins as 16 NRPU cluster, and this cluster has its normal complement of slices, and they are all data slices.

When the cluster size is doubled, or quadrupled (to 32 and 64 NRPU, respectively) the number of slices likewise doubles or quadruples (where these new slices are compute slices, not data slices).

Any slice, data or compute, can read data from any other slice, but it is only the original data slices which actually *have* any data, so the maximum number of slices which *can* participate in the query remains the same as the number of data slices in the original cluster (because you can't have two slices reading the data from one slice).

The reason for the weird variability for 16, 32 and 64 NRPU workgroups in the number of slices participating in the query is just Redshift being Redshift; to my eye it looks like Redshift is making the wrong choices and so is running the query inefficiently, but I'm *used* to seeing this kind of behaviour in provisioned Redshift - it's entirely common in a query for some slice to do no work, while others do twice as much (and this on idle clusters, so it's not load balancing).

If I saw here that the slices were consistently all being used, it would be a signal that there *is* something new in Serverless, and it's not just an ordinary cluster, because then we would be looking at behaviour which really is different to an ordinary provisioned cluster.

Now, moving on to where the cluster size is halved, or quartered, the number of nodes is reduced but the number of slices remains the same, and they are all data slices, crammed onto fewer and fewer nodes.

Redshift always runs only the number of slices given in the specification of the node type, so we do in fact see a halving and the a quartering of the number of slices participating in the query

- only now, they are reading data from not one slice, but from two or four slices, respectively (well, barring the natural variability in slice behaviour in Redshift).

Where any slice, data or compute, can read data, and where as part of an elastic resize data slices are distributed as evenly as possible over the new number of nodes, we expect all nodes to have some data slices (and indeed, when the cluster has been made smaller than the original cluster, all slices on all nodes are data slices) and so all nodes will have some slices which have data and so all nodes will have some slices participating in the query.

Slices (regardless of whether they are data or compute) are numbered, over the cluster as a whole, contiguously from 0, so slices numbers range from 0 to the number of slices minus one.

As such, the maximum slice number we find participating in the test query is telling us quite accurately the total number of slices in the workgroup, as all nodes have slices participating in the query, with the highest numbered node containing the highest numbered slices.

The original cluster, 16 NRPU, has a maximum slice number of 127, by which we can reasonably guess there are 128 slices in the cluster.

When the cluster is doubled, or quadrupled in the size, we see the highest slice number also closely doubles, or quadruples.

When the original 16 NRPU cluster is made smaller, and is halved, or quartered in size, the number of nodes is halved, or quartered, but the number of slices remains the same as on the original cluster. What happens is that all those slices, and they are all data slices, are crammed onto the smaller number of nodes now available.

In this situation, where there are more slices than there should be in the node specification, Redshift will run on each node only the number of slices specified by the node specification, where running slice now reads more than one slice's worth of data.

As such, there will be running slices on every node, but the number of running slices will now be much less than the number of data slices actually on the node, and Redshift can be seen to strongly tend to run the lowest numbered slices on each node.

As such, in this case, where the cluster has been made smaller, the maximum slice number is expected to be slice number of the highest numbered running slice, where the running slices will tend strongly to be the smallest slice numbers on the node.

This is why we see that the 16 NRPU has a maximum of 127, the 8 NRPU has a maximum of 119, the 4 NRPU has a maximum of 101. The number of slices on each node is becoming larger and larger, as the cluster becomes smaller and smaller, but the number of running slices is always the same, and are always strongly tending to be the lowest numbered slices on a node. As such, as the cluster becomes smaller, we see a moderately smaller maximum slice number.

We can see this behaviour directly by examining the list of participating slice numbers for each workgroup size.

Let's begin with 4 NRPU, where we see the following slice numbers participate in the query.

Slices
0, 1, 2, 4, 5, 6, 7, 32, 33, 34, 35, 36, 37, 38, 39, 64, 65, 66, 67, 68, 69, 70, 71, 96, 97, 99, 100, 101

This breaks down clearly into 4 groups of slice numbers;

```
0, 1, 2, 4, 5, 6, 7
32, 33, 34, 35, 36, 37, 38, 39
64, 65, 66, 67, 68, 69, 70, 71
96, 97, 99, 100, 101
```

There are eight slices in each group except of the final group, which has six (but in previous test runs that was usually also eight slices - what we see in the data here is Redshift's usual variability), but now look at the *first* slice number from each group - an incredible 0, 32, 64 and 96!

Moreover, we also see in each of the first three groups, we're getting contiguous consecutive numbers, where Redshift strongly tends to run the lowest numbered slices on a node.

The 16 NRPU workgroup has 128 slices (we can see this directly from the highest slice number being 127 and the practically contiguous list of slice numbers participating in the test query).

A 4 NRPU workgroup after an elastic resize will still have 128 slices, but all on only a quarter as many nodes, which means 32 slices per node.

This is why the first slice numbers of each group of slice numbers, which we saw above are 0, 32, 64 and 98, are a multiple of 32 - each node has 32 slices, but is running only 8 slices, and Redshift is using the lowest numbered slices on each node.

We can cross-check this by examining the data from the `size` column from `SVV_TABLE_INFO`.

The test table has 5 columns and 2 segments, and we predict 32 slices (all data slices) per node on 4 nodes, which means the prediction is for 5 * 2 * 32 * 4 = 1280 blocks of disk for the test table - and this is exactly what is found.

NRPU	# blocks	∴ # slices
4	1280	128

I assert then on the basis of the evidence so far that a 16 NRPU workgroup is an ordinary, normal 16 node Redshift cluster, each node with 8 slices, and that this node type originally (when the workgroup limits were 4 to 64) had a 4x elastic resize.

Now, remember that 1 NRPU is really 8 RPU?

A 16 NRPU workgroup is really a 128 RPU workgroup - and we see now it has 128 slices.

1 RPU is 1 slice.

Let's repeat this examination now on the 8 NRPU workgroup, and see if it holds up.

We see the following participating slice numbers;

Slices
0, 1, 2, 5, 6, 7, 16, 17, 18, 19, 20, 21, 22, 23, 32, 33, 34, 35, 38, 39, 48, 49, 50, 51, 52, 53, 54, 55, 64, 65, 66, 67, 68, 69, 70, 71, 80, 81, 82, 83, 84, 85, 86, 96, 97, 98, 99, 100, 101, 102, 103, 112, 113, 115, 116, 117, 118, 119

This breaks down into 8 groups of slice numbers;

```
0, 1, 2, 5, 6, 7
16, 17, 18, 19, 20, 21, 22, 23
32, 33, 34, 35, 38, 39
48, 49, 50, 51, 52, 53, 54, 55
64, 65, 66, 67, 68, 69, 70, 71
80, 81, 82, 83, 84, 85, 86
96, 97, 98, 99, 100, 101, 102, 103
112, 113, 115, 116, 117, 118, 119
```

With between 6 and 8 slices participating on each node (Redshift's usual variability).

Again, note the first slice numbers in each group : 0, 16, 32, 48, 64, 80, 96, 112 - there are now 16 slices on each node, which is what we would predict; half as many as for 16 NRPU, twice as many as for 4 NRPU.

We can again look at the `size` information from `SVV_TABLE_INFO`.

The test table has 5 columns and 2 segments, and we predict 16 slices (all data slices) per node on 8 nodes, which means the prediction is for 5 * 2 * 16 * 8 = 1280 blocks of disk for the test table - and this is exactly what is found.

NRPU	# blocks	∴ # data slices
8	1280	128

The 8 NRPU workgroup is behaving exactly like the 16 NRPU workgroup, and if you look in the test results at the 4 NRPU workgroup, you see the same again.

Now, I must point out that cramming many slices into a node is not without consequence, which is to say, inefficiency.

The first consequence, is that leads to *considerable* - I might even say *vast* - additional overhead in disk use.

The original 16 node cluster, has 128 slices - it's a big cluster. There's plenty of overhead - the test table, with five columns, if we write a single row to each slice, consumes 1280mb of disk - but this is what you get when you use Big Data methods and it's in line with the size of the cluster.

If we by contrast span up a cluster with 4 nodes in the first place, each with 8 slices, so no elastic resize, we would have 32 data slices rather than 128, and so the test table would consume 5 columns * 2 segments * 4 nodes * 8 slices per node = 320mb of disk.

By performing the elastic resize, the smaller clusters are all forced to have the disk overheads of the 16 node cluster, rather than the much smaller overheads they would have if the cluster were originally brought up at their smaller size.

This all turns up in your RMS bill, and in the performance of your queries where they must read so much more disk.

RMS is however extremely cheap - 24.576 USD per terabyte per month - but query time on Serverless is not cheap.

This problem is primarily an issue with small tables. When a table is large, the amount of data overwhelms or massively overwhelms the amount of overhead. However, there can be a lot of small tables in a cluster.

I once saved a customer 20% disk space by by converting all their small tables to being unsorted, which mean they had one segment only; that was with `dc2.8xlarge`, and it saves them buying three nodes (about 130k USD per year).

The second consequence, as we've seen, Redshift is only running the specification number of slices; so the data assigned to all those other data slices is on the node, but those slices are not running.

So what happens to that data? I've not investigated, but I think the slices that are running are processing it *serially*. Each slice processes its own data, and then the data of one (8 NRPU) or three (4 NRPU) other slices.

The amount of data remains the same, but it's now shared out over many slices. Broadly speaking, all other matters being equal, I think this means the amount of disk seek is multiplied by the number of slices worth of data being read.

Normally disk seek is death and ruin to performance, but in Redshift all disk I/O is in one megabyte blocks, which helps and I think strongly to minimize the impact of this issue. Nevertheless, it is a cost, and I've also not investigated it, and with Redshift, until you investigate, you don't know; there are often surprises, not least in that you often find what's being done is not what you'd expect to be being done.

Now, I said at the beginning I would return to the more recently introduced small workgroups, the 1, 2 and 3 NRPU workgroups.

Something interesting and different must be going on here, because these sizes are outside of the 4x elastic resize range of the 16 node cluster.

Let's start with the information about slices.

NRPU	# slices	min	max
3	24	0	110
2	16	0	99
1	8	0	67

These are the participating slice numbers;

3 NRPU Slices
0, 1, 2, 3, 22, 23, 24, 25, 44, 45, 46, 47, 65, 66, 67, 68, 86, 87, 88, 89, 107, 108, 109, 110

2 NRPU Slices
0, 1, 2, 3, 32, 33, 34, 35, 64, 65, 66, 67, 96, 97, 98, 99

1 NRPU Slices
0, 1, 2, 3, 64, 65, 66, 67

The 3 NRPU workgroup has six groups of slices (which means six nodes), the 2 NRPU workgroup has four groups of slices and so four nodes, the 1 NRPU workgroup has two group of slices and so two nodes.

This makes sense, because single node clusters in Redshift are almost aberrations. There's a leader node, which should have an instance all to itself, crammed into a single node along with a single worker slice. Never use single node systems except to have a look at Redshift - and we see here for the small workgroups the devs have avoided single node systems.

We can also see each group of slice numbers has contains about four slices, which means each node is running 4 slices - so they are in fact a smaller node type, quite probably `ra3.4xlarge`.

This however leads to a staggering realization that for the 1 NRPU workgroup, with its two nodes, that these nodes must have ***SIXTY-FOUR*** slices each.

Holy quantum cow on a light beam.

That's a *16x* elastic resize. The largest elastic resize permitted to you and me is 4x, and that's only for the larger node types. It's 2x for the smaller node types.

We've gone from 16 nodes of `ra3.8xlarge` to 2 nodes of `ra3.4xlarge`.

The overheads on disk use for small tables are going to be enormous.

As we've seen, the original cluster, 16 nodes of `ra3.8xlarge`, uses 1280 blocks to write one row to each slice of the test table.

If I made that table on a cluster which was created as two nodes of `ra3.4xlarge`, so no elastic resize, the disk space used would be 5 columns * 2 segments * 2 nodes * 4 slices = 160 blocks.

So - the overhead incurred by using elastic resize from 16 NRPU, rather than a cluster brought up directly with the desired number of nodes is over a *gigabyte* of disk, for our *single* test table.

I think the smallest workgroup sizes are most likely the most common, too - I bet there are more 1 NRPU than any other size.

As an aside, note that we see in the test results for blocks used, that every workgroup is using 1280 blocks for the test able.

NRPU	# blocks	∴ # data slices
64	1280	128
32	1280	128
16	1280	128
8	1280	128
4	1280	128
3	1280	128
2	1280	128
1	1280	128

This makes sense, as it reflects the fact that the number of data slices is always 128, and compute slices do not store data.

As a second aside, here's a table, from the test results, showing the block count for the test table on a 4 node `dc2.large` and a 2 node `dc2.large`, these node types having 2 slices each.

Nodes	Blocks
4	80
2	40

Going back to the table for the findings from `slice_num()`, the fact that the 1, 2 and 3 NRPU workgroups are 2, 4 and 6 nodes of 4 slices explains what we see with the number of slices participating the query.

NRPU	# slices	min	max
64	96	2	503
32	106	0	254
16	95	0	127
8	58	0	119
4	28	0	101
3	24	0	110
2	16	0	99
1	8	0	67

The 4 NRPU workgroup is 4 nodes of 8 slices, so the maximum number of slices which can participate in a query is 4 * 8 = 32.

The 3 NRPU workgroup is 6 nodes of 4 slices, so the maximum number of slices which can participate in a query is 6 * 4 = 24.

This explains why the maximum slice number *rises* when we go go from 4 NRPU to 3 NRPU.

We still have 128 data slices, but they are now on 6 nodes of 4 slices rather than 4 nodes of 8 slices. With 128 data slices, 6 nodes gives us 21.3 slices per node, and 4 nodes gives us 32 slices per node. The 4 slice nodes will run only 4 slices, where Redshift strongly tends to use the lowest numbered slices on a node, so the maximum slice number will be from the first four slices of the final 21 or 22 slices. The 6 slice nodes will run 8 slices, and so the maximum slice number will be from the first eight slices of the final 32 slices.

With 6 nodes, the final node has slice numbers starting at 128 - 21 = 107 or 128 - 22 = 108 (depending whether it receives 21 or 22 slices), and so the maximum slice number will probably be that number, plus four (so 111, or 112).

With 8 nodes, the final node has slices starting at 3 * 32 = 96, and runs eight slices, so it's maximum slice number will be about 104.

Explanations such as this, with so much detail and which explain such a tiny and otherwise inexplicable behaviour, add great weight to the theory that an elastic resize is occurring.

Finally, an upshot of everything we've seen now is that all RPUs are not equal, despite being advertised as serverless and being priced equally.

The 16 NRPU workgroup is the most efficient; it's pure data slices. Workgroups with more NRPU increase their slice count with compute slices, not data slices, and workgroups with less RPU and you're getting more disk seeks and more disk overhead, and if you drop down to 1, 2, or 3 NRPU, you have a different, smaller node type, with only four running slices, and you end up with 64, 32 and 21.3 slices per node, respectively (the 21.3 being where 128 data slices are distributed over 6 nodes, as 128 / 6 = 21.333, so some nodes have 22 slices, others have 21).

Secondary Evidence

Idle Workgroups

When a Serverless workgroup has no clients connected, and has received no queries for 60 or more minutes, when making a new connection, the new connection occurs at the normal speed (almost immediately), but the first query takes some tens of seconds. Queries after this behave and run normally, without this unusual behaviour.

Here's the timing for the first query issued after a 61 minute period with no queries.

Event	Duration (seconds)
Connection	2.052560806274414
First query	27.26804208755493
Second query	0.12465095520019531

In the table above, the first query takes 27 seconds. In earlier results, I saw larger values, such as 40 seconds.

The significance of this idle period and the delay of the initial is obvious, in that if Redshift Serverless is serverless, why would a workgroup going to sleep after one hour?

The delay is however easy to explain if we imagine that under the hood workgroups are ordinary, normal Redshift clusters : if no queries are being issued, all of the hardware for the cluster is provisioned and cannot be used by another user, but no income is accruing, so the cluster must sooner or later be shut down or suspended in some way.

However, the *briefness* of the delay, at 30 or so seconds, is a puzzle to me - and it is the and the only unexplained puzzle in this document.

With provisioned Redshift, there is functionality to *pause* and *resume* clusters.

On the face of it, that seems an obvious answer - but typically those operations take about five minutes each (and that's with an empty small cluster).

I can't see how a workgroup/cluster comes back to life in 30 seconds or so.

Note however, this, something most intriguing, from the general output of the test script;

Line	Output
1	# The Cluster Idle Test…
2	delete_workgroup(wg-u5-1)

Line	Output
3	Failed to delete workgroup, attempt 1, error An error occurred (ConflictException) when calling the DeleteWorkgroup operation: There is an operation running on the workgroup. Try to delete it later.
4	Failed to delete workgroup, attempt 2, error An error occurred (ConflictException) when calling the DeleteWorkgroup operation: There is an operation running on the workgroup. Try to delete it later.
5	Failed to delete workgroup, attempt 3, error An error occurred (ConflictException) when calling the DeleteWorkgroup operation: There is an operation running on the workgroup. Try to delete it later.
6	Failed to delete workgroup, attempt 4, error An error occurred (ConflictException) when calling the DeleteWorkgroup operation: There is an operation running on the workgroup. Try to delete it later.

So, to give context; the test runs, which is to say, it pauses for 61 minutes and then runs the test query, and then when that query has completed, runs the test query a second time (so I can compare how long the query took when it was run the first time after a 61 minute idle period, and how long the query takes when it is run without that idle period).

Having issued the second query, the test script then deletes the workgroup.

The output above is what is seen when the script tries to delete the workgroup; the delete command is refused, on the basis that an operation is already in progress on the workgroup.

I have to loop issuing the delete command until the workgroup is *not* busy with this other operation, whatever it is, so that the delete command can actually occur.

This usually takes about 20 seconds (I retry the delete with a five second pause between each attempt, so the four messages in the above table represent about 20 seconds).

That looks awfully like the cluster is waking up, and an operation is occurring which is the wake up, and so my delete is refused.

In any event, going back to the timing results and in particular now regarding the rapid connection time, I think when a new connection is made to a workgroup, the connection is *not* being made directly to the cluster - it's being made to a proxy, which immediately accepts the connection, and triggers the unpause of the cluster.

You can appear always to be able to immediately issue queries, but if the workgroup has paused, you of course really do have to wait for the workgroup to unpause, and once the cluster is back, the query is serviced, and following queries are then serviced normally.

I note also with provisioned Redshift, the user can select any port number for the cluster, but for Serverless, ports are restricted to the ranges 5431 to 5455 and 8191 to 8215. I suspect this

change is in some way or another related to the connection proxy.

System Tables

As described in Introduction to Redshift Serverless, the large majority of system tables have in Serverless been made inaccessible to both admin and normal users.

The `PG` system tables remain, as do most (but not all - anything which would carry information about what the workgroup is doing, or how, has been removed) of the `SVV` views, and most of the `SYS` views, which seem to be intended for Serverless and so already carry no information about what or how, remain.

As mentioned, the SQL for the `SVV` and `SYS` views has been censored by AWS.

However, we can still figure out some useful information about these views, because we *are* allowed to query the `SYS` views themselves - so we can issue `explain select * from [view_name];`, and the query plan emitted will tell us which real, actual tables are being scanned.

We can then use the `PG` system tables to enumerate the columns of those tables, and see what we think of their columns, and how those columns relate to the columns in the view.

Having done this, what I find is that a number of the underlying tables have columns for `pid` (short for *process ID*, a leader node concept), `node` and `slice`, but that these columns have and have always been removed from the view which uses these tables.

To put it more plainly; the underlying actual tables for Serverless *do* contain information about pids and about nodes and slices, but this information is removed from the views which the user is allowed to access.

The 50 or 60 or so `SVV` tables (they're all views, in fact) have been around for a long time, predating Serverless, but the information they contain is almost wholly about data - tables, users, group, etc - there are only *two* `SVV` views which particularly contain information about what the cluster is doing, and how, these being `svv_query_inflight` and `svv_query_state`.

Those two have been made inaccessible.

A few other `SVV` views have fragmentary or limited information about "what and how", and these also have been made inaccessible.

As part of the test script, I checked to see which `SVV` tables were accessible, and which not. This table gives the inaccessible views.

Name	Accessible
svv_diskusage	False
svv_geography_columns	False
svv_geometry_columns	False
svv_iam_privileges	False
svv_query_inflight	False
svv_query_state	False
svv_restore_table_state	False
svv_rls_applied_policy	False
svv_schema_quota_state	False
svv_sem_usage	False
svv_sem_usage_leader	False

Name	Accessible
svv_vacuum_progress	False
svv_vacuum_summary	False

The view `svv_query_inflight` is the only view in all of the `SVV` views which has a column indicating slice number.

I can imagine the vacuum-related views being removed as I see AWS pushing the line users do not need to even know about `VACUUM`, and can rely on auto-vacuum (but do note that when I investigated auto-vacuum, I found it to be wholly ineffective).

Also of course, needing to tell users they need to manually manage `VACUUM` does not jibe with serverless being presented as different to provisioned, and zero-maintenance for users.

`SYS_SERVERLESS_USAGE` There's is a `SYS` view called `sys_serverless_usage`, which deserves special mention.

This view measures serverless use and as such is specifically for and only for Redshift Serverless. Examining the query plan from `EXPLAIN`, it uses two actual tables, `stll_arcadia_billing_ris_reports` and `stll_xregion_metering`.

The table `stll_arcadia_billing_ris_reports` has the following definition;

name	column	data type
stll_arcadia_billing_ris_reports	billing_time	char(100)
stll_arcadia_billing_ris_reports	cluster_arn	char(128)
stll_arcadia_billing_ris_reports	***cluster_rpus***	int4
stll_arcadia_billing_ris_reports	***instance_type***	char(128)
stll_arcadia_billing_ris_reports	interval_endtime	timestamp
stll_arcadia_billing_ris_reports	interval_starttime	timestamp
stll_arcadia_billing_ris_reports	is_burst	int4
stll_arcadia_billing_ris_reports	***num_nodes***	int4
stll_arcadia_billing_ris_reports	report_arn	char(128)
stll_arcadia_billing_ris_reports	report_time	timestamp
stll_arcadia_billing_ris_reports	request_id	char(100)
stll_arcadia_billing_ris_reports	usage_rpu_sec	int4
stll_arcadia_billing_ris_reports	usage_sec	int4
stll_arcadia_billing_ris_reports	usage_sec_before_minimum	int4

Note the following columns : ***cluster_rpus***, ***instance_type***, ***num_nodes***.

What would a column named *cluster_rpus*, or columns indicating number of nodes, and the type of those nodes, be doing in a table used by a serverless version of Redshift?

Obviously, it is directly and easily explained if Serverless were actually ordinary, normal Redshift clusters, where AWS select the number and type of node based on the number of RPUs.

Function `reboot_cluster()`

Now for a little bit of anecdotal evidence, because it is no longer reproducible.

During the first days of June, running a 1 NRPU workgroup, I tried issuing a call to the `[reboot_cluster()]` function, to see what would happen.

To my mild surprise, it worked, and I was immediately disconnected.

I had the presence of mind to immediately reconnect, and to immediately re-issue the query prior to the reboot, because I already knew at this point that when a workgroup goes idle, you can connect instantly (you're really connecting to a proxy) but the first query takes a long time - I thought the same might happen here, and it did. I had rebooted the workgroup, but I could *instantly* reconnect and issue a new query - it's just I had to wait 30 seconds or so before the query was serviced. Issuing the query again (I have result cache disabled), it is serviced immediately.

So here we're again seeing the connection proxy at work.

I then wrote a test, which is still present in the test script but is now as you see commented out, to demonstrate that `reboot_cluster()` works, and that it behaves in the same way as connecting to and then querying an idle cluster.

The reason the test is commented out is because that night, a new version of provisioned Redshift was published over every region (I do not check Serverless versions, so I do not know if a new version of Serverless went out, but I would think it did, as Serverless workgroups look to be normal, ordinary provisioned Redshift clusters, and so updates are released at the same time as updates to provisioned Redshift), and the next day, `reboot_cluster()` no longer rebooted the workgroup - instead, it returns this message;

> *ERROR: function reboot_cluster is not available in serverless mode*

So I now cannot show you what I experienced the day before that Redshift update.

Now, of this anecdotal evidence, there are two comments to make.

Firstly, I could be completely wrong. I have no documentary evidence, and it might be that I somehow was confused, slipped up, thought I was doing something I wasn't, whatever it is.

Second, it seems improbable that this change occurred at just the time I was testing the function on Serverless.

However, I *have* had such a co-incidence happen once before, about two years ago, when I first began to write Redshift PDFs.

One of the very first PDFs, which looked at cross-database queries, was finished and I was ready to publish the next day. That next day, I ran the test one more time - and *pow*, everything was going wrong.

It turned out overnight there was a new Redshift release, and the way in which cross-database queries work had changed. Formerly, they created an actual temp table, and copied over into that the remote table - you could see the temp tables, how big they were, and they remained in place as long as the session continued, and were deallocated when the session ended.

Now there was no temp table, and in no record at all, of any kind, of the disk space being used for the local copy (but you could see the disk space being used by looking at the total number of blocks in use), and the data copied over did not go away when the session ended (or indeed, even when the remote table was dropped).

So it can happen. Redshift releases are made about every two weeks, I am working on Redshift most days, all that's needed is that something I happen to be working on changes just as I'm working on it. Highly unlikely and so very rare, but not so unlikely as to be unthinkable.

Cluster/Workgroup Limits

In the official documentation, we find the page *[Quotas and limits in Amazon Redshift]*, which enumerates a range of limits, for provisioned Redshift and for Redshift Serverless, such as the maximum number of schemas in a database, the maximum size of a cursor, and so on.

What's noteworthy are that the large majority of limitations that are listed for both systems are either identical or almost identical.

Limit	Serverless	Provisioned
Connections	2,000	2,000
Databases	60	100
Schemas	9,900	9,900
Tables	200,000	200,000

Note the maximum number of tables for normal Redshift varies by node type. You get to 200k, which is the maximum, which you reach you get to `ra3.4xlarge` (a four slice node type) and is unchanged for `ra3.16xlarge` (a sixteen slice node type), and it looks like workgroups use an eight slice node type, so we would expect the 200k limit to be applied.

Finally, note provisioned Redshift has a limit of 10,000 procedures per database. I suspect this also applies to Serverless, but there's no practical, cost-effective, way to find out, as each query to create a procedure would cost 60 seconds of query time, so that's 10000 minutes, which is 7 days of query time. The cheapest Serverless workgroup would charge 480 USD for that.

It's not so important :-)

The obvious matter raised here is that is not clear why a Serverless Redshift should continue to so nearly perfectly present the same, or very nearly the same, limits as Provisioned Redshift - but this is easily explained if, in fact, under the hood, Serverless is an ordinary, normal Redshift cluster.

Namespaces

Turning aside finally from workgroups, let us look now at namespaces.

A namespace is very nearly identical to a snapshot. It contains the complete state of a Redshift cluster - the only difference is that you can start any number of Redshift clusters from a snapshot, but a namespace is constrained to be operated upon by a single workgroup only.

To my eye, you could easily implement a namespace by using the existing snapshot functionality, just by adding a constraint to it such that it can be used with a single workgroup only.

Namespace creation is instant, which makes sense as they're so small - about 50mb or so - but namespace deletion (with almost no additional data since creation) takes a long time, and we see here an 11 minute deletion (for 2 NRPU).

NRPU	Create	Delete
64	2.6436049938201904	139.958984375
32	0.2426462173461914	154.62592577934265
16	0.3628230094909668	197.90971612930298
8	0.40954065322875977	130.91410112380981
4	0.38789892196655273	663.5637822151184
3	0.4128589630126953	225.63704991340637
2	0.28119683265686035	469.62274193763733
1	0.27158045768737793	190.16908884048462

Times seem highly variable, without any obvious connection to NRPU, and what's seen here, with one or two large outliers, is usual.

The following table shows the create and delete times for the seven namespaces brought up for the resize tests.

NRPU	Create	Delete
16	0.5747129917144775	195.5482771396637
16	0.2790834903717041	537.4131038188934
16	0.37318849563598633	267.475439786911
16	0.30323123931884766	264.1939504146576
16	0.41089797019958496	149.59405493736267
16	0.2731807231903076	146.12539196014404
16	0.409212589263916	302.38894510269165

Again, highly variable. I have no idea why.

Deductions

Workgroup Node Types

Now we're figured out what's going on with workgroups, we can look to pull together information about the node type in use for the normal 4 to 64 NRPU workgroups.

To begin with, it will be an `ra3` type node. This is the current (and third) generation of Redshift node hardware. (The `dc2` node type is the second generation of node hardware.)

Let's begin with a table of the existing `ra3` node types.

Name	Slices	Memory (GB/slice)	Memory (GB/node)	Resize (down/up)	Max Nodes	RMS (TB/node)	Price (USD)
`ra3.xlplus`	2	16	32	2x/4x	16/32	32	1.086
`ra3.4xlarge`	4	24	96	4x/4x	32/64	128	3.26
`ra3.16xlarge`	16	24	384	4x/4x	128/128	128	13.04

(The *Resize* column shows two values, the first is the maximum division in size, the second is the maximum multiplication in size. The *Max Nodes* column shows two values, the first being

the maximum number of nodes when bringing a cluster up from scratch, the second showing the maximum number allowed after an elastic resize. No cluster can ever be larger than 128 nodes, by any means :-)

So what of the node type being used by 4 to 64 NRPU workgroups?

Well, we know it has 8 slices, and in the naming convention for `ra3` the number represents the number of slices, so this would provisionally give it the name `ra3.8xlarge`.

(There is one exception to this naming pattern, which is`ra3.xlplus` This is because, to paraphrase one of the devs, "we had not anticipated this node specification, so we were stuck for a name" - the problem being you can't call it `ra3.2xlarge` because it has less memory and RMS per slice than the other node types; it's not linearly proportionate, unlike 4x and 16x.)

Now, both `ra3.4xlarge` and `ra3.16xlarge` have 24 GB of memory per slice.

The docs state 1 RPU, which is one slice, has 16 GB of memory, which means one node of `ra3.8xlarge` would have 128 GB of memory - and that to me seems *really* too little (the `ra3` nodes are generally considered to be under-provisioned for memory in the first place) and it also doesn't fit with the 4x and 16x node types, so the sizes above and below 8x, both having 24 GB of memory.

I think then the docs are wrong, and somewhere someone or something has mixed up the per-slice memory for `ra3.xlplus` (which is 16 GB) with the memory for `ra3.8xlarge`, and so that the real memory per slice is actually 24 GB per slice, which gives a total of 192 GB.

(Note here I see plenty of flat factual errors in the docs, all over the place, so I consider this hypothesis entirely reasonable. I think no one technical reviews the documentation, because the errors I've seen have been so in-your-face obvious that they could not have been missed, had the docs been read by anyone technical.)

So that gives a value for the memory per slice and per node.

Next, RMS, and what's really interesting here is what we find in the Serverless docs about RMS - which is almost nothing.

All I can find in the docs is on a single page, *Understanding Amazon Redshift Serverless capacity*, where there are two statements about storage;

> *The 8, 16, and 24 RPU base RPU capacities are targeted towards workloads that require less than 128TB of data. If your data requirements are greater than 128 TB, you must use a minimum of 32 RPU.*
>
> *Configurations of 8 or 16 RPU support Redshift managed storage capacity of up to 128 TB. If you're using more than 128 TB of managed storage, you can't downgrade to less than 32 RPU.*

Vague, isn't it? but I think there's a reason for that, as will become clear.

The first statement, as it stands, is most straightforwardly read that 8, 16 and 24 RPU have 128 TB of RMS, and 32 RPU supports more than 128 TB, with no upper limit mentioned.

The second statement is deficient - it says 8 and 16 RPU have 128 TB of RMS, says nothing about 24 RPU, and nothing about 32 RPU or greater.

So, where to begin with this mess.

First, I don't think I trust any of this.

With `ra3` node types, RMS storage is on a per-node basis; the more nodes you have, the more store the cluster has. I expect this to be true for Serverless workgroups too, because they are in fact normal, ordinary Redshift clusters.

The problem AWS and the author face here of course is that they can't say anything is ***per-node***, not for Serverless!

So the author must contort his language enough to avoid stating, or even implying, that anything is per-node, and thus we get enough waffle to make a Belgian proud.

So on one hand, I think the need to avoid writing *per-node* has led to sentence construction which is vague and deficient, but on the other hand I *also* actually disbelieve the assertions made here by the documentation even after I account for this obfuscation.

With provisioned `ra3` clusters, the 4x and 16x node types both bring 128 TB of RMS. The `ra3.xlplus` type brings 32 TB of RMS.

I expect then that the amount of RMS store available to a workgroup depends on the type and the number of nodes.

I think what the author was thinking when he wrote the docs, and is trying to obscure because it involves nodes, is that each NRPU brings 32 TB of store, and so 1 NRPU is 32 TB, 2 is 64 TB, 3 is 96 TB; in other words, when the author is saying *less than* or *up to*, what he's omitted saying is that storage varies by NRPU and the *different* workgroups have *different* amounts of total store.

However, I think what the author thinks is in fact *wrong*.

I think that just as the author thinks each RPU has 16 GB of memory - the amount from an `ra3.xlplus` - the author is also thinking each group of 8 RPU brings 32 TB of store - the amount from an `ra3.xlplus`.

The node type for the 4 to 64 NRPU workgroup looks to be an `ra3.8xlarge`, and for the smaller workgroups, `ra3.4xlarge`. Both the `ra3.4xlarge` and `ra3.16xlarge` are specified with 128 TB of RMS per node, so I expect the 8x to provide 128 TB per node as well.

As such, I expect the RMS store for all workgroups is 128 TB times the number of nodes, because the 4 to 64 NRPU workgroups use `ra3.8xlarge` and the 1 to 3 NRPU workgroups use `ra3.4xlarge`, and both of those node types provide or are expected to provide 128 TB of RMS per node.

This means the 1, 2 and 3 NRPU workgroups, which are 2, 4 and 6 nodes respectively, are 256 TB, 512 TB, and 1024 TB of RMS, but that the the 4 NRPU workgroup, being only four nodes, is 512 TB of RMS.

You can bet your bottom dollar AWS are not going to let *that* out in the documentation for Serverless.

So, I think the docs here are simply wrong from start to finish - but it's impossible to test, and I think also it's probably unheard of for users to actually run into such large limits anyway, and if they did, would they comment or mind? perhaps they might mention it to Support, who probably don't know about this, and/or give their usual clueless run-around to the user.

Now, taking the node type table, we can now slot in the workgroup node type as `ra3.8xlarge`, and doing so in fact allows us to extrapolate a *price*, as it would be if this were a provisioned node, because pricing for 4x and 16x is linear (16x is four times the price of 4x), so 8x will be double the price of 4x, and that price is 6.52 USD per hour per node.

Name	Slices	Memory (GB/slice)	Memory (GB/node)	Resize (down/up)	Max Nodes	RMS (TB/node)	Price (USD)
`ra3.xlplus`	2	16	32	2x/4x	16/32	32	1.086
`ra3.4xlarge`	4	24	96	4x/4x	32/64	128	3.26
`ra3.8xlarge`	8	24	192	4x/16x	64/128	128	**6.52**
`ra3.16xlarge`	16	24	384	4x/4x	128/128	128	13.04

Claims of Serverless Dynamic Auto-Scaling

In the light of everything so far, we can now think to revisit the claims of dynamic auto-scaling we saw in Introduction to Redshift Serverless.

AWS *do* claim dynamic, non-interruptive auto-scaling for Redshift Serverless - I find this claim made three times - but provide no information, at all, about the mechanism.

The mechanism is *not* by means of dynamically changing the number of RPU in a workgroup, because changing the number of RPU takes some minutes and brings the workgroup off-line - so how is auto-scaling provided?

Is there somewhere, some harrowing new functionality which actually makes Redshift Serverless, serverless?

Or…

…is it these claims are made on the basis of the and the only mechanism in provisioned Redshift which permits clusters to handle additional load without a resize and going off-line, which is that of Concurrency Scaling Clusters.

I may be wrong, but I do indeed think it's CSC, and I think absolutely nobody would expect that the auto-scaling claims published of Redshift Serverless are fact claims made on the back of Concurrency Scaling Clusters.

AutoWLM, `SYS_QUERY_HISTORY` and CSC First, I am assuming AutoWLM is in use, because I can't imagine for one second it would not be - for it not to be, we'd be looking at some sort of default set up, say the usual one queue with five slots, and given how much time AWS have spent on AutoWLM, the idea it would *not* be in use just doesn't fly.

I spent some time trying to obtain evidence that Concurrency Scaling Clusters (CSC) were in use.

I am able, and always have been, to simply and fully reliably induce the use of a CSC cluster on a normal Redshift cluster, when that cluster is using manual WLM.

This is done by configuring a single queue only, giving it one slot, activating CSC on that queue, issuing a long running query (which fills that slot) and then issuing a second query. The second query induces a CSC cluster, and runs on the CSC cluster.

My next step then was to find a method to induce a CSC cluster, on a normal Redshift cluster, when AutoWLM is in use, as AutoWLM will be in use on workgroups; and then having done so, to run that method on a workgroup.

I was unable to develop such a method.

I could not get a normal Redshift cluster, with AutoWLM, to use CSC.

Instead, I found AutoWLM was queuing the test queries in the single queue, when to my eye those queries could and should have been running in parallel (they were inserts, each insert going to a different table).

I also found that AutoWLM would, while test queries were running, for my manual selects actually add an extra slot to the single queue, run the select in that slot, and then remove that extra slot.

Given this behaviour, I could not get CSC to occur, and as such, I have not been able to prove CSC occurs on Serverless.

The work done to try to induce CSC use with AutoWLM led to my very first investigations, at all, into AutoWLM, which have, to the limited extent those investigations have progressed, raised questions - in the very limited testing I did, AutoWLM was leaving a lot of parallelism, and so performance, on the table.

The investigation into AutoWLM is interesting, but not directly relevant to Serverless, so I have written it up in Appendix B : AutoWLM Investigation.

There is however one piece of circumstantial evidence, for what it's worth.

There is a system table, a view, `sys_query_history`, and the `SYS` views are specifically described as being for Serverless, in which we we find the column *compute_type*, which is defined in the official docs as;

> *Indicates whether the query runs on the main cluster or concurrency scaling cluster. Possible values are primary (query runs on the main cluster) or primary-scale (query runs on the concurrency cluster).*

So, in a system table intended for Serverless, we find a column which indicates if CSC was used for a query.

Of course, it's circumstantial only. Maybe this column is in fact only intended for provisioned Redshift.

As mentioned before, it's not possible to inspect the view SQL and see if that provides any clues, because AWS have censored the view SQL for all `SYS` views.

Limitations of Concurrency Scaling Clusters I do think CSC will be in use, and so I want to provide some information about CSC clusters.

Rather like compute slices are half-fat data slices, CSC clusters are half-fat clusters.

There's a lot CSC clusters cannot do, and although I may be wrong, I am of the view AWS studiously avoid imparting information regarding *limitations* to clients.

The extent of these limitations are however revealed and effectively so by the text of a particular system view, `[svl_query_concurrency_scaling_status]`.

I've included the view text in [Appendix C].

What follows is what is returned by grepping the view text for the string `Concurrency Scaling ineligible query`.

```
THEN CAST('Concurrency Scaling ineligible query - Bootstrap user query' AS text)
THEN CAST('Concurrency Scaling ineligible query - System temporary table accessed' AS text)
THEN CAST('Concurrency Scaling ineligible query - User temporary table accessed' AS text)
THEN CAST('Concurrency Scaling ineligible query - System table accessed' AS text)
THEN CAST('Concurrency Scaling ineligible query - Query is an Unsupported DML' AS text)
THEN CAST('Concurrency Scaling ineligible query - No backup table accessed' AS text)
THEN CAST('Concurrency Scaling ineligible query - Zindex table accessed' AS text)
THEN CAST('Concurrency Scaling ineligible query - Query uses UDF' AS text)
THEN CAST('Concurrency Scaling ineligible query - Catalog tables accessed' AS text)
THEN CAST('Concurrency Scaling ineligible query - Dirty table accessed' AS text)
THEN CAST('Concurrency Scaling ineligible query - Direct dispatched query' AS text)
THEN CAST('Concurrency Scaling ineligible query - No tables accessed' AS text)
THEN CAST('Concurrency Scaling ineligible query - Spectrum queries are disabled' AS text)
THEN CAST('Concurrency Scaling ineligible query - Function not supported ' AS text)
THEN CAST('Concurrency Scaling ineligible query - Instance type not supported ' AS text)
THEN CAST('Concurrency Scaling ineligible query - Burst temporarily disabled ' AS text)
THEN CAST('Concurrency Scaling ineligible query - Unload queries are disabled ' AS text)
THEN CAST('Concurrency Scaling ineligible query - Unsupported unload type ' AS text)
THEN CAST('Concurrency Scaling ineligible query - Non VPC clusters cannot burst ' AS text)
THEN CAST('Concurrency Scaling ineligible query - VPCE not setup ' AS text)
THEN CAST('Concurrency Scaling ineligible query - Query has state on Main cluster ' AS text)
THEN CAST('Concurrency Scaling ineligible query - Query is inelegible for bursting Volt CTAS ' AS text)
THEN CAST('Concurrency Scaling ineligible query - Resource blacklisted ' AS text)
THEN CAST('Concurrency Scaling ineligible query - Non-retryable VoltTT queries are blacklisted ' AS text)
THEN CAST('Concurrency Scaling ineligible query - Query is retrying on Main cluster ' AS text)
THEN CAST('Concurrency Scaling ineligible query - Cannot burst Volt-created CTAS using cursors ' AS text)
THEN CAST('Concurrency Scaling ineligible query - Unsupported VoltTT Utility query ' AS text)
THEN CAST('Concurrency Scaling ineligible query - Write query generating Volt TTs ' AS text)
THEN CAST('Concurrency Scaling ineligible query - VoltTT query with invalid state ' AS text)
THEN CAST('Concurrency Scaling ineligible query - Explain query generating Volt TTs ' AS text)
THEN CAST('Concurrency Scaling ineligible query - Bursting Volt-generated queries is disabled ' AS text)
THEN CAST('Concurrency Scaling ineligible query - Resource of VoltTT UNLOAD is blacklisted ' AS text)
THEN CAST('Concurrency Scaling ineligible query - Multiple pre-Volt query trees ' AS text)
THEN CAST('Concurrency Scaling ineligible query - Target table is DistAll/DistAutoAll ' AS text)
THEN CAST('Concurrency Scaling ineligible query - Table that has diststyle changed in current txn accessed '
THEN CAST('Concurrency Scaling ineligible query - Cannot burst spectrum copy ' AS text)
THEN CAST('Concurrency Scaling ineligible query - Dirty transaction tables accessed  ' AS text)
THEN CAST('Concurrency Scaling ineligible query - Table that has identity column as a target table ' AS text
THEN CAST('Concurrency Scaling ineligible query - Datasharing remote tables accessed' AS text)
THEN CAST('Concurrency Scaling ineligible query - Target table with comp update' AS text)
THEN CAST('Concurrency Scaling ineligible query - Nested tables accessed' AS text)
THEN CAST('Concurrency Scaling ineligible query - Copy from EMR ' AS text)
THEN CAST('Concurrency Scaling ineligible query - Table that has column encode changed in current txn access
THEN CAST('Concurrency Scaling ineligible query - MV refresh disabled ' AS text)
THEN CAST('Concurrency Scaling ineligible query - Too many concurrent writes ' AS text)
THEN CAST('Concurrency Scaling ineligible query - Main cluster too big for writes ' AS text)
THEN CAST('Concurrency Scaling ineligible query - Datasharing VoltTT ' AS text)
THEN CAST('Concurrency Scaling ineligible query - Target table has super/geo column ' AS text)
THEN CAST('Concurrency Scaling ineligible query - Datasharing with burst-write' AS text)
```

```
THEN CAST('Concurrency Scaling ineligible query - CTAS with burst-write' AS text)
THEN CAST('Concurrency Scaling ineligible query - COPY on tables with identity columns' AS text)
THEN CAST('Concurrency Scaling ineligible query - Datasharing query with cursor' AS text)
THEN CAST('Concurrency Scaling ineligible query - Burst MERGE is disabled' AS text)
THEN CAST('Concurrency Scaling ineligible query - Redshift Table via Lake Formation accessed' AS text)
THEN CAST('Concurrency Scaling ineligible query - COPY on table with default value column' AS text)
ELSE CAST('Concurrency Scaling ineligible query - Unknown status' AS text)
```

We see there are a lot of ways in which a query cannot be run on a CSC cluster. Even I was surprised, when I found this code, by how many.

(Note that somewhere included in this list of forbidden queries is that Python UDFs cannot run on CSC clusters. I do not know if Lambda UDF can, or cannot, but I would suspect not - I think the code will be stored on the leader node of the main cluster, which is the reason Python queries cannot be run on CSC clusters.)

I was surprised to see, at it seems, that CSC clusters cannot run queries which access tables containing `geography` or `geometry` columns. To my knowledge, this is not documented.

Of all these issues, that which and always has concerned me most is this;

```
THEN CAST('Concurrency Scaling ineligible query - Dirty table accessed' AS
text)
```

To my eye, this means the CSC cluster has not yet caught up on changes made to a table, and so cannot run queries which use that table.

The question is - how long does it take changes to propagate to CSC clusters? milliseconds? seconds? many seconds? longer? presumably it depends on how much data has changed. What if a table *keeps* changing?

It's a critical question when you're designing a system and you're thinking to use CSC - well, that is, if you're doing more than just putting your hands over your eyes and praying it all works just fine while you hand the money over.

Serverless vs Provisioned Pricing

Now we have a sound-looking estimate for pricing, we can make some direct comparisons between the costs of provisioned and Serverless clusters.

I will use now the 16 NRPU (16 node) workgroup as the working example to compare costs between a provisioned cluster and a workgroup, as this is the only size where a workgroup runs efficiently (no compute slices, no excess disk space overheads) and so of all workgroups can most be directly compared to provisioned clusters brought up directly at their intended size, although we must still remember that for Serverless there are inefficiencies introduced by AutoWLM and CSC.

16 nodes of `ra3.8xlarge` running for one hour in a provisioned cluster costs 104.32 USD.

Assuming AutoWLM does not invoke CSC and will simply run ten queries concurrently for one hour in a Serverless workgroup, a 16 NRPU (16 node) workgroup running for one hour costs 46.08 USD.

I have no idea in Serverless when or upon what criteria CSC clusters will be invoked, and each CSC cluster will be billed at the same rate as the main cluster. If AutoWLM decides to spill a single query over into a CSC cluster, that will double your costs for the duration of that query.

Serverless is on the face of it substantially cheaper than Provisioned, and goes to zero with zero use, but we have to take into consideration the inefficiencies introduced by elastic resize, which become more severe the more the cluster deviates from 16 NRPU, that Serverless uses AutoWLM (which I advise clients *never* to use, and to my eye is a deal-breaker), the use of CSC (which is a band-aid for an incorrectly operated Redshift cluster), as well as the unknown nature of CSC use, and the 60 second minimum charge.

The consistent feedback I get from Redshift admin is that they try AutoWLM, it works fine for a bit, but then sooner or later a big backlog of queries forms, and in the end they turn it off - and AutoWLM as we have it now is at least the third re-write to AutoWLM. The earlier versions were much less successful, and users trying them out rapidly - I might say *frantically* - reverted.

Note that the workgroup will pause after an hour of idle time, and then need some time to resume operations, and you may well need to accommodate for this in your ETL system; it's often not a drop-in replacement.

Also, Serverless has no concept of maintenance and current release tracks - you have current, like it or not. The maintenance track was introduced a few years ago, after a particularly bad run of buggy Redshift releases, where every release broke some major functionality for some users. Maintenance was introduced as a way to protect users. Removing it is *not* an advantage.

Billing for CSC (and Redshift Spectrum, also) are not broken out into separate costs, but are included in a single cost for Redshift Serverless, so you can't tell from looking there what's going on. I can see no system table which provides this information, either.

There is then the question of `VACUUM` and `ANALYZE`, as the background process for auto-vacuum appears from anecdotal report and from my own investigation to be wholly ineffective - I begin in fact to suspect it may issue only `vacuum delete only`, never `vacuum sort`, which would explain a lot, but also render it useless as far as sorting is concerned, which renders it useless as far as *the* reason for using Redshift is concerned (I have not investigated the background process for `ANALYZE`); `VACUUM` on poorly sorted large tables is a multi-day command, where keeping tables sorted in central to the correct operation of sorting, and Redshift without correctly operated sorting makes no sense as a database choice.

All in all, that the cluster goes to zero cost with zero use is a profoundly useful new behaviour, but we must set against that the blunders of the implementation, outlined above.

It would have been much better if AWS had simply introduced zero-zero billing on Provisioned clusters. This would avoid many of the weaknesses and drawbacks of Serverless Redshift, which wholly unnecessarily devalue the Serverless product.

Observations

Capital to Current Billing

I may be wrong, but to my eye there has been a progression on AWS's part from billing arrangements which involve significant and usually up-front capital expenditure, which must be signed off, to arrangements which do *not* involve capital expenditure and so sign off, but are presented as ongoing monthly costs.

To begin with, there were `dc` and `ds` node types, and when a cluster needs more capacity, more nodes must be purchased. When the team managing Redshift wants to buy new nodes, up-front, for a year, it is a significant cost, and it will require approval, and this of course acts to discourage sales.

Then `ra3` nodes arrived. These use cloud for storage, and cloud storage is billed as a monthly cost which rises as storage use rises, and so although there are still one-off, up-front charges for more nodes, part of the price has now changed form, to a monthly running cost, which varies as storage use varies, and does not require sign-off.

Of course, storage use typically rises over time, and in the previous arrangement, would require regular and so visible purchases of new nodes, where-as a monthly running cost only requires a sign-off in the first place, not every month, nor for the relatively small increase in price which comes each month.

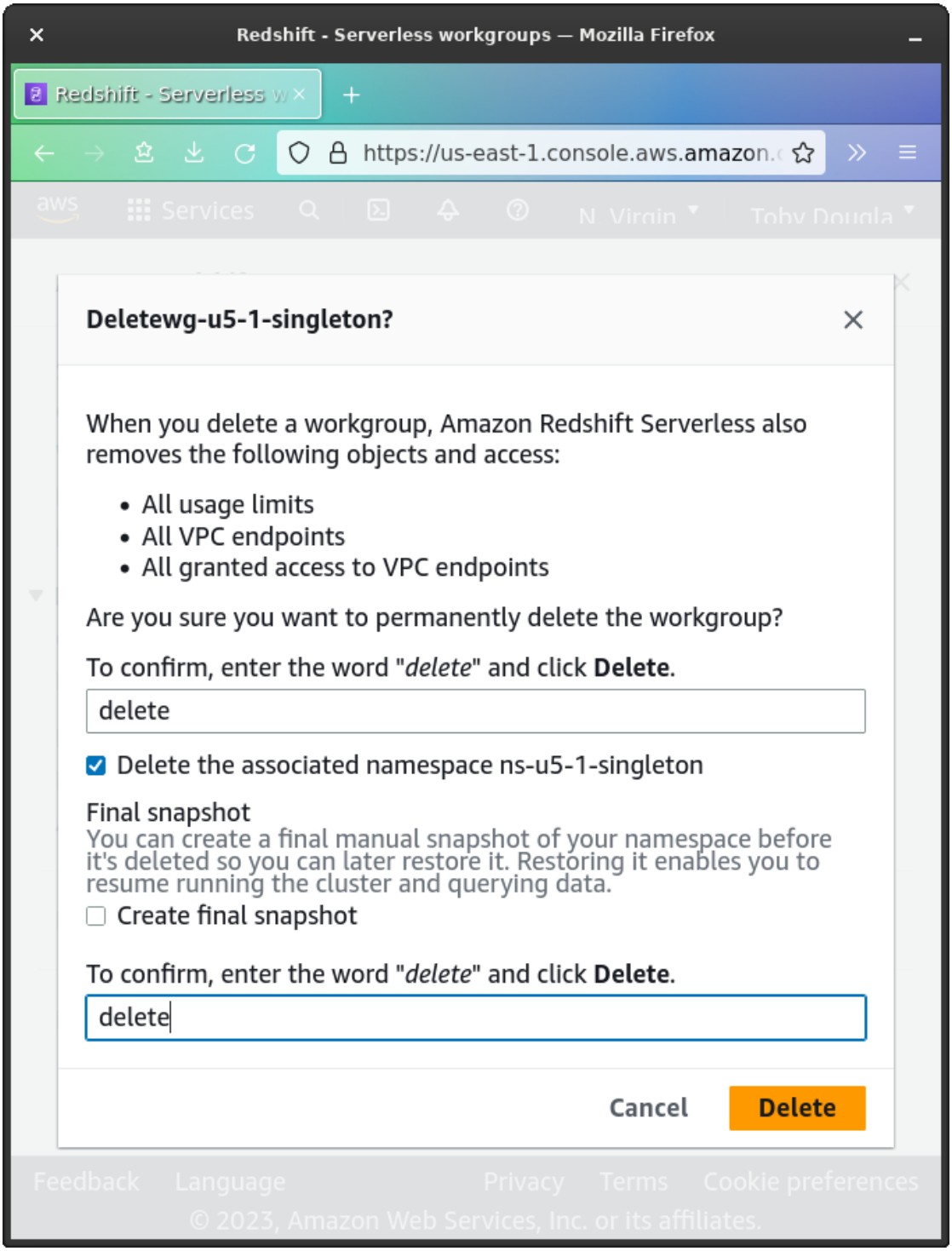
With now Serverless, the entire price of the system is a monthly running cost, both nodes and storage.

This is great for dev teams; they no longer have to get capital purchases past accounts. It's bad for the organization as a whole, because Serverless is expensive and inefficient and a significant cost has become rather invisible.

Serverless Quality and Reliability Issues

In the course of this investigation, both learning to operate Serverless through the Redshift WWW console, and then operating Serverless via `boto3` in the test script, I encountered both minor and major quality and reliability issues.

To begin with the minor but question-raising, look at this dialog window, from the Redshift WWW console.



Look at the first line of text, at the top, in bold, *Deletewg-u5-1-singleton?*.

Note the missing space.

Redshift Serverless had at the time of the screenshot been out for about a year, and this has not been fixed.

I may be wrong, but I think what I think happens at AWS, at least with the Redshift UI and the docs, is that when a project is active, someone is assigned to do the work on and only on that project, and once the project is gone, that person is gone, and *no* other work is done.

I think this then is how even trivial - but highly visible - bugs such as this go unfixed.

The Redshift team seems to be arranged in such a way that the team itself lacks certain vital skills, such as those to modify the Redshift WWW console, or the documentation.

Another example of this is that in provisioned Redshift, there's a missing documentation page, for `alter function`. It's just not there. I noticed this years ago. A while ago I noticed someone raised an issue on GitHub against the docs, stating this page is missing - I think this was in 2019, something like that. It's still not there.

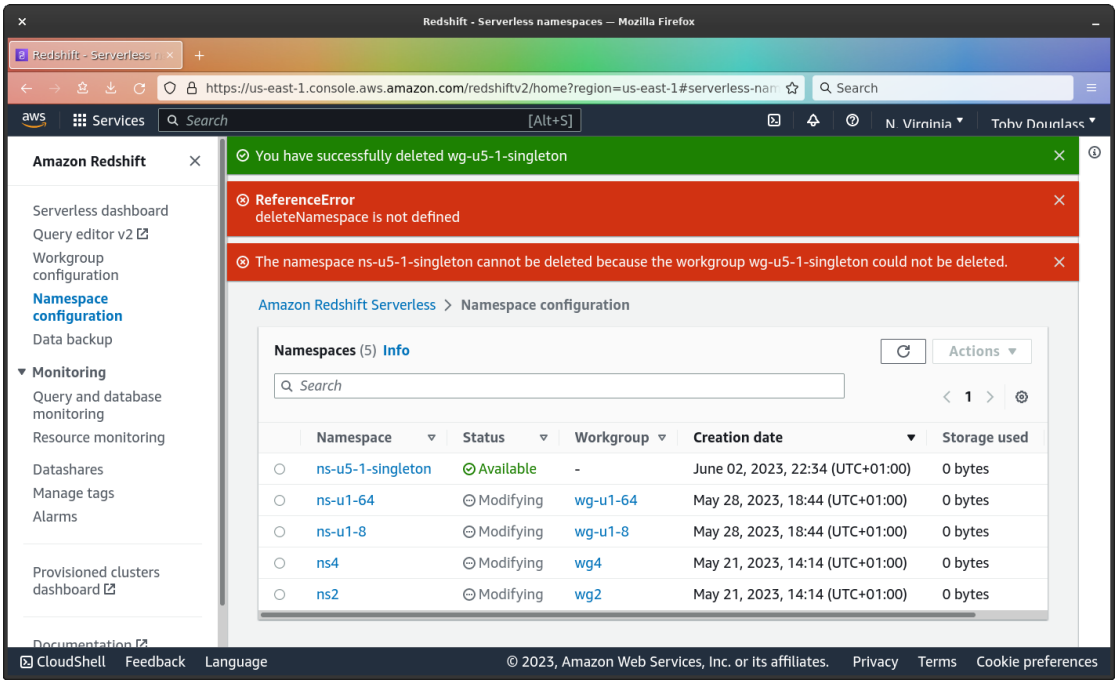
(Note the documentation in general is woefully decayed; the pages for the system tables are now, to my eye, about 15% incorrect even simply in their listing of the columns and data types for the system tables. It's obviously a case where given the very large number of system tables automatic generation of documentation is the only way to maintain correctness, but this is not done; it's manually maintained.)

Next, a slightly more serious issue.

On that same screenshot, note where the checkbox is ticked, for *Delete the associated namespace ns-u5-1 singleton*.

Well, that option, to delete the namespace when you delete the workgroup, *never works*.

There's always an error, telling you the namespace could not be deleted because the workspace exists. Here's a screenshot showing that error.

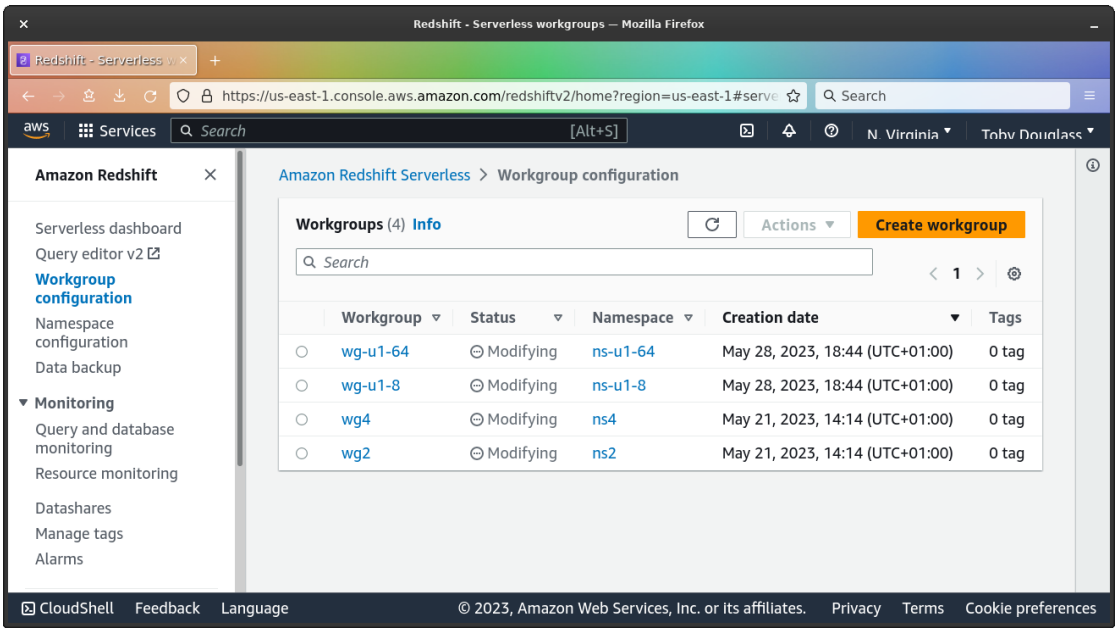


You always have to manually delete the namespace, once the workgroup has gone and the namespace also stops being in the `modifying` state.

Now for the serious stuff.

First, in my test script, if I fired up seven workgroups in parallel (each entirely independent of the other - different `boto3` client, different thread, etc), I could cause two of the workgroups to become stuck.

Here's a screenshot of four stuck workgroups.



I contacted billing support (I do not keep a technical support contract) and first they replied by telling me me that after I finish with a workgroup, I shut it down.

Well, they're billing support, not technical, so fair's fair :-)

After I explained more, they told me they had contacted technical support.

After a week, nothing had happened, and I had two more stuck workgroups.

After another week, the first two workgroups had gone - I've no idea how, billing support were still telling me they were going to chase technical support.

I still have two stuck workgroups, I suspect they will eventually go away by themselves.

It is not obvious to me how *any* operations by the *client* could or should induce a workgroup to become stuck. Something looks to be seriously amiss.

Next, when developing the test script, what I found was that it was impossible to obtain reliable concurrent bring-up, or shut-down, of multiple workgroups. Workgroups would get stuck, or `boto` command would by the looks of it silently fail, so a shut-down would never actually occur, and so on.

I had to serialize workgroup bring-up and shut-down.

I wanted to operate on all seven test workgroups at once (and I can do this just fine with provisioned Redshift), but with Serverless, it always led to problems of one kind or another. It simply was not reliable.

Next, `boto3` was *particularly* difficult to work with - no, let me improve that to *excruciating*; `boto3` is difficult to work with normally, but this was on a whole new level.

When you create a workgroup, you call the `boto3` function to create a workgroup.

The call is issued, and then you wait for the workgroup to become `available`.

What you then find is that being `available` does *not* mean the workgroup can be connected to.

So now you need a loop around your connection code, until that code succeeds.

Now you've connected, you issue queries.

Once you're done with queries, you then issue the function call to delete the workgroup.

Occasionally, this function call silently fails - no exceptions, no errors, just doesn't work. The workgroup remains in the `available` state. Often on test runs, before I serialized, I had to manually delete such workgroups, where the `boto3` function to do so had not worked, and the test script was now stuck, waiting forever for a workgroup which was not being deleted, to be deleted.

This still happens after serializing, but much less often.

So, when the shut-down command does work (which is most of the time, when you're serialized), you then loop, waiting for the workgroup to no longer be enumerated, which is how you know the workgroup has gone.

However, it turns out that the workgroup no longer being enumerated is *not* enough for the namespace to be deleted. If you try to do so after you first see the workgroup no longer exists, namespace deletion fails.

So you then need a loop on the command to delete the namespace, until that command succeeds.

Figuring this all out was exceedingly time consuming. These issues should not exist, and if they do, they should be documented, so rest of the world does not, one by one, have to independently figure these issues out.

Summary

Primary Evidence

Workgroups and Namespaces

Workgroups take a few minutes to create, and to resize, and some to several minutes to delete, and after creation is purportedly complete, a minute or two before they can actually be connected to.

Namespaces are created immediately, but take some to several minutes (more than ten minutes on occasion) to delete.

Creation times are pretty consistent, but there seems to be no rhyme or reason to the length of the delete times, for workgroups or for namespaces; seven back-to-back repetitions of deleting a 128 RPU workgroup took between 146 seconds and 537 seconds.

At creation, workgroups have their compute power specified, and this is *not* dynamic; a resize operation is required to change the compute assigned to a workgroup, which takes some minutes, during which the workgroup goes off-line.

A workgroup can be associated with (and so operates upon) one and only one namespace at a time, and a namespace can be associated with (and so be operated upon) by one and only one workgroup at a time.

Leader Node and Worker Node

With provisioned Redshift, there exists a query which when run on a leader node, and when run on the worker nodes, reveals a difference between the leader node and worker nodes, this being in how dates prior to 1 AD are formatted.

This query, when issued to a Serverless workgroup in exactly the way as is necessary on a provisioned cluster to ensure it runs on the leader node only, produces exactly the same output as on a provisioned cluster (the date formatted using AD/BC), and when run a second time, in exactly the way necessary on a provisioned cluster to ensure it runs on the worker nodes, produces exactly the same output as on a provisioned cluster (the date formatted using ISO 8001).

Slices

It is almost problematic to obtain information from the system tables about what is going on under the hood in Serverless, because the very large majority of the Redshift Serverless system tables have made inaccessible to users, and even the source code of such views as remain have been censored; the function which shows view SQL now returns an error message.

However, it has proved possible using the function slice_num() and through the column *size* in the table `svv_table_info` to obtain enough information about slice use in workgroups to demonstrate beyond any reasonable doubt that all workgroups are in the first instance created with 16 nodes, of 8 slices each, and then elastic resized to the workgroup size the user specified on workgroup creation, which is to say, workgroups must be ordinary, normal Redshift clusters.

A 128 RPU workgroup is 16 nodes of 8 slices, which is to say, 128 slices.

1 RPU is 1 slice.

This explains why the compute limit was originally 32 RPU to 512 RPU (inclusive both ends), with a default of 128 (which is 2 TB of memory - a remarkably large default); the default is the mid-point of a 4x elastic resize range, where we see also that a 4x resize is permitted with a provisioned cluster to `ra3.4xlarge` (four slices) and `ra3.16xlarge` (sixteen slices), and so is likely permitted also to the eight slice node type (which we can easily imagine being `ra3.8xlarge`) being used by the 32 to 512 RPU workgroups.

For workgroups in the range 8 to 24 RPU (inclusive both ends), which were introduced some months after Serverless went to GA (I think because the previous smallest workgroup was still expensive enough to be seen by AWS as discouraging uptake), the node type is different, and is a type with 4 slices (i.e. `ra3.4xlarge`), where here every 8 RPU gives 2 nodes, each with 4 slices. By using 2 nodes for the smallest workgroups, the devs avoided using a single node cluster (you should never use a single node cluster).

This means the smallest (and so I suspect most common) workgroup, of 8 RPU, which is 2 nodes of 4 slices, has ***sixty-four*** slices per node - a staggering 16x elastic resize.

To give perspective to this, the maximum permitted elastic resize change to AWS users is 4x, where as an example, the worst case (largest per-node slice burden) for an `ra3.4xlarge` cluster is to go from 32 nodes with 4 slices each down to 8 nodes with 16 slices each.

With 8 RPU, we have 2 `ra3.4xlarge` nodes with 64 slices each.

The reason for the usual 4x limit is that elastic resizing a cluster introduces inefficiencies.

The nature of the inefficiencies differs depending on whether a cluster is made larger or made smaller than the original cluster. The inefficiencies become greater as the degree of cluster resize (in either direction) becomes larger.

Explaining the nature of the inefficiencies requires too much explanation of fundamental Redshift concepts for a summary. The explanations are given in the Discussion.

To put it briefly, as a cluster becomes increasingly larger, it becomes increasingly inefficient with compute and disk I/O, and as a cluster becomes increasingly smaller, it becomes increasingly inefficient with disk space overheads for tables.

A 512 RPU cluster, the largest, consists of 128 real, full-fat slices ("data slices"), and 384 "filler", half-fat slices ("compute slices"), which do not store data on disk and participate in only a subset of the work which normal slices perform. A provisioned cluster of the same size has a full complement of 512 data slices.

An 8 RPU workgroup will for the worst case (a table with one block per column, e.g. a small table, perhaps 250k rows or less) consume 256mb of disk per column, compared to a provisioned cluster of the same size, which consumes 16mb of disk per column. (Storage is very cheap, so the cost of it is not such an issue, but I am concerned the performance impact, especially with Serverless where charging is based in part on query duration, is significant - I have not investigated this, so my concern is reasonable conjecture only.)

Only the original, 128 RPU workgroup, 16 node cluster, is fully efficient (but here excepting that it will be using AutoWLM, which I consider inefficient and even problematic, and also CSC, which I consider limited and improper.)

This means that RPUs are not all equal, despite being priced equally, and the product being termed Serverless.

Secondary Evidence

Idle Workgroups

Once a workgroup has been idle for 60 minutes, a new connection to the workgroup occurs at the normal speed of a second or less, but a then immediately issued test query takes a few tens of seconds to complete, with the same query issued immediately again after the first query completes running at the expected speed of a fraction of a second (query result cache was disabled).

Additionally, if the attempt is made immediately after the second query to delete the workgroup, the delete command fails (for about 20 seconds, in my tests) because an "operation is in progress on the workgroup".

None of this makes sense for a serverless system, is easy to explain if we imagine that under the hood workgroups are ordinary, normal Redshift clusters : if no queries are being issued, all of the hardware for the cluster is provisioned and cannot be used by another user, but no income is accruing, so the cluster must sooner or later be shut down or suspended in some way.

Note also here that Provisioned Redshift allows the cluster to listen on any port, while Serverless limits ports to the ranges 5431 to 5455 and 8191 to 8215.

Given this new limitation in ports, and given the immediate response when connecting to a workgroup/cluster combined with the delay before the first query executes, I think there is now a proxy in from of the cluster, and connections to the workgroup in fact go to the proxy, giving the appearance of an immediate connection, and the proxy then having received a connection for a shut down workgroup/cluster, triggers the bring-up of that workgroup/cluster, which honours the query once it is able to do so.

System Tables

The very large majority of the system tables have been made inaccessible.

The `PG` system tables remain, as do most of the `SVV` tables, and all of the `SYS` tables, where the `SVV` and `SYS` system tables are actually views.

The `PG` tables contain no information about *what* Serverless is doing, or how; of the `SVV` tables, those few which carry such information have been made inaccessible.

The `SYS` tables I think were intended for Serverless, and have been created in the first place in a sanitized form, carrying no information about what and how.

Furthermore, AWS have taken the extraordinary step of censoring the SQL of the system tables which do remain. There is a function in Postgres which displays the SQL of a view. For the few remaining system table views, this function does not return the SQL text, but rather a message informing the caller the text is not available.

This is astounding. Redshift has been out since about 2012, and never has this happened before - but it happens now, on Serverless.

However, we can still figure out some useful information about these views, because we *are* allowed to query the views themselves, so we can issue `explain select * from [view_name];`, and the query plan emitted will tell us which real, actual tables are being scanned.

We can then use the `PG` system tables to enumerate the columns of those tables, and see what we think of their columns, and how those columns relate to the columns in the view.

Having done this, what I find is that a number of the underlying tables have columns for `pid` (short for *process ID*, a leader node concept), `node` and `slice`, but that these columns have and have always been removed from the views which use these tables.

To put it more plainly; the underlying actual tables for Serverless *do* contain information about pids and about nodes and slices, but this information is removed from the views which the user is allowed to access (and users are not allowed to access the SQL of the system table views).

Finally, there is a system table, `SYS_SERVERLESS_USAGE`, which is specifically for billing information about Serverless, which uses two tables, and in one of those tables we find the columns *cluster_rpus*, *instance_type*, *num_nodes*. It's not obvious why a serverless system would contain a column named *cluster_rpus*, but it is very obvious why it would be so if workgroups are in fact ordinary, normal Redshift clusters.

To my eye, AWS have made a deliberate and comprehensive effort to obscure from users that they are running an ordinary, normal Redshift cluster.

Cluster/Workgroup Limits

Provisioned Redshift specifies a number of system-wide limits, such as the maximum number of tables, schemas, connections and so on.

Limit	Serverless	Provisioned
Connections	2,000	2,000
Databases	60	100
Schemas	9,900	9,900
Tables	200,000	200,000

These limits are almost all identical between provisioned Redshift and Redshift Serverless, which is hard to explain if Serverless is something new and genuinely serverless, but obviously explained if Serverless is actually an ordinary, normal Redshift cluster.

Namespaces

A namespace represents the full state of the data in a cluster; all the databases, all their schemas, tables and views and so on, all groups, all users, etc - in short, it is exactly like a snapshot.

Namespaces are created almost instantly, but take several minutes - sometimes most of ten minutes - to delete. (I never use snapshots, and I did not for this document investigate how long they take to delete.)

A namespace can be operated upon by one and only one workgroup at a time, and a workgroup can operate upon one and only one namespace at a time.

It seems clear simply from this behaviour that a namespace can be implemented by restricting a snapshot such that it operates with and only with a single workgroup at a time.

Deductions

Workgroup Node Types

Let's begin with a table of the existing current generation, `ra3`, node types.

Name	Slices	Memory (GB/slice)	Memory (GB/node)	Resize (down/up)	Max Nodes	RMS (TB/node)	Price (USD)
`ra3.xlplus`	2	16	32	2x/4x	16/32	32	1.086
`ra3.4xlarge`	4	24	96	4x/4x	32/64	128	3.26
`ra3.16xlarge`	16	24	384	4x/4x	128/128	128	13.04

The *Resize* column shows two values, the first is the maximum *reduction* resize, the second is the maximum *increase* in size. The *Max Nodes* column shows two values, the first being the maximum number of nodes when bringing a cluster up from scratch, the second showing the maximum number allowed by an elastic resize. No cluster can ever be larger than 128 nodes, by any means. Elastic resize has made life complicated.

We have seen the 32 to 512 RPU workgroups use an 8 slice node, which if we follow the naming convention will be an `ra3.8xlarge`.

The official docs state that 1 RPU provides 16 GB of memory, which would give 128 GB of memory for 8 slices. This is the memory available to an `ra3.xlplus`, which has 2 slices, and I think this *a priori* too little memory for 8 slices, and looking at the specifications for the `ra3.4xlarge` and `ra3.16xlarge` node types, we see these both, with 4 and 16 slices respectively, have 24 GB per slice. I think this, 24 GB, is actually the correct value.

(Here we must note that a 32 RPU workgroup has 4 nodes of 8 slices with 24 GB of per node, whereas a 24 RPU workgroup has 6 nodes of 4 slices with 24 GB of memory per node.)

Additionally, the official docs write vaguely and imprecisely about how much RMS store is available for workgroups.

With provisioned Redshift, RMS is on a per-node basis, with the 4x and 16x node types providing 128 TB of RMS per-node.

I think the docs regarding RMS are vague because the author of course cannot say anything is *per-node*, not for Serverless, and once we look through this obfuscation, it looks like the author thought that RMS is provisioned at 32 TB per node - again, the value for `ra3.xlplus`.

I think this also is wrong, and that the actual value is 128 TB of RMS per node, just as it is for `ra3.4xlarge` and `ra16x.large`.

Given this, we can slot the workgroup node type into the table, like so;

Name	Slices	Memory (GB/slice)	Memory (GB/node)	Resize (down/up)	Max Nodes	RMS (TB/node)	Price (USD)
`ra3.xlplus`	2	16	32	2x/4x	16/32	32	1.086
`ra3.4xlarge`	4	24	96	4x/4x	32/64	128	3.26
`ra3.8xlarge`	8	24	192	4x/16x	64/128	128	**6.52**
`ra3.16xlarge`	16	24	384	4x/4x	128/128	128	13.04

And this, critically, allows us also to extrapolate a pricing, at 6.52 USD per hour, which permits us now to directly compare Serverless and Provisioned pricing.

Claims of Serverless Dynamic Auto-Scaling

In the documentation for Serverless I can find three vague, woolly statements claiming or implying dynamic auto-scaling.

No information is provided, *at all*, is provided beyond this; no indication of limits, nature, cost, nothing. All there is are simple, usually one-sentence assertions that dynamic auto-scaling exists.

It is however clear that dynamic auto-scaling cannot be implemented by the number of RPUs assigned to a workgroup, as changing this is a manual operation, which takes some minutes, and takes the workgroup off-line (i.e. an elastic resize).

This leaves the and the only mechanism in provisioned Redshift - and all the evidence points and to my eye irrefutably to workgroups being ordinary, normal Redshift clusters - which permits clusters to handle additional load without a resize and going off-line : that of Concurrency Scaling Clusters.

I think absolutely nobody would expect that the auto-scaling claims published of Redshift Serverless are fact made on the back of the well-known, and long-established mechanism of Concurrency Scaling Clusters.

In fact, in the documentation page AWS provide which compares Serverless with Provisioned, there is a table where each rows which compares an aspects of each, one of which I assert is comparing AutoWLM and CSC to each other on both sides.

Serverless	Provisioned
Amazon Redshift can scale for periods of heavy load. Amazon Redshift Serverless also can scale to meet intermittent periods of high load. Amazon Redshift Serverless automatically manages resources efficiently and scales, based on workloads, within the thresholds of cost controls.	With a provisioned data warehouse, you enable concurrency scaling on your cluster to handle periods of heavy load.

Finally, note that Concurrency Scaling Clusters, like compute slices, are skimmed-milk clusters, and that AutoWLM is a black box, and it is completely unclear how and when it does, or does not, invoke CSC clusters; and all of this affects the efficiency of the cluster, and so the price you pay.

Serverless vs Provisioned Pricing

All prices are for `us-east-1`.

With Serverless, 128 RPU (which is to say, 16 nodes), which is the only cluster size which does not experience elastic resize, used for one hour with no CSC, costs 46.08 USD.

16 nodes of the same node type used in Serverless, `ra3.8xlarge`, can be seen to cost 6.52 USD per hour, and so used for one hour with no CSC would cost 104.32 USD.

Provisioned charges at the on-demand rate for CSC clusters. Serverless charges at the rate of the workgroup. With Serverless, AutoWLM is in use, which decides when to use or not use CSC, and there is no control or visibility in billing or in the system tables, of CSC use. If AutoWLM

decides to spill a single query over into a CSC cluster, you are now paying for another workgroup, for the duration of that query.

Serverless is on the face of it substantially cheaper than Provisioned, and goes to zero with zero use, but we have to take into consideration the inefficiencies introduced by elastic resize, which become more severe the more the cluster deviates from 16 NRPU (the largest workgroup is 384 compute slices and only 128 data slices), that Serverless uses AutoWLM (which I advise clients *never* to use, and to my eye is a deal-breaker), the use of CSC (which is a band-aid for an incorrectly operated Redshift cluster), as well as the unknown nature of CSC use, and the 60 second minimum charge.

All in all, that the cluster goes to zero cost with zero use is a profoundly useful new behaviour, but we must set against that the blunders of the implementation, outlined above.

It would have been much better if AWS had simply introduced zero-zero billing on Provisioned clusters. This would avoid many of the weaknesses and drawbacks of Serverless Redshift, which wholly unnecessarily devalue the Serverless product.

Observations

Capital to Current Billing

To my eye, there has been with Redshift a general change, over time, from capital billing to current billing, which makes life easier for dev teams, and improves revenue for AWS, but generally I suspect increases costs.

By this I mean to say that Redshift originally consisted of and only of node types which came with storage, and so when storage was exhausted, a new node or nodes had to be added to a cluster. These are usually by businesses purchased with a one year reservation, and so are a significant single cost and must be approved.

Obtaining approval is a bureaucratic process, which hinders purchasing - both in the time and effort it takes, and in that approval may not be granted.

The first step toward current billing were the `ra3` nodes, which move storage to the cloud, charging 24.576 USD per terabyte per month. Accounts will know there is a monthly AWS bill, which does not require specific authorization or sign-off, unlike purchasing a new node for a year, and storage costs have now moved into that monthly bill.

Now we have the second step, where Serverless, being billed on a per-query-second basis, has moved what was the cost of a node for a year into the monthly bill.

Quality and Reliability Issues

In the course of writing this document and its associated test script, I encountered various quality and reliability issues with Serverless and `boto3`.

For example, originally, the test script, which runs tests on many workgroups, started many (typically seven, I recall) workgroups in parallel, ran tests, and shut down the workgroups.

This turned out to consistently caused two workgroups to become stuck in the modifying state.

After a week or so, the first stuck workgroups went away, and another week or so, the second set went away.

It should not be possible for anything I do, as a client issuing commands through `boto3`, to cause workgroups to become stuck.

Consequently, I have had to serialize workgroup bring-up and shut-down, with the test script at one point taking nine hours to run, as the cumulative time consumed by workgroup bring-up and shut-down, and namespace shut-down, became very large (in the end, I did not need all the tests, and so the test script now is down to about six hours).

In my experience the Python library `boto3` is normally difficult to work with, but was particularly difficult - even for `boto3` - with Serverless. Essentially, every `boto3` call had to be treated as unreliable, both in the status it returned (e.g. a cluster is now available), and also when being called, as often a `boto3` call will fail, because although the previous function indicates it has completed, in fact, it has not, and so following function calls will not work for some time.

Additionally, the Redshift Serverless Console on the AWS web-site, has a blatantly obvious typo, a missing space in an important dialog, and a somewhat more serious issue, whereby when deleting a workgroup, if the option provided to delete the associated namespace is also chosen, the delete of the namespace *always* fails. I suspect these problems have existed from day one of Serverless, and that they have not been fixed is indicative of fundamental weaknesses with Redshift development.

Conclusion

Amazon Redshift Serverless is *not* serverless.

Workgroups are ordinary, normal Redshift clusters.

All workgroups are initially created as a 16 node cluster with 8 slices per node, which is the default 128 RPU workgroup, and then elastic resized to the size specified by the user.

This is why the original RPU range is 32 to 512 in units of 8 and the default is 128 RPU; the default is the mid-point of a 4x elastic resize range, and we see a 128 RPU workgroup being 16 nodes of 8 slices has 128 slices. 1 RPU is 1 slice.

Workgroups do not themselves dynamically auto-scale their number of RPU/slices; to change means changing the number of nodes, which requires as usual a cluster resize. The claims made by Serverless of dynamic auto-scaling, I cannot prove to be (due I think to AutoWLM getting in the way) but I am absolutely certain are based on the well-known and long-established mechanisms of AutoWLM and Concurrency Scaling Clusters.

Elastic resize inherently introduces inefficiencies.

With elastic resize, when the workgroup is made larger than 128 RPU, new nodes are added, the number of data slices does not change from that of the original cluster, and the existing data slices are redistributed over the expanded set of nodes. Any nodes which find themselves with less slices than given in their specification have the gaps filled in with compute slices, which are skimmed-milk slices; they store no data on disk and participate in only a few of the types of work performed by queries. Better than nothing, but not the real thing.

A 512 RPU cluster has 128 data slices and 384 compute slices.

When the workgroup is made smaller than 128 RPU, nodes are removed, the number of data slices does not change from that of the original cluster, and so the data slices are redistributed over the reduced set of nodes, leading to a larger number of data slices on a node.

This leads to inefficiency because all disk I/O in Redshift is in one megabyte blocks. Tables are distributed over data slices (compute slices store no data), are composed of separately stored columns, and each column has a sorted segment and an unsorted segment.

When writing a single row to a table, that row will go to a single slice, which will write one block for every column, to both segments.

If we then write one row per slice, we find the disk used is;

```
[number slices] * [number columns] * [number segments]
```

All tables have the number of columns specified by the user plus three columns, which belong to the system.

Where the original cluster has 128 data slices, if we have a table with a total of 5 columns (two user, three system), and we write one row per slice, we end up with;

```
128 * 5 * 2 = 1280 blocks
```

That's 1.28 gigabytes of disk.

This is the case for as long as we have one block per column per slice, and a block, very roughly speaking - bearing in mind all the different data types and encodings - could be thought on

mean average to have maybe about 150k rows. As such, a lot of small tables are going to be this inefficient.

Every workgroup, larger or smaller than the original, always uses this much disk for its tables, because the number of data slices never changes. Larger workgroups are failing to utilize the hardware they have available, smaller workgroups are using excess disk space.

Some months after the original release of Serverless, workgroup sizes 8, 16 and 24 RPU were introduced. These workgroups use a smaller node type, with 4 slices, and each unit of 8 RPU is two nodes.

The 8 RPU workgroup is then a 16 node 8 slice cluster which has been elastic resized to 2 nodes of 4 slices, giving a staggering ***sixty-four*** data slices per node. This is an 8x elastic resize. To give colour to this, the largest elastic resize permitted to normal users with `ra3.4xlarge` (the publicly available 4 slice current generation Redshift node), goes from 16 nodes of 4 slices down to 4 nodes of 16 slices.

If a 8 RPU workgroup has in the first place been created as 2 nodes of 4 slices, we would have for our 5 column table;

```
8 * 5 * 2 = 80 blocks
```

Which is over a gigabyte less disk used, for this one test table.

Users pay for that gigabyte in their cloud storage costs, but such costs seem very small (about 25 USD per terabyte-month) but also in performance, where so much more network I/O and disk I/O has to be performed.

Now, it must be understood that the situation of one block per slice is the worse-case scenario. As we write more and more rows to the table, there will be more and more blocks, and the overhead of the partially filled final block, and the single empty unsorted block per column of a sorted table, becomes an ever smaller and smaller fraction of the total table size.

However, small tables, roughly 150k or less rows, which are one block per slice per column and are the worst case, are common, and their combined overheads can be huge. (With one client, on `ds2` nodes, I converted all small tables to be unsorted, which removes one of the segments. This saved 20% disk space, and saved them needing to buy 3 nodes, which was worth about 130k USD per year, one year reservation prices.)

Finally, note also that when a node has more slices than given in its specification, it runs only the number of slices equal to its specification. As such, the 8 RPU workgroup, despite having 64 slices per node, runs 4 slices only. These 4 slices (a slice is one running copy of all the processes which are a query) necessarily read the data from all the other slices on the node, which is less efficient than having the data distributed over only the number of actually running slices.

It is then that only the original, 128 RPU workgroup, 16 node cluster, is fully efficient, avoiding both compute slices and undue disk overhead and inefficient disk access (but still, barring AutoWLM and CSC).

This means that all RPUs are not equal, despite being billed for equally, and the product being marketed as serverless, and the smallest workgroup, 1 NRPU, is the extreme of the inefficiency induced by resizing to a smaller workgroup.

The node type in use by the 32 to 512 RPU workgroup looks to be what would be named an `ra3.8xlarge`, which we can fit into the table of publicly available `ra3` node types like so;

Name	Slices	Memory (GB/slice)	Memory (GB/node)	Resize (up/down)	Max Nodes	RMS (TB/node)	Price (USD)
`ra3.xlplus`	2	16	32	2x/4x	16/32	32	1.086
`ra3.4xlarge`	4	24	96	4x/4x	32/64	128	3.26
`ra3.8xlarge`	8	24	192	4x/8x	64/128	128	**6.52**
`ra3.16xlarge`	16	24	384	4x/4x	128/128	128	13.04

I have extrapolated the price as being mid-way between the 4x and 16x node types, as the 8x type is in hardware mid-way between the 4x and 16x node types.

Using the 128 RPU workgroup as the comparison point, as this is the only workgroup non-resized workgroup and so that most able to be directly compared to a Provisioned cluster, we find that for one hour, with no CSC, a provisioned cluster costs 104.32 USD, while a workgroup costs 46.08 USD.

I have no idea in Serverless when or upon what criteria CSC clusters will be invoked, and each CSC cluster will be billed at the same rate as the main cluster; if AutoWLM decides to spill a single query over into a CSC cluster, you are now paying for another workgroup, for the duration of that query.

Serverless is on the face of it substantially cheaper than Provisioned, and goes to zero with zero use, but we have to take into consideration the inefficiencies introduced by elastic resize, which become more severe the more the cluster deviates from 16 NRPU, that Serverless uses AutoWLM (which I advise clients *never* to use, and to my eye is a deal-breaker), the use of CSC (which is a band-aid for an incorrectly operated Redshift cluster), as well as the unknown nature of CSC use, and the 60 second minimum charge.

All in all, that the cluster goes to zero cost with zero use is a profoundly useful new behaviour, but we must set against that the blunders of the implementation, outlined above.

It would have been much better if AWS had simply introduced zero-zero billing on Provisioned clusters. This would avoid many of the weaknesses and drawbacks of Serverless Redshift, which wholly unnecessarily devalue the Serverless product.

Finally, the very large majority of system tables have in Redshift Serverless been made inaccessible. There appears to have been a deliberate and comprehensive effort to remove all information about *what* Serverless is doing, or *how* - up to and including the remarkable step of censoring the SQL of the system table views which do remain; if you use the function which shows the SQL of a view, it emits not the SQL of the view, but a message informing you the view in question cannot be examined.

However, we can still figure out some useful information about the remaining views, because we *are* allowed to query the views themselves, so we can issue `explain select * from [view_name];`, and the query plan emitted will tell us which real, actual tables are being scanned, and we can then use the `PG` system tables to enumerate the columns of those tables, and see what we think of their columns, and how those columns relate to the columns in the view.

Having done this, a number of the underlying tables have columns for pids, nodes and slices, but that these columns have and have always been removed from the views which use these tables.

Credits

1. Currently (2023-10-04) anonymous.

   A reader pointed out the error in my understanding of Serverless pricing. That reader is Stateside, and so although I have asked if they want a credit (everyone is given credit, but of course it has to be something they actually wish for) they are not yet awake so cannot reply. The magnitude of the misunderstanding about pricing is large enough I've immediately made the corrections and feel the PDF has to go out, so for now I've put this in as a placeholder. I will release again, updating the credits, once I have a reply from the reader in question.

Revision History

v1

- Initial release.

v2

- Corrected misunderstanding regarding Serverless pricing.

  Having read the documentation, I understood - incorrectly - that pricing was *per-query*. The docs talk about pricing being per RPU-hour, but billed on a per-second basis, where if a query runs for less than 60 seconds, it is billed for 60 seconds, and this is for a serverless product. Therefore pricing is per-query - as it is with Athena, and with Lambda.

  In fact, it is not. Pricing is per workgroup-second. I still have not found this stated in the docs; I was pointed to a re:Invent talk where an AWS developer presented a slide which made this clear.

  I could write at length about the issues and ambiguities in the documentation, but there's no point, and better ways to spend a life.

  The Redshift docs are normally very poor. In the case of Serverless, where they are also striving to obscure the fact Serverless is not serverless, they are at their worst.

Appendix A : Raw Results Dump

Note these results are completely unprocessed; they are a raw dump of the results, so the original, wholly unprocessed data, is available.

The test script is split into three, a set of slow tests, and a set of fast tests, and a set of tests on a provisioned Redshift cluster.

It used to be the slow tests took about seven hours, because workgroups and `boto3` were not reliable when parallelized, so I had to iterate through a large number of workgroup creations and deletions, which took a long, long time, but after I'd later developed additional evidence and the creation/deletion times no longer much mattered, it was fine to reduce the number of creation/deletion iterations, and so the slow test now takes about three hours.

Nevertheless, it's still slow enough to keep separate from the fast tests.

The fast tests take about ten minutes, although with some variability, as workgroup creation and deletion, and namespace deletion, are rather variable.

The provisioned tests were added on last of all, to illustrate a particular behaviour in workgroups, when they are smaller than 16 NRPU.

As such, all in all, we have three sets of raw results.

First Set of Slow Tests

```
{'idle_test': {'beginning_of_wait_time': 1687216862.7623107,
               'connection_first_query_duration': 27.26804208755493,
               'connection_second_query_duration': 0.12465095520019531,
               'connection_time': 2.052560806274414,
               'end_of_wait_time': 1687220522.7625782},
 'resize': {1: {'create_namespace_duration': 0.409212589263916,
                'create_workgroup_available_duration': 127.59397530555725,
                'create_workgroup_connect_duration': 5.925532102584839,
                'delete_namespace_duration': 302.38894510269165,
                'delete_workgroup_duration': 159.54181098937988,
                'resize_available_duration': 261.23654770851135,
                'resize_connect_duration': 1.3217377662658691},
            2: {'create_namespace_duration': 0.2731807231903076,
                'create_workgroup_available_duration': 127.83186173439026,
                'create_workgroup_connect_duration': 93.99888372421265,
                'delete_namespace_duration': 146.12539196014404,
                'delete_workgroup_duration': 254.6711506843567,
                'resize_available_duration': 214.12547516822815,
                'resize_connect_duration': 2.4654417037963867},
            3: {'create_namespace_duration': 0.41089797019958496,
                'create_workgroup_available_duration': 127.80255341529846,
                'create_workgroup_connect_duration': 104.72378063201904,
                'delete_namespace_duration': 149.59405493736267,
                'delete_workgroup_duration': 273.8036949634552,
                'resize_available_duration': 223.4217014312744,
                'resize_connect_duration': 1.3041656017303467},
            4: {'create_namespace_duration': 0.30323123931884766,
                'create_workgroup_available_duration': 124.32334208488464,
                'create_workgroup_connect_duration': 75.41635346412659,
```

```
                'delete_namespace_duration': 264.1939504146576,
                'delete_workgroup_duration': 105.22236895561218,
                'resize_available_duration': 237.87361240386963,
                'resize_connect_duration': 2.4608988761901855},
            8: {'create_namespace_duration': 0.37318849563598633,
                'create_workgroup_available_duration': 122.82265400886536,
                'create_workgroup_connect_duration': 40.22297286987305,
                'delete_namespace_duration': 267.475439786911,
                'delete_workgroup_duration': 143.2932939529419,
                'resize_available_duration': 307.84038615226746,
                'resize_connect_duration': 2.2486279010772705},
            16: {},
            32: {'create_namespace_duration': 0.2790834903717041,
                 'create_workgroup_available_duration': 158.3465723991394,
                 'create_workgroup_connect_duration': 8.1436448097229,
                 'delete_namespace_duration': 537.4131038188934,
                 'delete_workgroup_duration': 392.1579940319061,
                 'resize_available_duration': 290.20202445983887,
                 'resize_connect_duration': 1.7749905586242676},
            64: {'create_namespace_duration': 0.5747129917144775,
                 'create_workgroup_available_duration': 142.00040555000305,
                 'create_workgroup_connect_duration': 42.092533111572266,
                 'delete_namespace_duration': 195.5482771396637,
                 'delete_workgroup_duration': 303.86812567710876,
                 'resize_available_duration': 221.70308017730713,
                 'resize_connect_duration': 1.1366949081420898}},
 'total_duration': 11070.495510578156}
```

Second Set of Slow Tests

```
{1: {'blocks': [[1280]],
     'create_namespace_duration': 0.27158045768737793,
     'create_workgroup_available_duration': 116.43005561828613,
     'create_workgroup_connect_duration': 22.12782907485962,
     'delete_namespace_duration': 190.16908884048462,
     'delete_workgroup_duration': 314.46830320358276,
     'slices': [[0], [1], [2], [3], [64], [65], [66], [67]]},
 2: {'blocks': [[1280]],
     'create_namespace_duration': 0.28119683265686035,
     'create_workgroup_available_duration': 120.54271459579468,
     'create_workgroup_connect_duration': 15.95941710472107,
     'delete_namespace_duration': 469.62274193763733,
     'delete_workgroup_duration': 57.93810534477234,
     'slices': [[0],
                [1],
                [2],
                [3],
                [32],
                [33],
                [34],
                [35],
                [64],
                [65],
```

```
                [66],
                [67],
                [96],
                [97],
                [98],
                [99]]},
 3: {'blocks': [[1280]],
     'create_namespace_duration': 0.4128589630126953,
     'create_workgroup_available_duration': 153.5724790096283,
     'create_workgroup_connect_duration': 78.02273607254028,
     'delete_namespace_duration': 225.63704991340637,
     'delete_workgroup_duration': 284.17035365104675,
     'slices': [[0],
                [1],
                [2],
                [3],
                [22],
                [23],
                [24],
                [25],
                [44],
                [45],
                [46],
                [47],
                [65],
                [66],
                [67],
                [68],
                [86],
                [87],
                [88],
                [89],
                [107],
                [108],
                [109],
                [110]]},
 4: {'blocks': [[1280]],
     'create_namespace_duration': 0.38789892196655273,
     'create_workgroup_available_duration': 122.49102735519409,
     'create_workgroup_connect_duration': 79.7402982711792,
     'delete_namespace_duration': 663.5637822151184,
     'delete_workgroup_duration': 238.34082007408142,
     'slices': [[0],
                [1],
                [2],
                [4],
                [5],
                [6],
                [7],
                [32],
                [33],
                [34],
                [35],
```

```
                [36],
                [37],
                [38],
                [39],
                [64],
                [65],
                [66],
                [67],
                [68],
                [69],
                [70],
                [71],
                [96],
                [97],
                [99],
                [100],
                [101]]},
 8: {'blocks': [[1280]],
     'create_namespace_duration': 0.40954065322875977,
     'create_workgroup_available_duration': 121.4271936416626,
     'create_workgroup_connect_duration': 69.81401205062866,
     'delete_namespace_duration': 130.91410112380981,
     'delete_workgroup_duration': 184.42388319969177,
     'slices': [[0],
                [1],
                [2],
                [5],
                [6],
                [7],
                [16],
                [17],
                [18],
                [19],
                [20],
                [21],
                [22],
                [23],
                [32],
                [33],
                [34],
                [35],
                [38],
                [39],
                [48],
                [49],
                [50],
                [51],
                [52],
                [53],
                [54],
                [55],
                [64],
                [65],
```

```
                [66],
                [67],
                [68],
                [69],
                [70],
                [71],
                [80],
                [81],
                [82],
                [83],
                [84],
                [85],
                [86],
                [96],
                [97],
                [98],
                [99],
                [100],
                [101],
                [102],
                [103],
                [112],
                [113],
                [115],
                [116],
                [117],
                [118],
                [119]]},
 16: {'blocks': [[1280]],
      'create_namespace_duration': 0.3628230094909668,
      'create_workgroup_available_duration': 122.33415484428406,
      'create_workgroup_connect_duration': 23.383538246154785,
      'delete_namespace_duration': 197.90971612930298,
      'delete_workgroup_duration': 164.1367542743683,
      'slices': [[0],
                 [1],
                 [3],
                 [5],
                 [6],
                 [8],
                 [9],
                 [10],
                 [11],
                 [12],
                 [13],
                 [14],
                 [16],
                 [17],
                 [20],
                 [21],
                 [22],
                 [26],
                 [27],
```

```
[28],
[29],
[30],
[31],
[33],
[35],
[36],
[37],
[38],
[39],
[40],
[42],
[43],
[46],
[47],
[50],
[51],
[52],
[54],
[55],
[58],
[59],
[61],
[62],
[63],
[64],
[66],
[67],
[68],
[69],
[71],
[72],
[73],
[74],
[75],
[76],
[77],
[79],
[80],
[82],
[83],
[84],
[85],
[86],
[87],
[88],
[91],
[93],
[94],
[95],
[96],
[97],
[99],
```

```
                [100],
                [102],
                [103],
                [104],
                [105],
                [106],
                [107],
                [108],
                [109],
                [110],
                [111],
                [113],
                [114],
                [115],
                [116],
                [117],
                [118],
                [119],
                [121],
                [122],
                [125],
                [126],
                [127]]},
32: {'blocks': [[1280]],
     'create_namespace_duration': 0.2426462173461914,
     'create_workgroup_available_duration': 158.84450340270996,
     'create_workgroup_connect_duration': 3.484558582305908,
     'delete_namespace_duration': 154.62592577934265,
     'delete_workgroup_duration': 277.4090383052826,
     'slices': [[0],
                [2],
                [5],
                [7],
                [8],
                [10],
                [12],
                [13],
                [15],
                [16],
                [19],
                [20],
                [22],
                [27],
                [30],
                [32],
                [40],
                [44],
                [46],
                [48],
                [49],
                [50],
                [52],
                [53],
```

```
[56],
[58],
[60],
[61],
[64],
[65],
[66],
[69],
[70],
[72],
[73],
[78],
[80],
[84],
[85],
[88],
[93],
[97],
[98],
[100],
[101],
[105],
[109],
[113],
[114],
[116],
[118],
[120],
[121],
[124],
[125],
[126],
[128],
[132],
[134],
[138],
[139],
[141],
[146],
[150],
[154],
[155],
[156],
[157],
[158],
[160],
[161],
[165],
[166],
[167],
[168],
[169],
[175],
```

```
                [177],
                [194],
                [196],
                [199],
                [202],
                [205],
                [206],
                [207],
                [209],
                [210],
                [213],
                [215],
                [216],
                [218],
                [222],
                [223],
                [226],
                [230],
                [232],
                [234],
                [236],
                [237],
                [238],
                [240],
                [243],
                [244],
                [247],
                [248],
                [254]]},
 64: {'blocks': [[1280]],
      'create_namespace_duration': 2.6436049938201904,
      'create_workgroup_available_duration': 162.94810509681702,
      'create_workgroup_connect_duration': 36.03853249549866,
      'delete_namespace_duration': 139.958984375,
      'delete_workgroup_duration': 149.57489109039307,
      'slices': [[2],
                 [4],
                 [8],
                 [10],
                 [12],
                 [14],
                 [16],
                 [18],
                 [20],
                 [21],
                 [22],
                 [24],
                 [31],
                 [34],
                 [38],
                 [40],
                 [42],
                 [47],
```

```
[48],
[50],
[52],
[54],
[56],
[62],
[64],
[66],
[68],
[71],
[72],
[74],
[76],
[78],
[80],
[82],
[84],
[88],
[90],
[92],
[94],
[96],
[97],
[99],
[104],
[107],
[112],
[115],
[117],
[118],
[120],
[124],
[126],
[128],
[148],
[151],
[161],
[173],
[176],
[203],
[206],
[210],
[212],
[215],
[223],
[225],
[229],
[236],
[237],
[260],
[261],
[266],
[274],
```

```
                  [286],
                  [302],
                  [304],
                  [308],
                  [312],
                  [314],
                  [326],
                  [341],
                  [345],
                  [375],
                  [390],
                  [411],
                  [428],
                  [429],
                  [434],
                  [435],
                  [448],
                  [452],
                  [455],
                  [458],
                  [460],
                  [464],
                  [475],
                  [497],
                  [503]]},
 'total_duration': 5400.682659864426}
```

Fast Tests

```
{'ResponseMetadata': {'HTTPHeaders': {'content-length': '1364',
                                      'content-type': 'application/x-amz-json-1.1',
                                      'date': 'Sun, 18 Jun 2023 15:42:32 GMT',
                                      'x-amzn-requestid': '663da721-58e6-43a3-82bc-831bcc084bc6'},
                      'HTTPStatusCode': 200,
                      'RequestId': '663da721-58e6-43a3-82bc-831bcc084bc6',
                      'RetryAttempts': 0},
 'workgroup': {'baseCapacity': 8,
               'configParameters': [{'parameterKey': 'auto_mv',
                                     'parameterValue': 'true'},
                                    {'parameterKey': 'datestyle',
                                     'parameterValue': 'ISO, MDY'},
                                    {'parameterKey': 'enable_case_sensitive_identifier',
                                     'parameterValue': 'false'},
                                    {'parameterKey': 'enable_user_activity_logging',
                                     'parameterValue': 'true'},
                                    {'parameterKey': 'query_group',
                                     'parameterValue': 'default'},
                                    {'parameterKey': 'search_path',
                                     'parameterValue': '$user, public'},
                                    {'parameterKey': 'max_query_execution_time',
                                     'parameterValue': '14400'}],
               'creationDate': datetime.datetime(2023, 6, 18, 15, 39, 22, 450000, tzinfo=tzutc()),
               'endpoint': {'address': 'wg-u5-singleton.113054258080.us-east-1.redshift-serverless.amazonaws
```

```
                            'port': 5439,
                            'vpcEndpoints': [{'networkInterfaces': [{'availabilityZone': 'us-east-1a',
                                                                     'networkInterfaceId': 'eni-0cc68c57838
                                                                     'privateIpAddress': '10.0.0.209',
                                                                     'subnetId': 'subnet-0d17cfec748d93a6f'
                                              'vpcEndpointId': 'vpce-089b71cae8b1ee764',
                                              'vpcId': 'vpc-0ea69c6fc1b412aa8'}]},
                  'enhancedVpcRouting': False,
                  'namespaceName': 'ns-u5-singleton',
                  'publiclyAccessible': True,
                  'securityGroupIds': ['sg-04527b3477c3cde4b'],
                  'status': 'DELETING',
                  'subnetIds': ['subnet-032e9ad380734f725',
                                'subnet-0933d224f4f307906',
                                'subnet-0d17cfec748d93a6f'],
                  'workgroupArn': 'arn:aws:redshift-serverless:us-east-1:113054258080:workgroup/bdb43f37-d471-4
                  'workgroupId': 'bdb43f37-d471-4719-a00e-e4448337ce54',
                  'workgroupName': 'wg-u5-singleton'}}
wait_for_workgroup_deleted( wg-u5-singleton )
Sleeping... (wg-u5-singleton, 0)
Sleeping... (wg-u5-singleton, 1)
Sleeping... (wg-u5-singleton, 2)
Sleeping... (wg-u5-singleton, 3)
Sleeping... (wg-u5-singleton, 4)
Sleeping... (wg-u5-singleton, 5)
Sleeping... (wg-u5-singleton, 6)
Sleeping... (wg-u5-singleton, 7)
Sleeping... (wg-u5-singleton, 8)
Sleeping... (wg-u5-singleton, 9)
Sleeping... (wg-u5-singleton, 10)
Sleeping... (wg-u5-singleton, 11)
Sleeping... (wg-u5-singleton, 12)
Sleeping... (wg-u5-singleton, 13)
Sleeping... (wg-u5-singleton, 14)
Sleeping... (wg-u5-singleton, 15)
Sleeping... (wg-u5-singleton, 16)
Sleeping... (wg-u5-singleton, 17)
Sleeping... (wg-u5-singleton, 18)
Sleeping... (wg-u5-singleton, 19)
Sleeping... (wg-u5-singleton, 20)
Sleeping... (wg-u5-singleton, 21)
Sleeping... (wg-u5-singleton, 22)
Sleeping... (wg-u5-singleton, 23)
Sleeping... (wg-u5-singleton, 24)
Sleeping... (wg-u5-singleton, 25)
Sleeping... (wg-u5-singleton, 26)
Sleeping... (wg-u5-singleton, 27)
Sleeping... (wg-u5-singleton, 28)
Sleeping... (wg-u5-singleton, 29)
Sleeping... (wg-u5-singleton, 30)
Sleeping... (wg-u5-singleton, 31)
Sleeping... (wg-u5-singleton, 32)
Sleeping... (wg-u5-singleton, 33)
```

```
Sleeping... (wg-u5-singleton, 34)
Sleeping... (wg-u5-singleton, 35)
Sleeping... (wg-u5-singleton, 36)
Sleeping... (wg-u5-singleton, 37)
Sleeping... (wg-u5-singleton, 38)
delete_namespace( ns-u5-singleton )
wait_for_namespace_deleted( ns-u5-singleton )
Sleeping... (ns-u5-singleton, 0)
Sleeping... (ns-u5-singleton, 1)
Sleeping... (ns-u5-singleton, 2)
Sleeping... (ns-u5-singleton, 3)
Sleeping... (ns-u5-singleton, 4)
Sleeping... (ns-u5-singleton, 5)
Sleeping... (ns-u5-singleton, 6)
Sleeping... (ns-u5-singleton, 7)
Sleeping... (ns-u5-singleton, 8)
Sleeping... (ns-u5-singleton, 9)
Sleeping... (ns-u5-singleton, 10)
Sleeping... (ns-u5-singleton, 11)
Sleeping... (ns-u5-singleton, 12)
Sleeping... (ns-u5-singleton, 13)
Sleeping... (ns-u5-singleton, 14)
Sleeping... (ns-u5-singleton, 15)
Sleeping... (ns-u5-singleton, 16)
Sleeping... (ns-u5-singleton, 17)
Sleeping... (ns-u5-singleton, 18)
Sleeping... (ns-u5-singleton, 19)
Sleeping... (ns-u5-singleton, 20)
Sleeping... (ns-u5-singleton, 21)
Sleeping... (ns-u5-singleton, 22)
Sleeping... (ns-u5-singleton, 23)
Sleeping... (ns-u5-singleton, 24)
Sleeping... (ns-u5-singleton, 25)
Sleeping... (ns-u5-singleton, 26)
Sleeping... (ns-u5-singleton, 27)
Sleeping... (ns-u5-singleton, 28)
Sleeping... (ns-u5-singleton, 29)
Sleeping... (ns-u5-singleton, 30)
Sleeping... (ns-u5-singleton, 31)
Sleeping... (ns-u5-singleton, 32)
Sleeping... (ns-u5-singleton, 33)
Sleeping... (ns-u5-singleton, 34)
Sleeping... (ns-u5-singleton, 35)
Sleeping... (ns-u5-singleton, 36)
Sleeping... (ns-u5-singleton, 37)
Sleeping... (ns-u5-singleton, 38)
Sleeping... (ns-u5-singleton, 39)
{'explain_plans': {'pg_catalog.sys_connection_log': [['XN Seq Scan on '
                                                      'stll_connection_log  '
                                                      '(cost=0.00..0.20 '
                                                      'rows=20 width=3194)']],
                   'pg_catalog.sys_copy_job': [['LD Seq Scan on '
                                                'pg_auto_copy_job job_table  '
```

```
                              '(cost=0.00..4.00 rows=400 '
                              'width=142)']],
 'pg_catalog.sys_datashare_change_log': [['XN Subquery Scan '
                                          'sys_datashare_change_log  '
                                          '(cost=0.00..3.33 '
                                          'rows=20 '
                                          'width=592)'],
                                         ['  ->  XN Append  '
                                          '(cost=0.00..3.13 '
                                          'rows=20 '
                                          'width=4548)'],
                                         ['        ->  XN '
                                          'Subquery Scan '
                                          '"*SELECT* 1"  '
                                          '(cost=0.00..1.68 '
                                          'rows=10 '
                                          'width=4548)'],
                                         ['              ->  '
                                          'XN Seq Scan on '
                                          'stll_datashare_changes_producer  '
                                          '(cost=0.00..1.58 '
                                          'rows=10 '
                                          'width=4548)'],
                                         ['        ->  XN '
                                          'Subquery Scan '
                                          '"*SELECT* 2"  '
                                          '(cost=0.00..1.45 '
                                          'rows=10 '
                                          'width=4144)'],
                                         ['              ->  '
                                          'XN Seq Scan on '
                                          'stll_datashare_changes_consumer  '
                                          '(cost=0.00..1.35 '
                                          'rows=10 '
                                          'width=4144)']],
 'pg_catalog.sys_datashare_usage_consumer': [['XN Seq Scan '
                                              'on '
                                              'stll_datashare_request_consumer  '
                                              '(cost=0.00..0.92 '
                                              'rows=1 '
                                              'width=1880)'],
                                             ['  Filter: '
                                              '((((api_call '
                                              '<= 5) AND '
                                              '(api_call >= '
                                              '2)) OR '
                                              '((api_call <= '
                                              '15) AND '
                                              '(api_call >= '
                                              '14))) AND '
                                              '(dbtype > '
                                              '0))']],
 'pg_catalog.sys_datashare_usage_producer': [['XN Seq Scan '
```

```
                                          'on '
                                          'stll_datashare_request_producer  '
                                          '(cost=0.00..0.52 '
                                          'rows=1 '
                                          'width=2960)'],
                                         ['  Filter: '
                                          '((((api_call '
                                          '<= 5) AND '
                                          '(api_call >= '
                                          '2)) OR '
                                          '((api_call <= '
                                          '15) AND '
                                          '(api_call >= '
                                          '14))) AND '
                                          '(shareid <> '
                                          '0))']],
'pg_catalog.sys_external_query_detail': [['XN Subquery Scan '
                                          'sys_external_query_detail  '
                                          '(cost=0.05..0.12 '
                                          'rows=1 '
                                          'width=12316)'],
                                         ['  ->  XN '
                                          'HashAggregate  '
                                          '(cost=0.05..0.11 '
                                          'rows=1 '
                                          'width=25876)'],
                                         ['        ->  XN '
                                          'Seq Scan on '
                                          'stll_external_query_detail_base  '
                                          '(cost=0.00..0.00 '
                                          'rows=1 '
                                          'width=25876)'],
                                         ['              '
                                          'Filter: (user_id '
                                          '> 1)']],
'pg_catalog.sys_integration_activity': [['XN Seq Scan on '
                                         'stll_integration_log  '
                                         '(cost=0.00..0.25 '
                                         'rows=19 '
                                         'width=1640)'],
                                        ['  Filter: (status '
                                         '<> '
                                         "'parsed'::bpchar)"]],
'pg_catalog.sys_integration_table_state': [['XN Seq Scan on '
                                            'stv_integration_table_state  '
                                            '(cost=0.00..0.20 '
                                            'rows=20 '
                                            'width=3100)']],
'pg_catalog.sys_load_detail': [['XN Hash Join '
                                'DS_BCAST_INNER  '
                                '(cost=19.45..56000036.06 '
                                'rows=14 width=800)'],
                               ['  Hash Cond: '
```

```
                                    '((("outer".query)::bigint '
                                    '= '
                                    '"inner".rewritten_query_id) '
                                    'AND ("outer".userid = '
                                    '"inner".user_id))'],
                                   ['  ->  XN Seq Scan on '
                                    'stll_load_commits  '
                                    '(cost=0.00..0.40 rows=40 '
                                    'width=796)'],
                                   ['  ->  XN Hash  '
                                    '(cost=18.45..18.45 '
                                    'rows=200 width=20)'],
                                   ['        ->  XN Subquery '
                                    'Scan l  (cost=0.00..18.45 '
                                    'rows=200 width=20)'],
                                   ['              ->  XN '
                                    'Unique  (cost=0.00..16.45 '
                                    'rows=200 width=20)'],
                                   ['                    ->  XN '
                                    'Seq Scan on '
                                    'stll_user_query_map  '
                                    '(cost=0.00..9.40 rows=940 '
                                    'width=20)']],
'pg_catalog.sys_load_error_detail': [['XN Hash Left Join '
                                      'DS_BCAST_INNER  '
                                      '(cost=160000.84..5760001.23 '
                                      'rows=7 width=2849)'],
                                     ['  Hash Cond: '
                                      '("outer".query_id = '
                                      '"inner".query_id)'],
                                     ['  ->  XN Seq Scan on '
                                      'stll_user_load_error_detail  '
                                      '(cost=0.00..0.25 '
                                      'rows=7 width=2809)'],
                                     ['        Filter: '
                                      '(user_id > 1)'],
                                     ['  ->  XN Hash  '
                                      '(cost=160000.81..160000.81 '
                                      'rows=10 width=48)'],
                                     ['        ->  XN '
                                      'Subquery Scan '
                                      'history  '
                                      '(cost=160000.64..160000.81 '
                                      'rows=10 width=48)'],
                                     ['              ->  XN '
                                      'HashAggregate  '
                                      '(cost=160000.64..160000.71 '
                                      'rows=10 width=224)'],
                                     ['                    '
                                      '->  XN Hash Left '
                                      'Join DS_BCAST_INNER  '
                                      '(cost=0.00..160000.51 '
                                      'rows=10 width=224)'],
```

```
                                        ['                          '
                                         'Hash Cond: '
                                         '("outer".query_id = '
                                         '"inner".query_id)'],
                                        ['                        '
                                         '->  XN Seq Scan on '
                                         'stll_load_history_base  '
                                         '(cost=0.00..0.38 '
                                         'rows=10 width=224)'],
                                        ['                              '
                                         'Filter: (user_id > '
                                         '1)'],
                                        ['                          '
                                         '->  XN Hash  '
                                         '(cost=0.00..0.00 '
                                         'rows=1 width=8)'],
                                        ['                                '
                                         '->  XN Seq Scan on '
                                         'stv_user_query_state '
                                         'suqs  '
                                         '(cost=0.00..0.00 '
                                         'rows=1 width=8)']],
 'pg_catalog.sys_load_history': [['XN Subquery Scan '
                                  'sys_load_history  '
                                  '(cost=160001.16..160002.06 '
                                  'rows=10 width=272)'],
                                 ['  ->  XN HashAggregate  '
                                  '(cost=160001.16..160001.96 '
                                  'rows=10 width=1504)'],
                                 ['        ->  XN Hash Left '
                                  'Join DS_BCAST_INNER  '
                                  '(cost=0.00..160000.51 '
                                  'rows=10 width=1504)'],
                                 ['              Hash Cond: '
                                  '("outer".query_id = '
                                  '"inner".query_id)'],
                                 ['              ->  XN Seq '
                                  'Scan on '
                                  'stll_load_history_base  '
                                  '(cost=0.00..0.38 rows=10 '
                                  'width=1496)'],
                                 ['                    '
                                  'Filter: (user_id > 1)'],
                                 ['              ->  XN '
                                  'Hash  (cost=0.00..0.00 '
                                  'rows=1 width=8)'],
                                 ['                    ->  '
                                  'XN Seq Scan on '
                                  'stv_user_query_state '
                                  'suqs  (cost=0.00..0.00 '
                                  'rows=1 width=8)']],
 'pg_catalog.sys_query_detail': [['XN Subquery Scan '
                                  'sys_query_detail  '
```

```
 '(cost=0.38..56000032.43 '
 'rows=28 width=320)'],
['  ->  XN Append  '
 '(cost=0.38..56000032.15 '
 'rows=28 width=3807)'],
['        ->  XN Subquery '
 'Scan "*SELECT* 1"  '
 '(cost=0.38..0.63 rows=1 '
 'width=3807)'],
['              ->  XN '
 'HashAggregate  '
 '(cost=0.38..0.62 rows=1 '
 'width=3807)'],
['                    ->  '
 'XN Seq Scan on '
 'stll_query_detail  '
 '(cost=0.00..0.30 rows=1 '
 'width=3807)'],
['                          '
 'Filter: ((internal_query '
 "= 'f'::bpchar) AND "
 '(user_id > 1))'],
['        ->  XN Subquery '
 'Scan "*SELECT* 2"  '
 '(cost=56000030.78..56000031.52 '
 'rows=27 width=819)'],
['              ->  XN '
 'HashAggregate  '
 '(cost=56000030.78..56000031.25 '
 'rows=27 width=819)'],
['                    ->  '
 'XN Hash Join '
 'DS_BCAST_INNER  '
 '(cost=23.30..56000030.04 '
 'rows=27 width=819)'],
['                          '
 'Hash Cond: '
 '(("outer".query)::bigint '
 '= '
 '"inner".rewritten_query_id)'],
['                          '
 '->  XN Seq Scan on '
 'stv_exec_state se  '
 '(cost=0.00..1.00 rows=27 '
 'width=811)'],
['                                '
 'Filter: (userid > 1)'],
['                          '
 '->  XN Hash  '
 '(cost=22.80..22.80 '
 'rows=200 width=20)'],
['                                '
 '->  XN Subquery Scan m  '
```

```
                                    '(cost=0.00..22.80 '
                                    'rows=200 width=20)'],
                                   ['                                        '
                                    '->  XN Unique  '
                                    '(cost=0.00..20.80 '
                                    'rows=200 width=24)'],
                                   ['                                                '
                                    '->  XN Seq Scan on '
                                    'stv_user_query_map  '
                                    '(cost=0.00..10.40 '
                                    'rows=1040 width=24)']],
 'pg_catalog.sys_query_history': [['XN Subquery Scan '
                                   'derived_table18  '
                                   '(cost=0.00..0.15 rows=2 '
                                   'width=393)'],
                                  ['  ->  XN Append  '
                                   '(cost=0.00..0.05 rows=2 '
                                   'width=15788)'],
                                  ['        ->  XN Subquery '
                                   'Scan "*SELECT* 1"  '
                                   '(cost=0.00..0.02 rows=1 '
                                   'width=15788)'],
                                  ['              ->  XN Seq '
                                   'Scan on '
                                   'stll_user_query_history  '
                                   '(cost=0.00..0.01 rows=1 '
                                   'width=15788)'],
                                  ['                    '
                                   'Filter: ((internal_query '
                                   "= 'f'::bpchar) AND "
                                   '(query_text <> '
                                   "''::bpchar) AND (status "
                                   "<> 'planning'::bpchar) "
                                   'AND (status <> '
                                   "'queued'::bpchar) AND "
                                   '(status <> '
                                   "'returning'::bpchar) AND "
                                   '(status <> '
                                   "'running'::bpchar))"],
                                  ['        ->  XN Subquery '
                                   'Scan "*SELECT* 2"  '
                                   '(cost=0.00..0.03 rows=1 '
                                   'width=15788)'],
                                  ['              ->  XN Seq '
                                   'Scan on '
                                   'stv_user_query_state  '
                                   '(cost=0.00..0.02 rows=1 '
                                   'width=15788)'],
                                  ['                    '
                                   'Filter: ((internal_query '
                                   "= 'f'::bpchar) AND "
                                   '((status = '
                                   "'Queued'::bpchar) OR "
```

```
                                    '(status = '
                                    "'Returning'::bpchar) OR "
                                    '(status = '
                                    "'Running'::bpchar) OR "
                                    '(status = '
                                    "'planning'::bpchar) OR "
                                    '(status = '
                                    "'running'::bpchar)) AND "
                                    '(end_time < start_time) '
                                    'AND (query_text <> '
                                    "''::bpchar))"]],
 'pg_catalog.sys_query_text': [['XN Seq Scan on '
                                'stll_user_query_text  '
                                '(cost=0.00..1.10 rows=110 '
                                'width=640)']],
 'pg_catalog.sys_serverless_usage': [['XN Subquery Scan '
                                      'sys_serverless_usage  '
                                      '(cost=1000333704889.90..1000333704896.90 '
                                      'rows=200 width=56)'],
                                     ['  ->  XN Window  '
                                      '(cost=1000333704889.90..1000333704894.90 '
                                      'rows=200 width=56)'],
                                     ['        Order: '
                                      'start_time'],
                                     ['        ->  XN Sort  '
                                      '(cost=1000333704889.90..1000333704890.40 '
                                      'rows=200 width=56)'],
                                     ['              Sort '
                                      'Key: start_time'],
                                     ['              ->  XN '
                                      'Network  '
                                      '(cost=333704879.26..333704882.26 '
                                      'rows=200 width=56)'],
                                     ['                    '
                                      'Send to slice 0'],
                                     ['                    '
                                      '->  XN Subquery Scan '
                                      'usage_info  '
                                      '(cost=333704879.26..333704882.26 '
                                      'rows=200 width=56)'],
                                     ['                         '
                                      '->  XN HashAggregate  '
                                      '(cost=333704879.26..333704880.26 '
                                      'rows=200 width=56)'],
                                     ['                              '
                                      '->  XN Nested Loop '
                                      'Left Join '
                                      'DS_BCAST_INNER  '
                                      '(cost=45600038.36..333697688.86 '
                                      'rows=410880 '
                                      'width=56)'],
                                     ['                                    '
                                      'Join Filter: '
```

```
 '(("inner".eventtime < '
 '"outer".end_time) AND '
 '("inner".eventtime >= '
 '"outer".start_time))'],
['                                        '
 '->  XN Nested Loop '
 'Left Join '
 'DS_BCAST_INNER  '
 '(cost=45600038.36..295205240.86 '
 'rows=23112 width=48)'],
['                                          '
 'Join Filter: '
 '(("outer".start_time '
 '<= "inner".logtime) '
 'AND ("outer".end_time '
 '> "inner".logtime))'],
['                                          '
 '->  XN Subquery Scan '
 'billing_and_reports_log  '
 '(cost=45600038.36..45600040.86 '
 'rows=200 width=40)'],
['                                                '
 '->  XN HashAggregate  '
 '(cost=45600038.36..45600038.86 '
 'rows=200 width=52)'],
['
 '->  XN Hash Full Join '
 'DS_DIST_BOTH  '
 '(cost=21.35..45600034.86 '
 'rows=200 width=52)'],
['
 'Outer Dist Key: '
 'bl.end_time'],
['
 'Inner Dist Key: '
 'stll_arcadia_billing_ris_reports.interval_endtime']
['
 'Hash Cond: '
 '(("outer".end_time = '
 '"inner".interval_endtime) '
 'AND '
 '("outer".start_time = '
 '"inner".interval_starttime))'],
['
 '->  XN Subquery Scan '
 'bl  '
 '(cost=20.75..29.25 '
 'rows=200 width=32)'],
['
 '->  XN HashAggregate  '
 '(cost=20.75..27.25 '
 'rows=200 width=28)'],
['
```

```
                                          '->  XN Seq Scan on '
                                          'stll_arcadia_billing_log  '
                                          '(cost=0.00..8.30 '
                                          'rows=830 width=28)'],
                                         ['
                                          '->  XN Hash  '
                                          '(cost=0.40..0.40 '
                                          'rows=40 width=20)'],
                                         ['
                                          '->  XN Seq Scan on '
                                          'stll_arcadia_billing_ris_reports  '
                                          '(cost=0.00..0.40 '
                                          'rows=40 width=20)'],
                                         ['                                          '
                                          '->  XN Seq Scan on '
                                          'stll_cluster_blocks_stats  '
                                          '(cost=0.00..10.40 '
                                          'rows=1040 width=16)'],
                                         ['                                      '
                                          '->  XN Seq Scan on '
                                          'stll_xregion_metering  '
                                          '(cost=0.00..1.60 '
                                          'rows=160 width=16)'],
                                         ['----- Nested Loop '
                                          'Join in the query '
                                          'plan - review the '
                                          'join predicates to '
                                          'avoid Cartesian '
                                          'products -----']],
'pg_catalog.sys_spectrum_scan_error': [['XN Hash Left Join '
                                        'DS_BCAST_INNER  '
                                        '(cost=0.55..4000000.60 '
                                        'rows=1 '
                                        'width=14022)'],
                                       ['  Hash Cond: '
                                        '("outer".query = '
                                        '"inner".child_query_id)'],
                                       ['  ->  XN Seq Scan '
                                        'on '
                                        'stll_spectrum_scan_error  '
                                        '(cost=0.00..0.00 '
                                        'rows=1 '
                                        'width=14018)'],
                                       ['  ->  XN Hash  '
                                        '(cost=0.50..0.50 '
                                        'rows=20 width=12)'],
                                       ['        ->  XN '
                                        'Subquery Scan d  '
                                        '(cost=0.00..0.50 '
                                        'rows=20 width=12)'],
                                       ['              ->  '
                                        'XN Unique  '
                                        '(cost=0.00..0.30 '
```

```
                                        'rows=20 width=12)'],
                                       ['                    '
                                        '->  XN Seq Scan on '
                                        'stll_query_detail  '
                                        '(cost=0.00..0.20 '
                                        'rows=20 '
                                        'width=12)']],
'pg_catalog.sys_stream_scan_errors': [['XN Subquery Scan '
                                       'sys_stream_scan_errors  '
                                       '(cost=0.00..0.70 '
                                       'rows=20 width=260)'],
                                      ['  ->  XN Append  '
                                       '(cost=0.00..0.50 '
                                       'rows=20 '
                                       'width=2371)'],
                                      ['        ->  XN '
                                       'Subquery Scan '
                                       '"*SELECT* 1"  '
                                       '(cost=0.00..0.20 '
                                       'rows=10 '
                                       'width=2371)'],
                                      ['              ->  '
                                       'XN Seq Scan on '
                                       'stll_kinesis_scan_errors  '
                                       '(cost=0.00..0.10 '
                                       'rows=10 '
                                       'width=2371)'],
                                      ['        ->  XN '
                                       'Subquery Scan '
                                       '"*SELECT* 2"  '
                                       '(cost=0.00..0.30 '
                                       'rows=10 '
                                       'width=1992)'],
                                      ['              ->  '
                                       'XN Seq Scan on '
                                       'stll_kafka_scan_errors  '
                                       '(cost=0.00..0.20 '
                                       'rows=10 '
                                       'width=1992)']],
'pg_catalog.sys_stream_scan_states': [['XN Subquery Scan '
                                       'sys_stream_scan_states  '
                                       '(cost=0.00..2.50 '
                                       'rows=70 width=224)'],
                                      ['  ->  XN Append  '
                                       '(cost=0.00..1.80 '
                                       'rows=70 '
                                       'width=2004)'],
                                      ['        ->  XN '
                                       'Subquery Scan '
                                       '"*SELECT* 1"  '
                                       '(cost=0.00..0.60 '
                                       'rows=30 '
                                       'width=2004)'],
```

```
                                        ['             ->  '
                                         'XN Seq Scan on '
                                         'stll_kinesis_scan_states  '
                                         '(cost=0.00..0.30 '
                                         'rows=30 '
                                         'width=2004)'],
                                        ['        ->  XN '
                                         'Subquery Scan '
                                         '"*SELECT* 2"  '
                                         '(cost=0.00..1.20 '
                                         'rows=40 '
                                         'width=1625)'],
                                        ['              ->  '
                                         'XN Seq Scan on '
                                         'stll_kafka_scan_states  '
                                         '(cost=0.00..0.80 '
                                         'rows=40 '
                                         'width=1625)']],
 'pg_catalog.sys_unload_detail': [['XN Hash Join '
                                   'DS_BCAST_INNER  '
                                   '(cost=24.15..80000032.46 '
                                   'rows=7 width=3934)'],
                                  ['  Hash Cond: '
                                   '((("outer".query)::bigint '
                                   '= '
                                   '"inner".rewritten_query_id) '
                                   'AND ("outer".userid = '
                                   '"inner".user_id))'],
                                  ['  ->  XN Seq Scan on '
                                   'stll_unload_log  '
                                   '(cost=0.00..0.20 rows=20 '
                                   'width=3918)'],
                                  ['  ->  XN Hash  '
                                   '(cost=23.15..23.15 '
                                   'rows=200 width=32)'],
                                  ['        ->  XN Subquery '
                                   'Scan 1  '
                                   '(cost=0.00..23.15 '
                                   'rows=200 width=32)'],
                                  ['              ->  XN '
                                   'Unique  '
                                   '(cost=0.00..21.15 '
                                   'rows=200 width=32)'],
                                  ['                    ->  '
                                   'XN Seq Scan on '
                                   'stll_user_query_map  '
                                   '(cost=0.00..9.40 '
                                   'rows=940 width=32)']],
 'pg_catalog.sys_unload_history': [['XN Subquery Scan '
                                    'sys_unload_history  '
                                    '(cost=160001.46..160002.65 '
                                    'rows=14 width=264)'],
                                   ['  ->  XN HashAggregate  '
```

```
                                                  '(cost=160001.46..160002.51 '
                                                  'rows=14 width=1716)'],
                                                 ['        ->  XN Hash '
                                                  'Left Join '
                                                  'DS_BCAST_INNER  '
                                                  '(cost=0.00..160000.69 '
                                                  'rows=14 width=1716)'],
                                                 ['              Hash '
                                                  'Cond: ("outer".query_id '
                                                  '= "inner".query_id)'],
                                                 ['              ->  XN '
                                                  'Seq Scan on '
                                                  'stll_unload_history_base '
                                                  'suhb  (cost=0.00..0.50 '
                                                  'rows=14 width=1708)'],
                                                 ['                    '
                                                  'Filter: (user_id > 1)'],
                                                 ['              ->  XN '
                                                  'Hash  (cost=0.00..0.00 '
                                                  'rows=1 width=8)'],
                                                 ['                    ->  '
                                                  'XN Seq Scan on '
                                                  'stv_user_query_state '
                                                  'suqs  (cost=0.00..0.00 '
                                                  'rows=1 width=8)']]},
 'svv_access': {'svv_all_columns': True,
                'svv_all_schemas': True,
                'svv_all_tables': True,
                'svv_alter_table_recommendations': True,
                'svv_attached_masking_policy': True,
                'svv_column_privileges': True,
                'svv_columns': True,
                'svv_database_privileges': True,
                'svv_datashare_consumers': True,
                'svv_datashare_objects': True,
                'svv_datashare_privileges': True,
                'svv_datashares': True,
                'svv_default_privileges': True,
                'svv_diskusage': False,
                'svv_external_columns': True,
                'svv_external_databases': True,
                'svv_external_partitions': True,
                'svv_external_schemas': True,
                'svv_external_tables': True,
                'svv_function_privileges': True,
                'svv_geography_columns': False,
                'svv_geometry_columns': False,
                'svv_iam_privileges': False,
                'svv_identity_providers': True,
                'svv_integration': True,
                'svv_interleaved_columns': True,
                'svv_language_privileges': True,
                'svv_masking_policy': True,
```

```
                 'svv_ml_model_info': True,
                 'svv_ml_model_privileges': True,
                 'svv_mv_dependency': True,
                 'svv_mv_info': True,
                 'svv_query_inflight': False,
                 'svv_query_state': False,
                 'svv_redshift_columns': True,
                 'svv_redshift_databases': True,
                 'svv_redshift_functions': True,
                 'svv_redshift_schemas': True,
                 'svv_redshift_tables': True,
                 'svv_relation_privileges': True,
                 'svv_restore_table_state': False,
                 'svv_rls_applied_policy': False,
                 'svv_rls_attached_policy': True,
                 'svv_rls_policy': True,
                 'svv_rls_relation': True,
                 'svv_role_grants': True,
                 'svv_roles': True,
                 'svv_schema_privileges': True,
                 'svv_schema_quota_state': False,
                 'svv_sem_usage': False,
                 'svv_sem_usage_leader': False,
                 'svv_system_privileges': True,
                 'svv_table_info': True,
                 'svv_tables': True,
                 'svv_transactions': True,
                 'svv_user_grants': True,
                 'svv_user_info': True,
                 'svv_vacuum_progress': False,
                 'svv_vacuum_summary': False},
 'total_duration': 600.8878445625305}
```


Provisioned Tests


```
{'us-east-1': {'ra3.xlplus': {2: {'slices_num': [[0], [1], [2], [3]],
                                  'stv_slices': [[0, 0, 0, 'data'],
                                                 [0, 1, 1, 'data'],
                                                 [0, 6, 2, 'data'],
                                                 [0, 7, 3, 'data'],
                                                 [1, 2, 0, 'data'],
                                                 [1, 3, 1, 'data'],
                                                 [1, 4, 2, 'data'],
                                                 [1, 5, 3, 'data']]},
                              4: {'blocks': [[80]],
                                  'slices_num': [[0],
                                                 [1],
                                                 [2],
                                                 [3],
                                                 [4],
                                                 [5],
                                                 [6],
                                                 [7]],
```

```
                                    'stv_slices': [[0, 0, 0, 'data'],
                                                   [0, 1, 1, 'data'],
                                                   [1, 2, 0, 'data'],
                                                   [1, 3, 1, 'data'],
                                                   [2, 4, 0, 'data'],
                                                   [2, 5, 1, 'data'],
                                                   [3, 6, 0, 'data'],
                                                   [3, 7, 1, 'data']]}}}}

{'us-east-1': {'ra3.xlplus': {2: {'blocks': [[40]],
                                    'slices_num': [[0], [1], [2], [3]],
                                    'stv_slices': [[0, 0, 0, 'data'],
                                                   [0, 1, 1, 'data'],
                                                   [1, 2, 0, 'data'],
                                                   [1, 3, 1, 'data']]},
                              4: {'slices_num': [[1], [3], [4], [5]],
                                    'stv_slices': [[0, 0, 0, 'data'],
                                                   [0, 4, 1, 'compute'],
                                                   [1, 2, 0, 'data'],
                                                   [1, 5, 1, 'compute'],
                                                   [2, 1, 0, 'data'],
                                                   [2, 6, 1, 'compute'],
                                                   [3, 3, 0, 'data'],
                                                   [3, 7, 1, 'compute']]}}}}
```

Appendix B : AutoWLM Investigation

As part of this investigation, I wanted to develop a way to induce the use of a CSC cluster while AutoWLM is in use.

I was unable to do so, but the work which was done lead to some interesting first findings about AutoWLM. I have performed no investigation into AutoWLM, I never use it, and I advise clients never to use it, so it's been a closed book until now.

Now, with manual WLM, I know how to invoke a CSC cluster; configure one queue with CSC enabled and one slot only, issue a long running query (I use a self-join insert) and then issue a second query (in this case, a select).

The select will invoke a CSC cluster, always, and it should do, according to the described behaviour of CSC - a CSC cluster will be used when a queue which has CSC enabled is full.

So far so good, but what about AutoWLM?

With AutoWLM, you control how many queues there are, but that's it; you no longer control the number of slots or the amount of memory assigned to the queue (and SQA is always enabled, which is another mechanism I advise clients not to use).

You also cannot *see* this information, because it is no longer published in the system tables. With manual WLM, you can see how much memory and how many slots each queue has. With AutoWLM, this information is no longer provided and is `NULL`.

Note however Redshift has and even with AutoWLM I expect still has a total, system-wide limit of 50 slots.

If CSC clusters are still invoked in the same way - a queue is full - then it's going to be necessary I would think to bombard Redshift with enough queries that a queue becomes full, where all slots, however many there are, being in use concurrently, forcing a CSC cluster to be used.

Where Redshift Serverless charges by query-second and has its minimum 60 second charge, this would be an expensive test to develop :-)

As such, I began on a normal two node `dc2.large` cluster, which I rent for half a dollar per hour :-)

If I could reliably induce a CSC cluster, with AutoWLM is use, then I could use that method on a Serverless workgroup, and see what happens.

I created a cluster, and configured it to have one queue only (the default queue), with CSC active, and with AutoWLM enabled.

I wrote a test script, in Python, where the test script runs, and every one second, spawns a new thread, that thread connects to the cluster, creates a new table, inserts four initial rows, and then begins to issue back-to-back pairs of an insert using a self-join and a `select count(*)`.

So here we see each thread is fully independent; its own connection, its own table, no dependencies or involvement with anything else.

During the development process, I made one or two discoveries, which allowed a little insight in the queue and slot configuration currently chosen by AutoWLM.

Firstly, it is still possible to see how many slots are being used by a query - you can't see information about queues, but you can still see information about queries.

This information, the slots used by a query, is found in `stv_wlm_query_state`, which shows currently running queries, and also in `stl_wlm_query`, which shows the history of all completed queries.

The latter tables allows me to write a query which tracks over time the total number of slots in use - which gives us some insight into what AutoWLM is doing.

Secondly, it turns out the `SET` variable `wlm_query_slot_count` still seems to be functional.

By default, a query uses a single slot. However, if I set `wlm_query_slot_count` to a higher value, I see that value being shown in the `slot_count` column of `stv_wlm_query_state`.

The obvious thought then is to set `wlm_query_slot_count` to a high value, like 55 (more than the maximum possible number of slots) and issue a query.

Note the docs state you cannot set this value to higher than 50. This is not true. What actually happens, at least with AutoWLM (I've not tried it with manual WLM), is if you specify a slot count greater than the available number of slots is that Redshift silently reduces the number of slots allocated to the number of slots currently available in the queue the query is sent to, and you can perfectly well set the value to any number you like.

As such, when I set `wlm_query_slot_count` to 55, and issued a query, I could see it was being assigned 3 (three) slots.

My conclusion was that AutoWLM had at that time configured a single queue with three slots.

My expectation then was that issuing a select at this point would induce the use of a CSC cluster.

This, however, did *not* occur.

The select was serviced, while the three-slot query was ongoing, without a CSC cluster.

This left me quite bemused.

I then ran the test script described earlier - one new thread per second, each which makes a new table, inserts four initial rows, and then starts issuing back-to-back pairs of insert self-joins and `select count(*)`s.

To my *considerable* surprise, despite this script running, a select *still* did not invoke a CSC cluster.

Examining `stv_wlm_query_state`, I would see this (the final column is just off the page - sorry - limitations of the document tooling);

```
dev=# select * from stv_wlm_query_state;
  xid  |  task  | query  | service_class | slot_count |       wlm_start_time       |      state       | queue_time | exec_time |    query
-------+--------+--------+---------------+------------+----------------------------+------------------+------------+-----------+---------
 16122 | 600287 | 601150 |           100 |          3 | 2023-05-29 12:31:20.639253 | QueuedWaiting    |   68859749 |         0 | High
 16145 | 600291 | 601180 |           100 |          3 | 2023-05-29 12:31:25.34471  | QueuedWaiting    |   64154303 |         0 | High
 16169 | 600295 | 601210 |           100 |          3 | 2023-05-29 12:31:31.297203 | QueuedWaiting    |   58201735 |         0 | High
 16195 | 600299 | 601240 |           100 |          3 | 2023-05-29 12:31:36.164369 | QueuedWaiting    |   53334654 |         0 | High
 16224 | 600303 | 601270 |           100 |          3 | 2023-05-29 12:31:39.692616 | QueuedWaiting    |   49806400 |         0 | High
 16253 | 600307 | 601300 |           100 |          3 | 2023-05-29 12:31:44.737511 | QueuedWaiting    |   44761503 |         0 | High
 16280 | 600312 | 601331 |           100 |          3 | 2023-05-29 12:31:49.807809 | QueuedWaiting    |   39691087 |         0 | High
 16300 | 600316 | 601340 |           100 |          1 | 2023-05-29 12:32:29.496731 | Running          |          0 |      1990 | Normal
 16112 | 600285 | 601137 |           100 |          3 | 2023-05-29 12:31:18.329078 | Running          |          0 |  71169857 | Normal
(9 rows)
```

The first thread has its insert self-join running, and all the other threads have their queries queued.

Quite simply, CSC was not occurring, and indeed, even basic parallelism was not occurring - to my eye, even without CSC, many of those queued queries could and should have been running in parallel - remember, they're all writing to different tables.

(Note the `query_priority` stuff has nothing to do with me - I did not set it - AutoWLM is doing that by itself, and the single query with a slot count of 1 is the query which examined the system tables to produce the data you see in the table above.)

On the face of it, with this toe-in-the-water test, AutoWLM seems to be leaving a lot of concurrency, and so performance, on the table.

Now, in this situation, where the test script is running, one thread is having its queries serviced and the others are all blocked, but if I manually issue a select, it *is* serviced, and no CSC cluster arises.

So the question is how the hell is that select being serviced? why doesn't it queue?

It turns out what's happening is that AutoWLM sneaky like adds an extra slot to the queue, runs the query in that slot, and then removes that slot once the select has completed!

I can see this by looking at `stl_wlm_query`, and running a query which shows me the total number of slots in use over time.

I get this, where `tsc` is the `total slot count` in use. Time is ordered descending.

```
          event_ts          | tsc
----------------------------+-----
 2023-05-29 13:19:08.366717 |   1
 2023-05-29 13:19:09.095557 |   0
 2023-05-29 13:19:09.133869 |   1
 2023-05-29 13:19:09.360735 |   0
 2023-05-29 13:19:09.798422 |   1
 2023-05-29 13:19:09.958103 |   0
 2023-05-29 13:19:48.315023 |   1
 2023-05-29 13:19:48.471904 |   0
 2023-05-29 13:19:55.791994 |   1
 2023-05-29 13:19:55.854427 |   0
 2023-05-29 13:19:58.091691 |   1
 2023-05-29 13:19:58.102345 |   0
 2023-05-29 13:19:58.370454 |   3
 2023-05-29 13:19:58.500976 |   0
 2023-05-29 13:19:58.754763 |   3
 2023-05-29 13:19:58.772187 |   0
 2023-05-29 13:19:59.043939 |   3
 2023-05-29 13:19:59.782303 |   4
 2023-05-29 13:19:59.851293 |   3
 2023-05-29 13:19:59.972588 |   4
 2023-05-29 13:20:00.585337 |   3
 2023-05-29 13:20:01.128958 |   0
 2023-05-29 13:20:02.76454  |   1
 2023-05-29 13:20:02.793358 |   0
```

The initial pattern of 1 and 0 are the polling of the query which produces this output.

I then kick off the test script, and we then see the pattern of 3 and 0, where `wlm_query_slot_count` is set to 55.

With the test script running, the first thread is having its queries serviced (each of which uses three slots) and all other threads have their query queued.

We then, critically, observe *four* slots in use.

That happens when the polling which produces this output runs while one of the three-slot test script queries is running. AutoWLM has snuck in an extra slot, and then removed it when the query completes.

So, all in all, I could not find a way to induce CSC with AutoWLM, so I could not prove CSC is in use with Serverless.

Appendix C : `svl_query_concurrency_scaling_status`

First, the columns for the system tables view `svl_query_concurrency_scaling_status`, from;

https://www.redshiftresearchproject.org/system_table_tracker/1.0.55524/pg_catalog.svl_query_concurrency_scaling_status.html;

column	data type
concurrency_scaling_status	int4
concurrency_scaling_status_txt	text
endtime	timestamp
pid	int4
query	int4
source_query	int4
starttime	timestamp
userid	int4
xid	int8

```
SELECT sq.userid,
       sq.query,
       sq.xid,
       sq.pid,
       sr.source_query,
       sq.concurrency_scaling_status,
       CASE
         WHEN (sq.userid = 1)
           THEN CAST('Concurrency Scaling ineligible query - Bootstrap user query' AS text)
         WHEN (sq.concurrency_scaling_status = 1)
           THEN CAST('Ran on a Concurrency Scaling cluster' AS text)
         WHEN (    (sq.concurrency_scaling_status = 0)
               AND (sr.source_query IS NOT NULL))
           THEN CAST('Ran on the main cluster - Cache hit' AS text)
         WHEN (    (sq.concurrency_scaling_status = 0)
               AND (sw.service_class = 14))
           THEN CAST('Ran on the main cluster - SQA' AS text)
         WHEN (    (sq.concurrency_scaling_status = 32)
               AND (sw.service_class = 14))
           THEN CAST('Ran on the main cluster - Concurrency scaling is not enabled in SQA' AS text)
         WHEN (    (sq.concurrency_scaling_status = 0)
               AND sq.query IN (SELECT stl_burst_prepare.query
                                FROM stl_burst_prepare)
               AND (NOT sq.query IN (SELECT stl_burst_prepare.query
                                     FROM stl_burst_prepare
                                     WHERE (stl_burst_prepare.code = 0))))
           THEN CAST('Concurrency Scaling elegible query - Failed to prepare cluster ' AS text)
         WHEN (    (sq.concurrency_scaling_status = 0)
               AND sq.query IN (SELECT stl_burst_async_mark.query
                                FROM stl_burst_async_mark
                                WHERE (stl_burst_async_mark.event ~~ CAST(('%Mark%') AS text)))
               AND sq.query IN (SELECT stl_burst_async_mark.query
                                FROM stl_burst_async_mark
                                WHERE (stl_burst_async_mark.event ~~ CAST(('%Unmark%') AS text))))
           THEN CAST('Concurrency Scaling elegible query - Cluster was not prepared in time ' AS text)
         WHEN (sq.concurrency_scaling_status = 0)
           THEN CAST('Ran on the main cluster' AS text)
         WHEN (sq.concurrency_scaling_status = 2)
           THEN CAST('Concurrency Scaling not enabled' AS text)
```

```
WHEN (sq.concurrency_scaling_status = 4)
  THEN CAST('Concurrency Scaling ineligible query - System temporary table accessed' AS text)
WHEN (sq.concurrency_scaling_status = 5)
  THEN CAST('Concurrency Scaling ineligible query - User temporary table accessed' AS text)
WHEN (sq.concurrency_scaling_status = 6)
  THEN CAST('Concurrency Scaling ineligible query - System table accessed' AS text)
WHEN (sq.concurrency_scaling_status = 3)
  THEN CAST('Concurrency Scaling ineligible query - Query is an Unsupported DML' AS text)
WHEN (sq.concurrency_scaling_status = 7)
  THEN CAST('Concurrency Scaling ineligible query - No backup table accessed' AS text)
WHEN (sq.concurrency_scaling_status = 8)
  THEN CAST('Concurrency Scaling ineligible query - Zindex table accessed' AS text)
WHEN (sq.concurrency_scaling_status = 9)
  THEN CAST('Concurrency Scaling ineligible query - Query uses UDF' AS text)
WHEN (sq.concurrency_scaling_status = 10)
  THEN CAST('Concurrency Scaling ineligible query - Catalog tables accessed' AS text)
WHEN (sq.concurrency_scaling_status = 11)
  THEN CAST('Concurrency Scaling ineligible query - Dirty table accessed' AS text)
WHEN (sq.concurrency_scaling_status = 12)
  THEN CAST('Concurrency Scaling ineligible query - Direct dispatched query' AS text)
WHEN (sq.concurrency_scaling_status = 16)
  THEN CAST('Concurrency Scaling ineligible query - No tables accessed' AS text)
WHEN (sq.concurrency_scaling_status = 17)
  THEN CAST('Concurrency Scaling ineligible query - Spectrum queries are disabled' AS text)
WHEN (sq.concurrency_scaling_status = 18)
  THEN CAST('Concurrency Scaling ineligible query - Function not supported ' AS text)
WHEN (sq.concurrency_scaling_status = 19)
  THEN CAST('Concurrency Scaling ineligible query - Instance type not supported ' AS text)
WHEN (sq.concurrency_scaling_status = 20)
  THEN CAST('Concurrency Scaling ineligible query - Burst temporarily disabled ' AS text)
WHEN (sq.concurrency_scaling_status = 21)
  THEN CAST('Concurrency Scaling ineligible query - Unload queries are disabled ' AS text)
WHEN (sq.concurrency_scaling_status = 22)
  THEN CAST('Concurrency Scaling ineligible query - Unsupported unload type ' AS text)
WHEN (sq.concurrency_scaling_status = 23)
  THEN CAST('Concurrency Scaling ineligible query - Non VPC clusters cannot burst ' AS text)
WHEN (sq.concurrency_scaling_status = 24)
  THEN CAST('Concurrency Scaling ineligible query - VPCE not setup ' AS text)
WHEN (sq.concurrency_scaling_status = 25)
  THEN CAST('Concurrency Scaling failed query - Inelegible to rerun on main cluster due to failure handling not
WHEN (sq.concurrency_scaling_status = 26)
  THEN CAST('Concurrency Scaling failed query - Inelegible to rerun on main cluster due to concurrency scaling
WHEN (sq.concurrency_scaling_status = 27)
  THEN CAST('Concurrency Scaling failed query - Inelegible to rerun on main cluster due to results already retu
WHEN (sq.concurrency_scaling_status = 28)
  THEN CAST('Concurrency Scaling failed query - Inelegible to rerun on main cluster due to non retriable error
WHEN (sq.concurrency_scaling_status = 29)
  THEN CAST('Concurrency Scaling failed query - Elegible to rerun on main cluster ' AS text)
WHEN (sq.concurrency_scaling_status = 30)
  THEN CAST('Concurrency Scaling inelegible query - Cumulative time not met ' AS text)
WHEN (sq.concurrency_scaling_status = 31)
  THEN CAST('Concurrency Scaling inelegible query - Paused query ' AS text)
WHEN (sq.concurrency_scaling_status = 32)
  THEN CAST('Query assigned to non Concurrency Scaling queue ' AS text)
WHEN (sq.concurrency_scaling_status = 33)
  THEN CAST('Concurrency Scaling ineligible query - Query has state on Main cluster ' AS text)
WHEN (sq.concurrency_scaling_status = 34)
  THEN CAST('Concurrency Scaling ineligible query - Query is inelegible for bursting Volt CTAS ' AS text)
WHEN (sq.concurrency_scaling_status = 35)
  THEN CAST('Concurrency Scaling ineligible query - Resource blacklisted ' AS text)
WHEN (sq.concurrency_scaling_status = 36)
  THEN CAST('Concurrency Scaling ineligible query - Non-retryable VoltTT queries are blacklisted ' AS text)
```

```
WHEN (sq.concurrency_scaling_status = 37)
  THEN CAST('Concurrency Scaling ineligible query - Query is retrying on Main cluster ' AS text)
WHEN (sq.concurrency_scaling_status = 38)
  THEN CAST('Concurrency Scaling ineligible query - Cannot burst Volt-created CTAS using cursors ' AS text)
WHEN (sq.concurrency_scaling_status = 39)
  THEN CAST('Concurrency Scaling usage limit reached ' AS text)
WHEN (sq.concurrency_scaling_status = 40)
  THEN CAST('Concurrency Scaling ineligible query - Unsupported VoltTT Utility query ' AS text)
WHEN (sq.concurrency_scaling_status = 41)
  THEN CAST('Concurrency Scaling ineligible query - Write query generating Volt TTs ' AS text)
WHEN (sq.concurrency_scaling_status = 42)
  THEN CAST('Concurrency Scaling ineligible query - VoltTT query with invalid state ' AS text)
WHEN (sq.concurrency_scaling_status = 43)
  THEN CAST('Concurrency Scaling ineligible query - Explain query generating Volt TTs ' AS text)
WHEN (sq.concurrency_scaling_status = 44)
  THEN CAST('Concurrency Scaling ineligible query - Bursting Volt-generated queries is disabled ' AS text)
WHEN (sq.concurrency_scaling_status = 45)
  THEN CAST('Concurrency Scaling ineligible query - Resource of VoltTT UNLOAD is blacklisted ' AS text)
WHEN (sq.concurrency_scaling_status = 46)
  THEN CAST('Concurrency Scaling ineligible query - Multiple pre-Volt query trees ' AS text)
WHEN (sq.concurrency_scaling_status = 48)
  THEN CAST('Concurrency Scaling ineligible query - Target table is DistAll/DistAutoAll ' AS text)
WHEN (sq.concurrency_scaling_status = 49)
  THEN CAST('Concurrency Scaling ineligible query - Table that has diststyle changed in current txn accessed '
WHEN (sq.concurrency_scaling_status = 50)
  THEN CAST('Concurrency Scaling ineligible query - Cannot burst spectrum copy ' AS text)
WHEN (sq.concurrency_scaling_status = 51)
  THEN CAST('Concurrency Scaling ineligible query - Dirty transaction tables accessed  ' AS text)
WHEN (sq.concurrency_scaling_status = 52)
  THEN CAST('Concurrency Scaling ineligible query - Table that has identity column as a target table ' AS text)
WHEN (sq.concurrency_scaling_status = 53)
  THEN CAST('Concurrency Scaling ineligible query - Datasharing remote tables accessed' AS text)
WHEN (sq.concurrency_scaling_status = 54)
  THEN CAST('Concurrency Scaling ineligible query - Target table with comp update' AS text)
WHEN (sq.concurrency_scaling_status = 55)
  THEN CAST('Concurrency Scaling ineligible query - Nested tables accessed' AS text)
WHEN (sq.concurrency_scaling_status = 56)
  THEN CAST('Concurrency Scaling ineligible query - Copy from EMR ' AS text)
WHEN (sq.concurrency_scaling_status = 59)
  THEN CAST('Concurrency Scaling ineligible query - Table that has column encode changed in current txn accesse
WHEN (sq.concurrency_scaling_status = 60)
  THEN CAST('Concurrency Scaling ineligible query - MV refresh disabled ' AS text)
WHEN (sq.concurrency_scaling_status = 61)
  THEN CAST('Concurrency Scaling ineligible query - Too many concurrent writes ' AS text)
WHEN (sq.concurrency_scaling_status = 62)
  THEN CAST('Concurrency Scaling ineligible query - Main cluster too big for writes ' AS text)
WHEN (sq.concurrency_scaling_status = 63)
  THEN CAST('Concurrency Scaling ineligible query - Datasharing VoltTT ' AS text)
WHEN (sq.concurrency_scaling_status = 64)
  THEN CAST('Concurrency Scaling ineligible query - Target table has super/geo column ' AS text)
WHEN (sq.concurrency_scaling_status = 65)
  THEN CAST('Concurrency Scaling rejected query - Ineligible to queue on burst cluster, eligible to rerun on an
WHEN (sq.concurrency_scaling_status = 66)
  THEN CAST('Concurrency Scaling ineligible query - Datasharing with burst-write' AS text)
WHEN (sq.concurrency_scaling_status = 67)
  THEN CAST('Concurrency Scaling ineligible query - CTAS with burst-write' AS text)
WHEN (sq.concurrency_scaling_status = 68)
  THEN CAST('Concurrency Scaling ineligible query - COPY on tables with identity columns' AS text)
WHEN (sq.concurrency_scaling_status = 70)
  THEN CAST('Concurrency Scaling ineligible query - Datasharing query with cursor' AS text)
WHEN (sq.concurrency_scaling_status = 71)
  THEN CAST('Ran on a Multi-AZ secondary cluster' AS text)
```

```
        WHEN (sq.concurrency_scaling_status = 72)
          THEN CAST('Concurrency Scaling ineligible query - Burst MERGE is disabled' AS text)
        WHEN (sq.concurrency_scaling_status = 73)
          THEN CAST('Concurrency Scaling ineligible query - Redshift Table via Lake Formation accessed' AS text)
        WHEN (sq.concurrency_scaling_status = 74)
          THEN CAST('Concurrency Scaling ineligible query - COPY on table with default value column' AS text)
        ELSE CAST('Concurrency Scaling ineligible query - Unknown status' AS text)
      END AS concurrency_scaling_status_txt,
      sq.starttime,
      sq.endtime
FROM stl_query AS sq
     LEFT JOIN stl_result_cache_history AS sr ON sq.query = sr.cache_hit_query
     LEFT JOIN stl_wlm_query AS sw ON (sq.query = sw.query)
                                  AND (sq.userid = sw.userid)
                                  AND (sq.xid = sw.xid)
```

Appendix D : Notes on the Test Script

Unusually, this script does not configure the Redshift environment and AWS networking environment before running - I am using my own existing RS and network configuration.

Usually my scripts create and then remove all necessary configuration, which is non-trivial - it took a full, solid, non-stop week of agonizing experimentation to figure that out - but where Serverless is new and also the networking setup is more complex, firstly, and primarily, I simply cannot face the emotional agony involved in working with AWS networking configuration via `boto`, and secondly, where `boto` is not reliable, and the networking configuration is complex, I begin to have a concern about using it to make significant changes to other peoples configurations.

I really, really, really want this script to automatically perform all configuration (and that's complex - AWS have made a complete mess of this, and so you have to deal with multiple possible scenarios), but it was a month of hell - pure and ever increasing agony - trying to get this script working in the first place. It was like trying to rent in Amsterdam. I'm not a robot, I'm human, and I have limits, and I've reached my uttermost limit for working with `boto` and Serverless.

So, if you want to run this script, you can see the command line arguments, you know what the script needs, and either you know how to configure AWS networking correctly such that you have or can create the necessary setup, or you do not.

I will walk anyone and everyone who does want to run this script through what has to be done. That's fine - it'll be done via the console, not `boto`.

I actually have a largely complete PDF which covers AWS networking setup for RS, via console and also via `boto`, which I need to review and get out the door.