Cross-Region Benchmarks

Max Ganz II @ Redshift Research Project

28th January 2022 (updated 1st May 2024)

Abstract

Within the resolving power of the benchmark suite used, Redshift node types are identical in performance across regions, with the single exception of the `dc2.large` node type, which is much slower in five regions; `af-south-1`, `ap-east-1`, `ap-southeast-3`, `eu-south-1`, and `me-south-1`.

Contents

Introduction

AWS provide in the documentation the specification of each type of Redshift node, and the specification is given in terms of processing power, memory in gigabytes, storage capacity, I/O (disk or network is not clear) and the price of the node type.

There are here two matters of interest; how do nodes compare to each other, and does node performance vary by region.

If the performance of a node type significantly varies by region, there are in effect additional node types, and so first we need to know if node types are consistent across regions.

Comparing nodes involves creating a benchmark suite and running that suite on the different node types.

Comparing regions involves running the benchmark suite, on the different node types, in each AWS region.

With regard to differences by region, in the documentation, the only aspect of specification which varies by region is price.

In a soon to be published white paper investigating query compilation, I stumbled across the first evidence that node performance could vary significantly by region.

If this is the case, it is important. If a cluster costs say 100k per year in a slow region, it might cost only (say) 85k per year in a fast region, because it would need fewer nodes.

However, I note prices vary by region, and it might be price differences in fact reflect (at least in part) performance differences.

This white paper then implements a benchmark suite and runs it on the different node types, and in each region, and examines cross-region performance.

Although this naturally provides benchmark data for the different node types, this white paper does not discuss cross-node performance, because first it must be seen if node types are consistent across regions, and because this is itself a large topic which needs a white paper and plenty of investigation of its own; any single white paper should be compact and focused.

Test Method

To begin with, in each region being tested, which is all of them except China and US Government, a two node cluster of each node type being tested (normally the three inexpensive small types, but with the expensive large types being tested in two regions, so at least some numbers for them exist) is brought up.

Each benchmark in the benchmark suite is then run five times, with the slowest and fastest result being discarded (actually of course, all data is retained, but the slowest and fastest are not used when computing statistics).

The clusters are then shut down.

There are four benchmarks, two for disk, one for network and one for processor, which will be described in detail.

All benchmarks produce a time in seconds, where the shorter the time taken, the better the performance of the node type.

Now, it became apparent during development that you cannot in fact really test a cluster.

What I mean by this is that a cluster is composed of nodes, and the nodes are basically EC2 instances, and they can vary a *lot* in individual performance (it's not out of the ordinary to see a node taking twice as long as another node, for exactly the same work).

A query issued to a cluster normally runs on all nodes, and so completes only when the slowest node finishes working.

Benchmarking a cluster - issuing benchmarking queries to the cluster as a whole - actually really means benchmarking the slowest node.

This made me realize that what I really had to do was benchmark *nodes*. It is the population of nodes, which are drawn upon to form clusters, and their spectrum of performance, which are actually of interest.

A second (and unsurprising) observation during development was that all the slices on a node run at about the same speed. If that node is slow, all its slices are slow; if that node is fast, all its slices are fast. Slices are after all simply the group of processes which are a query.

Accordingly, the benchmarks run on a single slice per node, but in parallel and independently on all the nodes in a cluster, so we produce as many benchmarks of individual nodes as possible from a cluster; the more benchmarks, the more

data, the better. (There are one or two exceptions to this, where we actually or effectively end up with less than one slice per node, which will be explained when the benchmarks are described).

For example, the disk (read/write) benchmark reads 512mb of data and then writes 512mb of data. This is done on a single slice on every node, with the tables being read and written using key distribution to keep all their values on a single slice. No network traffic occurs; the nodes operate wholly independently, and we time only the single segment which performed the `scan` and `insert`.

It is then that the results being published, which are the mean duration and the standard deviation of that mean on the basis of date (day resolution), region, node type, benchmark type, have the mean being computed from *all* individual node benchmarks of that type produced on that day, in that region, for that node type, regardless of the cluster the node was in.

Note however that each benchmark runs five times, with the slowest and fastest result being discarded, and this discarding of results *is* performed on a per-cluster basis. This is because we are trying to find a reasonable performance figure for those nodes at that time; it might have been some spurious activity specific to that cluster (Redshift itself runs a metric ton of its own queries, all the time) happened to impact a particular run of the benchmark and it is this type of spurious variance we look to eliminate.

The alternative is to gather up all the benchmark results of the same type and *then* cull the slowest and fastest results - but imagine we *did* have a cluster composed of nodes which were all genuinely slow, and in fact had no spurious variance at all, all the benchmark run results would because of the slowness of the node would fall into the extreme end of the range of results *for all nodes* and be culled, and we would have eliminated a set of perfectly valid benchmark results.

With regard to timing the benchmarks, timings are taken from the Redshift system tables, and so there is none of the uncertainty and variability introduced by client-side timing.

The timing offered by Redshift is quite limited. What's available are the start and end times of each segment, on a per-slice basis. The benchmarks have been designed so that the work being benchmarked occurs entirely in the first segment, and so the timing is the time taken for the first segment. Of course, all segments run in parallel, but in the benchmarks the later segments are by the work being done by the benchmark queries blocked from doing work, and so are idle, until the first segment completes.

All clusters are configured with one queue of five slots which has 100% memory, SQA off, CSC off, auto-analyze off, query rewrite to use MVs off (although, of course, auto-creation of now totally unused MVs cannot be disabled and so could be messing up these benchmarks - but there's absolutely nothing which can be done about this, or about auto-vacuum, which also cannot be disabled.)

Note in all cases the benchmark is a single query, which operates all nodes concurrently, and we then procure from the system tables timings for each node (which is to say, for each slice, and there is normally one per node, which is running the benchmark). Multiple concurrent queries are only issued during

the benchmark setup phase, where the tables used by the benchmarks are being populated with rows.

Before I begin to describe the benchmarks in detail, note that the benchmarks are not very pure; by this I mean to say it's usually impossible to benchmark *only* disk, or *only* network, or *only* processor, because a benchmark is necessarily an SQL query, which is quite a blunt instrument when it comes to benchmarking, as almost any query will involve all three.

I have striven to produce benchmarks which are as pure as possible and in this, the first disk benchmark is pure (it is disk read only), the second disk benchmark is impure (it performs disk read and disk write, because there is no way to obtain write without read), the network benchmark is impure as it performs a disk read then a network distribution, and the processor benchmark is pure.

As such, when considering the second and third benchmarks, you need to mentally take into account the performance of the disk read benchmark, and subtract that. (This cannot be done automatically, because timings in the system tables are only available on a per-segment basis, and the second and third benchmarks both perform their `scan` in the same segment they perform the work we're really interested in (disk write/network activity), so I cannot know how long the `scan` step took - all I can do is make a benchmark for `scan` steps on their own, and then provide that benchmark to the reader.)

However, in any event, pure or impure, where the benchmarks are identical across regions and node types, they *do* perfectly well allow for reasonable meaningful comparisons between nodes, and between regions; if a given query can be seen normally to take say, ten seconds in one region, but only five in another, something is most definitely going on, whether or not that query is pure.

Finally, I must explain the absence, for the first time, of the source code used to produce the evidence this white paper is based upon.

There are a number of reasons for this;

1. The code starts Redshift clusters, and does so in almost all regions. To start a Redshift cluster, either a default VPC and its associated objects must exist, or they must be created by the script. I have code to create a VPC and its associated objects, but there are reliability issues with `boto3` even with a single cluster in a single region; I would have to do a lot of debugging, expensive debugging, to reach the point where I had enough reliability for the code to be safe. As such, users must ensure they have configured all regions correctly for use with the script, which is a non-trivial requirement, both for them, and for me to document, because VPC configuration is a nightmare.

2. Where the code is not currently safe to create and delete so many VPCs, I current use, where possible, the default VPC. I have configured all the regions I use to have a default VPC (and Redshift configuration) and checked they are correct. However, I have two regions which have EC2 Classic and which have had their default VPC deleted; such regions cannot have a default VPC re-created. As such, my code uses default VPC in all but those two regions, where it uses a pre-configured (to avoid the reliability issues of creating and deleting VPCs and their associated objects)

non-default VPC. As such, I am not able to actually test the code other users would run, because I am unable to run code which uses a default VPC in all regions. Code which is not tested does not work.

3. The code spins us, when used to test all node types, about 150 clusters, including plenty of the big, expensive, 15 USD an hour nodes. This is not a toy. This is an expensive mistress who can leave you with a very large bill to pay. I would be comfortable releasing the code *if* I had strong confidence in `boto3`, which necessarily is used to start and stop clusters, and in the code which operates the clusters - but I absolutely do not. With so many nodes, all sorts of rare bugs emerge, such as the Redshift WLM process manager crashing and the query which caused it simply hanging forever. There is no way this code has the necessary reliability to be safe in the hands of whomever might download it.

4. The code, as it is, reads significant information from, and writes its results to, a Postgres database. I could rewrite the code so it runs without Postgres, but this is a non-trivial piece of work, and given the other issues, this contributes to the choice of not releasing the software with the white paper.

However, ***of course***, the software has to be available so other people can examine it for mistakes and the like, and as such is available on request. Email me "max dot ganz at redshiftresearchproject dot org".

Benchmarks

I am sensitive to the time taken for the benchmarks to run, because testing two node clusters of three node types over twenty-two regions is a lot of Redshift nodes and they cost money. (This white paper cost about 500 USD, as it took quite a few runs to debug reliability issues first with my code, and then with Redshift.)

Fortunately, AWS some time ago for Redshift moved from a minimum billing period of one hour to one second; without that, none of this would be happening.

The most time consuming work in a benchmark is populating the source tables used to provide data to the benchmark. Accordingly, as it is the most time efficient method I can see, it is then that a source table with a single column is set up and populated with exactly 256mb of data on the first slice only of each node, and this source table is used by all benchmarks (which in some cases controls which slices actually do work by specifying in a `where` clause which values to operate upon, which is properly tantamount to specifying which slices perform work).

Normally I would populate a source table from scratch for each benchmark and indeed, for each iteration of a benchmark.

(Note that although I write 256mb of data, necessarily 256mb of data is also written by Redshift, to the system managed `oid` column, which is an uncompressed `int8`, so a total of 512mb is written.)

The single column is the distribution key, and the value of each slice's 256mb

of data is that necessary for the data to be on the first slice of each node in the cluster (the benchmark determines these values during setup).

The amount of data is most definitely on the small side for the disk (read) benchmark (which really needs at least something like 32 gigabytes), but about right for the for the disk (read and write) operation and the networking benchmark.

The source table DDL;

```
create table source
(
  column_1  int8  not null encode raw distkey
)
diststyle key;
```

Note the table is unsorted. Remember that each row is being specifically directed to a given slice by having a particular, chosen value written; so all the rows on any slice always have the same value. Sorting is irrelevant and a complicating factor.

Unfortunately, I forgot that the devs made what appears to be an amateur-hour blunder in the introduction of `sortkey auto`. It used to be if a sortkey was not specified, the table was unsorted. Now instead, when sorting is not specified the table is made `auto` - so Redshift can change the sort order at any time. However, I often specifically want unsorted tables, because with small tables you save a lot of disk space. It looks to me this functionality was *inadvertently removed*, which is mind-blowing.

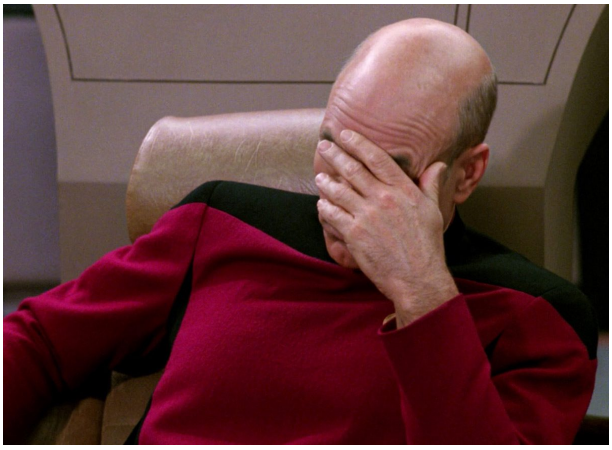

Figure 1: You did *what?*

As it is, all the benchmarks have been producing the same results over many runs, over many regions, and so I do not think table re-sorting has always just happened to be happening at the right moment to mess things up. In the future though, I'll handle this by using a compound sorted table.

Disk (read)

The benchmark simply issues a `count( column_1 )` on the source table, but with a `where` clause which restriction the count to rows on the first slice of the first node.

As such only one benchmark is produced per cluster.

The reason for this is that changing the `where` clause to indicate more than one slice *completely* changes the step-plan, with the work being done being utterly different (and not useful in any way for the purpose of this benchmark).

I hope to improve this benchmark, but it suffices for now.

The benchmark SQL;

```
select
  count( column_1 )
from
  source
where
  column_1 = 5;
```

The value used in the `where` clause instructs the query to select from and only from the first slice on the first node; here the example value 5 is used.

The step-plan of the query;

qid	stream	segment	step	node_id	slice_id	step_type	rows	bytes	seg_start_time	seg_duration	schematable_name	notes
383	0	0	0	0	0	scan	33534464	536551424	19:50:27.371562	0.145862	public.source	scan data from user table
383	0	0	1	0	0	project	33534464		19:50:27.371562	0.145862		
383	0	0	2	0	0	project	33534464		19:50:27.371562	0.145862		
383	0	0	3	0	0	aggregate	1	8	19:50:27.371562	0.145862		ungrouped, scalar aggregation
383	1	1	0	0	0	scan	1	8	19:50:27.519093	8.4e-05		scan data from temp table
383	1	1	1	0	0	return	1	8	19:50:27.519093	8.4e-05		
383	1	2	0		12811	scan	1	8	19:50:27.518628	0.000578	pg_catalog.pg_tablespace	scan data from network to tem
383	1	2	1		12811	aggregate	1	16	19:50:27.518628	0.000578		ungrouped, scalar aggregation
383	2	3	0		12811	scan	1	16	19:50:27.520323	5.9e-05		scan data from temp table
383	2	3	1		12811	project	1		19:50:27.520323	5.9e-05		
383	2	3	2		12811	project	1		19:50:27.520323	5.9e-05		
383	2	3	3		12811	return	1	14	19:50:27.520323	5.9e-05		

The start and end times of the first segment are the timings produced by the benchmark.

As you can see, only a single slice on a single node participates in the disk read.

Disk (read and write)

Ideally, a benchmark would be provided for reading disk and a benchmark would be provided for writing disk.

As far as I can make out, it is impossible to produce a query which writes to disk without first reading from disk; a pure write benchmark cannot be produced.

When considering the timings then from this benchmark, reference must be made to the disk (read) benchmark, to give a feeling for how much of the time taken is being consumed by the read part of this benchmark.

The benchmark creates an empty destination table identical to the source table, where the benchmark query inserts the contents of the source table into the destination table. This is a single query, but where every slice is inserting rows

which are distributed to it, to another table which also distributes those rows to it, no network traffic occurs. All work is local.

The benchmark DDL and SQL;

```
create table destination
(
  column_1  int8  not null encode raw distkey
)
diststyle key;

insert into
  destination( column_1 )
select
  column_1
from
  source;
```

The step-plan of the benchmark;

qid	stream	segment	step	node_id	slice_id	step_type	rows	bytes	seg_start_time	seg_duration	schematable_name	notes
369	0	0	0	0	0	scan	33534464	536551424	19:50:15.708894	4.942746	public.source	scan data from user table
369	0	0	0	0	1	scan	0	0	19:50:15.708894	0.000584	public.source	scan data from user table
369	0	0	0	1	2	scan	33534464	536551424	19:50:15.710429	4.551398	public.source	scan data from user table
369	0	0	0	1	3	scan	0	0	19:50:15.710083	0.000453	public.source	scan data from user table
369	0	0	1	0	0	project	33534464		19:50:15.708894	4.942746		
369	0	0	1	0	1	project	0		19:50:15.708894	0.000584		
369	0	0	1	1	2	project	33534464		19:50:15.710429	4.551398		
369	0	0	1	1	3	project	0		19:50:15.710083	0.000453		
369	0	0	2	0	0	project	33534464		19:50:15.708894	4.942746		
369	0	0	2	0	1	project	0		19:50:15.708894	0.000584		
369	0	0	2	1	2	project	33534464		19:50:15.710429	4.551398		
369	0	0	2	1	3	project	0		19:50:15.710083	0.000453		
369	0	0	4	0	0	project	33534464		19:50:15.708894	4.942746		
369	0	0	4	0	1	project	0		19:50:15.708894	0.000584		
369	0	0	4	1	2	project	33534464		19:50:15.710429	4.551398		
369	0	0	4	1	3	project	0		19:50:15.710083	0.000453		
369	0	0	5	0	0	insert	33534464		19:50:15.708894	4.942746		
369	0	0	5	0	1	insert	0		19:50:15.708894	0.000584		
369	0	0	5	1	2	insert	33534464		19:50:15.710429	4.551398		
369	0	0	5	1	3	insert	0		19:50:15.710083	0.000453		
369	0	0	6	0	0	aggregate	1	8	19:50:15.708894	4.942746		ungrouped, scalar aggregation in memo
369	0	0	6	0	1	aggregate	1	8	19:50:15.708894	0.000584		ungrouped, scalar aggregation in memo
369	0	0	6	1	2	aggregate	1	8	19:50:15.710429	4.551398		ungrouped, scalar aggregation in memo
369	0	0	6	1	3	aggregate	1	8	19:50:15.710083	0.000453		ungrouped, scalar aggregation in memo
369	1	1	0	0	0	scan	1	8	19:50:20.699767	0.000111		scan data from temp table
369	1	1	0	0	1	scan	1	8	19:50:20.700102	9.9e-05		scan data from temp table
369	1	1	0	1	2	scan	1	8	19:50:20.665446	0.001381		scan data from temp table
369	1	1	0	1	3	scan	1	8	19:50:20.666812	8.8e-05		scan data from temp table
369	1	1	1	0	0	return	1	8	19:50:20.699767	0.000111		
369	1	1	1	0	1	return	1	8	19:50:20.700102	9.9e-05		
369	1	1	1	1	2	return	1	8	19:50:20.665446	0.001381		
369	1	1	1	1	3	return	1	8	19:50:20.666812	8.8e-05		
369	1	2	0		12813	scan	4	32	19:50:20.662658	0.037684		scan data from network to temp table
369	1	2	1		12813	aggregate	1	16	19:50:20.662658	0.037684		ungrouped, scalar aggregation in memo
369	2	3	0		12813	scan	1	16	19:50:20.701237	4.4e-05		scan data from temp table
369	2	3	1		12813	return	0	0	19:50:20.701237	4.4e-05		

The start and end times of the first segment are the timings produced by the benchmark.

The first segment begins with the `scan` and finishes with the `aggregate` which is used by Redshift to compute the number of inserted rows (and is, as we see from the `notes` column, in memory and so will not distort the time taken for the query, which is massively dominated by disk I/O).

You will note segment 0 is missing step 3. You will in the next benchmark notice *two* steps are missing from its query plan.

I manually examined all the low-level `STL_*` views/tables used to produce the step-plan, and they were all correct, recording what is seen here, no more and no less. I then checked for any new, extra low-level `STL_*` views/tables, in

the system table explorer, there are none. I then examined SVL_QUERY_REPORT, the completely broken (in a tremendous blunder, is has no column for stream, so you can never order the rows correctly - clearly, whoever made this view never used it, nor was checked by anyone using it for real) Redshift system-tables' equivalent of a step-plan view; it too is missing the steps my view, which creates the step-plans used in this document, is missing.

So, what's going on?

Well, either step numbers are and without warning no longer contiguous, or logging is now broken and failing to log some steps, or new step types have been introduced, but without a system table for logging.

Every single one of these options paints the dev team in a bad light, the only difference being some are worse than others.

Not being able to see all step-types would be the most catastrophic outcome, as it would no longer be possible to actually know what how a query is working.

Moving on, we see one slice per node is being benchmarked.

Note the slice ID 12811 (which varies a little) is the leader node. This used to be 6411, but it's changed moderately recently and is now not actually a fixed number, just to make reading and using the system tables that bit more awkward.

Network

Network benchmark as with disk write proved impossible, because I could find no way to separate a networking step (`broadcast`, `distribute` or `return`) from a `scan`.

This benchmark then has a query which in its first segment reads all the rows on one slice per node, and then emits the rows being read to one slice on another node. The sender and receiver nodes are different nodes, and the sender only sends and the receiver only receives. As such, unlike the disk test (where each node produces one benchmark), here we need two nodes to produce one benchmark.

As with the disk (read and write) benchmark, reference must be made to the disk (read) benchmark, to give a feeling for how much of the time taken is being consumed by the read part of this benchmark.

The benchmark DDL and SQL;

```
create table destination
(
  column_1  int8  not null encode raw distkey
)
diststyle key;

insert into
  destination( column_1 )
select
  case
```

```
    when 5 then 2
  end as column_1
from
  source
where
  column_1 in ( 5 );
```

The code determines, for each slice, a value which will distribute to that slice.

The `insert` uses a case to convert the distribution value for the source slice (which is on one node), to that of the destination slice (which is another node), and so allows in a single `insert` for all node pairs to produce a benchmark.

The step-plan of the benchmark;

qid	stream	segment	step	node_id	slice_id	step_type	rows	bytes	seg_start_time	seg_duration	schematable_name	notes
404	0	0	0	0	0	scan	33534464	536551424	19:51:02.939158	5.199308	public.source	scan data from user table
404	0	0	0	0	1	scan	0	0	19:51:02.939158	0.00044	public.source	scan data from user table
404	0	0	0	1	2	scan	0	0	19:51:02.939497	0.000585	public.source	scan data from user table
404	0	0	0	1	3	scan	0	0	19:51:02.939511	0.000571	public.source	scan data from user table
404	0	0	1	0	0	project	33534464		19:51:02.939158	5.199308		
404	0	0	1	0	1	project	0		19:51:02.939158	0.00044		
404	0	0	1	1	2	project	0		19:51:02.939497	0.000585		
404	0	0	1	1	3	project	0		19:51:02.939511	0.000571		
404	0	0	2	0	0	project	33534464		19:51:02.939158	5.199308		
404	0	0	2	0	1	project	0		19:51:02.939158	0.00044		
404	0	0	2	1	2	project	0		19:51:02.939497	0.000585		
404	0	0	2	1	3	project	0		19:51:02.939511	0.000571		
404	0	0	5	0	0	distribute	33534464	268269024	19:51:02.939158	5.199308		
404	0	0	5	0	1	distribute	0	0	19:51:02.939158	0.00044		
404	0	0	5	1	2	distribute	0	0	19:51:02.939497	0.000585		
404	0	0	5	1	3	distribute	0	0	19:51:02.939511	0.000571		
404	0	1	0	0	0	scan	0	0	19:51:02.937737	5.201228		scan data from network to temp table
404	0	1	0	0	1	scan	0	0	19:51:02.937737	5.201228		scan data from network to temp table
404	0	1	0	1	2	scan	33534464	268275712	19:51:02.938155	5.792375		scan data from network to temp table
404	0	1	0	1	3	scan	0	0	19:51:02.938162	5.202014		scan data from network to temp table
404	0	1	1	0	0	project	0		19:51:02.937737	5.201228		
404	0	1	1	0	1	project	0		19:51:02.937737	5.201228		
404	0	1	1	1	2	project	33534464		19:51:02.938155	5.792375		
404	0	1	1	1	3	project	0		19:51:02.938162	5.202014		
404	0	1	2	0	0	insert	0		19:51:02.937737	5.201228		
404	0	1	2	0	1	insert	0		19:51:02.937737	5.201228		
404	0	1	2	1	2	insert	33534464		19:51:02.938155	5.792375		
404	0	1	2	1	3	insert	0		19:51:02.938162	5.202014		
404	0	1	3	0	0	aggregate	1	8	19:51:02.937737	5.201228		ungrouped, scalar aggregation in mem
404	0	1	3	0	1	aggregate	1	8	19:51:02.937737	5.201228		ungrouped, scalar aggregation in mem
404	0	1	3	1	2	aggregate	1	8	19:51:02.938155	5.792375		ungrouped, scalar aggregation in mem
404	0	1	3	1	3	aggregate	1	8	19:51:02.938162	5.202014		ungrouped, scalar aggregation in mem
404	1	2	0	0	0	scan	1	8	19:51:08.742505	0.000178		scan data from temp table
404	1	2	0	0	1	scan	1	8	19:51:08.744217	0.000101		scan data from temp table
404	1	2	0	1	2	scan	1	8	19:51:08.789994	8.3e-05		scan data from temp table
404	1	2	0	1	3	scan	1	8	19:51:08.789994	0.000113		scan data from temp table
404	1	2	1	0	0	return	1	8	19:51:08.742505	0.000178		
404	1	2	1	0	1	return	1	8	19:51:08.744217	0.000101		
404	1	2	1	1	2	return	1	8	19:51:08.789994	8.3e-05		
404	1	2	1	1	3	return	1	8	19:51:08.789994	0.000113		
404	1	3	0		12813	scan	4	32	19:51:08.74055	0.049247		scan data from network to temp table
404	1	3	1		12813	aggregate	1	16	19:51:08.74055	0.049247		ungrouped, scalar aggregation in mem
404	2	4	0		12813	scan	1	16	19:51:08.791234	4.7e-05		scan data from temp table
404	2	4	1		12813	return	0	0	19:51:08.791234	4.7e-05		

The start and end times of the first segment are the timings produced by the benchmark.

The first segment begins with the `scan` and finishes with the `distribute`.

You will note here *two* steps are missing, steps 3 and 4 in segment 0.

Finally, if you look at the first segment, all its work is on slice 0 of node 0, but then after the `distribute` work moves to slice 2 of node 1, which is the destination node.

Processor

As ever, the aim and the difficulty is producing a benchmark which tests only the processor.

Now, the leader node is identical in hardware to the worker nodes.

A processor test which runs only on the leader node tells us perfectly well the performance of the node type.

The leader-node, being descending from Postgres, provides a wide range of functionality not available on the worker nodes and this includes the function `generate_series()`, which is used to produce rows.

The benchmark code then uses `generate_series()` to produce in a subquery a large number of rows, with the outer query performing a `count(*)` and returning this total (thus very nearly eliminating network traffic).

This is to my eye a pure processor test.

I think it will be single-threaded, and so not test the extent to which the processor supports multiple concurrent threads in hardware, and so is providing a direct comparison between single-thread performance across node types.

The benchmark code;

```
select
  -1                              as node_id,
  -1                              as slice_id,
  sysdate::timestamp              as transaction_start_time_utc,
  timeofday()::varchar::timestamp as query_end_time_utc,
  count(*)                        as number_rows
from
  (
    select
      generate_series( 0, power(2,23)::int8, 1 )
  ) as gs;
```

The inner query does all the work. The outer query provide timestamps for the start of the transaction and the time the single output row was emitted. The `node_id` and `slice_id` are set to -1 to signify the leader-node.

A step-plan cannot be provided as the benchmark runs wholly on the leader-node.

Results

These results are available on-line, where they will intermittently be updated, to track performance over time.

The results here as for the first complete benchmark run, dated 2022-01-24.

All results are times in seconds, with shorter times meaning higher performance.

See Appendix A for the Python `pprint` dump of the results dictionaries (one per region, per node type).

af-south-1

node	disk(r)	disk(rw)	network	processor
dc2.large	0.72/0.01	14.83/1.32	12.87/0.09	3.55/0.00
ds2.xlarge	0.14/0.00	4.53/0.02	5.63/0.05	3.46/0.02

ap-east-1

node	disk(r)	disk(rw)	network	processor
dc2.large	0.72/0.00	13.25/0.07	12.82/0.04	3.61/0.04
ds2.xlarge	0.14/0.00	4.58/0.02	5.38/0.06	3.44/0.01
ra3.xlplus	0.10/0.00	3.83/0.01	3.26/0.04	2.69/0.00
dc2.8xlarge	0.15/0.00	4.61/0.02	3.62/0.00	3.51/0.00
ds2.8xlarge	0.14/0.00	4.56/0.01	3.75/0.01	3.81/0.01
ra3.4xlarge	0.10/0.00	3.80/0.02	2.96/0.01	2.73/0.01
ra3.16xlarge	0.10/0.00	3.67/0.01	2.93/0.00	2.53/0.00

ap-northeast-1

node	disk(r)	disk(rw)	network	processor
dc2.large	0.15/0.00	4.74/0.18	5.29/0.06	3.66/0.12
ds2.xlarge	0.14/0.00	4.48/0.03	5.30/0.07	3.44/0.01

node	disk(r)	disk(rw)	network	processor
ra3.xlplus	0.11/0.00	3.85/0.02	3.23/0.03	2.71/0.00

ap-northeast-2

node	disk(r)	disk(rw)	network	processor
dc2.large	0.15/0.00	4.61/0.09	8.26/0.08	3.57/0.03
ds2.xlarge	0.14/0.00	4.59/0.01	6.98/0.02	3.42/0.01
ra3.xlplus	0.10/0.00	3.88/0.03	3.17/0.02	2.78/0.01

ap-northeast-3

node	disk(r)	disk(rw)	network	processor
dc2.large	0.15/0.00	4.67/0.11	7.19/0.07	3.53/0.07
ds2.xlarge	0.14/0.00	4.54/0.02	5.74/0.03	3.44/0.01

ap-south-1

node	disk(r)	disk(rw)	network	processor
dc2.large	0.15/0.00	4.44/0.08	5.04/0.00	3.67/0.03
ds2.xlarge	0.14/0.00	4.39/0.03	5.12/0.03	3.52/0.01
ra3.xlplus	0.11/0.00	3.62/0.05	3.00/0.02	2.79/0.01

ap-southeast-1

node	disk(r)	disk(rw)	network	processor
dc2.large	0.15/0.00	5.25/1.35	5.02/0.02	3.55/0.03
ds2.xlarge	0.14/0.00	4.58/0.04	5.74/0.00	3.48/0.03
ra3.xlplus	0.10/0.00	3.87/0.01	3.37/0.08	2.63/0.01

ap-southeast-2

node	disk(r)	disk(rw)	network	processor
dc2.large	0.16/0.00	4.67/0.12	5.25/0.06	3.51/0.01
ds2.xlarge	0.14/0.00	4.61/0.05	5.83/0.04	3.43/0.02
ra3.xlplus	0.11/0.00	3.92/0.03	3.31/0.02	2.76/0.00

ap-southeast-3

node	disk(r)	disk(rw)	network	processor
dc2.large	0.73/0.00	13.33/0.20	13.12/0.01	3.51/0.01

ca-central-1

node	disk(r)	disk(rw)	network	processor
dc2.large	0.15/0.00	4.78/0.30	5.28/0.10	3.55/0.01
ds2.xlarge	0.14/0.00	4.62/0.04	5.76/0.03	3.41/0.01
ra3.xlplus	0.11/0.00	3.91/0.04	3.26/0.07	2.78/0.04

cn-north-1

No data.

cn-northwest-1

No data.

eu-central-1

node	disk(r)	disk(rw)	network	processor
dc2.large	0.15/0.00	4.46/0.08	4.99/0.01	3.61/0.02
ds2.xlarge	0.14/0.00	4.38/0.02	5.04/0.02	3.54/0.00
ra3.xlplus	0.10/0.00	3.70/0.02	3.12/0.02	2.88/0.07

eu-north-1

node	disk(r)	disk(rw)	network	processor
dc2.large	0.15/0.00	4.62/0.09	5.16/0.17	3.57/0.02
ds2.xlarge	0.14/0.00	4.58/0.09	5.59/0.04	3.54/0.18
ra3.xlplus	0.10/0.00	3.85/0.01	3.24/0.03	2.69/0.01

eu-south-1

node	disk(r)	disk(rw)	network	processor
dc2.large	0.73/0.01	13.38/0.24	13.02/0.07	3.52/0.02

node	disk(r)	disk(rw)	network	processor
ds2.xlarge	0.14/0.00	4.54/0.01	5.47/0.02	3.42/0.00

eu-west-1

node	disk(r)	disk(rw)	network	processor
dc2.large	0.15/0.00	4.65/0.11	5.15/0.06	3.63/0.02
ds2.xlarge	0.14/0.00	4.49/0.04	4.99/0.07	3.55/0.20
ra3.xlplus	0.11/0.00	3.87/0.02	3.18/0.06	2.86/0.13

eu-west-2

node	disk(r)	disk(rw)	network	processor
dc2.large	0.15/0.00	4.78/0.28	5.20/0.04	3.47/0.01
ds2.xlarge	0.14/0.00	4.55/0.05	6.09/0.03	3.47/0.00
ra3.xlplus	0.11/0.00	3.92/0.04	3.25/0.01	2.86/0.14

eu-west-3

node	disk(r)	disk(rw)	network	processor
dc2.large	0.15/0.00	4.91/0.39	5.17/0.07	3.54/0.03
ds2.xlarge	0.14/0.00	4.59/0.07	5.71/0.04	3.37/0.01
ra3.xlplus	0.10/0.00	3.92/0.02	3.23/0.03	2.78/0.07

me-south-1

node	disk(r)	disk(rw)	network	processor
dc2.large	0.71/0.00	13.21/0.07	12.69/0.03	3.58/0.01
ds2.xlarge	0.14/0.00	4.56/0.03	5.66/0.01	3.42/0.01

sa-east-1

node	disk(r)	disk(rw)	network	processor
dc2.large	0.15/0.00	4.62/0.10	5.12/0.01	3.62/0.09
ds2.xlarge	0.14/0.00	4.62/0.09	5.62/0.03	3.40/0.02
ra3.xlplus	0.11/0.00	3.88/0.02	3.23/0.02	2.67/0.00

us-east-1

node	disk(r)	disk(rw)	network	processor
dc2.large	0.14/0.00	4.60/0.12	7.04/0.21	3.48/0.01
ds2.xlarge	0.14/0.00	4.52/0.04	7.48/0.05	3.40/0.00
ra3.xlplus	0.11/0.00	3.81/0.05	3.08/0.04	2.74/0.00
dc2.8xlarge	0.14/0.00	4.59/0.03	3.47/0.01	3.43/0.00
ds2.8xlarge	0.14/0.00	4.53/0.04	3.66/0.09	3.33/0.00
ra3.4xlarge	0.11/0.00	3.87/0.04	3.05/0.04	2.73/0.00
ra3.16xlarge	0.11/0.00	3.58/0.03	2.83/0.01	2.50/0.00

us-east-2

node	disk(r)	disk(rw)	network	processor
dc2.large	0.15/0.00	4.43/0.03	5.40/0.12	3.65/0.00
ds2.xlarge	0.14/0.00	4.45/0.08	5.03/0.07	3.54/0.01
ra3.xlplus	0.10/0.00	3.65/0.04	3.15/0.03	2.87/0.06

us-gov-east-1

No data.

us-gov-secret-1

No data.

us-gov-topsecret-1

No data.

us-gov-topsecret-2

No data.

us-gov-west-1

No data.

us-west-1

node	disk(r)	disk(rw)	network	processor
dc2.large	0.15/0.00	4.44/0.04	5.36/0.01	3.65/0.00
ds2.xlarge	0.14/0.00	4.43/0.04	6.04/0.10	3.56/0.02
ra3.xlplus	0.10/0.00	3.57/0.01	3.08/0.03	2.74/0.02

us-west-2

node	disk(r)	disk(rw)	network	processor
dc2.large	0.15/0.00	4.63/0.07	5.20/0.03	3.53/0.03
ds2.xlarge	0.14/0.00	4.47/0.03	5.28/0.09	3.55/0.02
ra3.xlplus	0.11/0.00	3.86/0.03	3.16/0.02	2.77/0.00

Discussion

The result which most immediately catches the eye is that in certain regions, the `dc2.large` node type is taking between two and five times as long to run benchmarks as in other regions.

The affected regions are;

region
af-south-1
ap-east-1
ap-southeast-3
eu-south-1
me-south-1

I normally test only the small node types, but I tested the large node types in two regions, so there would be at least some benchmarks for them, and I chose one region with slow `dc2.large` nodes, and one without.

The benchmarks for a typical affected region (`ap-east-1`);

node	disk(r)	disk(rw)	network	processor
dc2.large	0.72/0.00	13.25/0.07	12.82/0.04	3.61/0.04
ds2.xlarge	0.14/0.00	4.58/0.02	5.38/0.06	3.44/0.01
ra3.xlplus	0.10/0.00	3.83/0.01	3.26/0.04	2.69/0.00
dc2.8xlarge	0.15/0.00	4.61/0.02	3.62/0.00	3.51/0.00
ds2.8xlarge	0.14/0.00	4.56/0.01	3.75/0.01	3.81/0.01
ra3.4xlarge	0.10/0.00	3.80/0.02	2.96/0.01	2.73/0.01
ra3.16xlarge	0.10/0.00	3.67/0.01	2.93/0.00	2.53/0.00

And these for `us-east-1`, a normal region;

node	disk(r)	disk(rw)	network	processor
dc2.large	0.14/0.00	4.60/0.12	7.04/0.21	3.48/0.01
ds2.xlarge	0.14/0.00	4.52/0.04	7.48/0.05	3.40/0.00
ra3.xlplus	0.11/0.00	3.81/0.05	3.08/0.04	2.74/0.00
dc2.8xlarge	0.14/0.00	4.59/0.03	3.47/0.01	3.43/0.00

node	disk(r)	disk(rw)	network	processor
ds2.8xlarge	0.14/0.00	4.53/0.04	3.66/0.09	3.33/0.00
ra3.4xlarge	0.11/0.00	3.87/0.04	3.05/0.04	2.73/0.00
ra3.16xlarge	0.11/0.00	3.58/0.03	2.83/0.01	2.50/0.00

Only `dc2.large` is affected. Note the processor benchmark is not affected.

I have no idea why this is so. It is not obviously a property of the region, as other node types are unaffected.

I have no idea if the Redshift team are aware of this difference in performance, but it surely seems unthinkable they would not know; this would require a lack of performance telemetry, and I have the impression that Redshift clusters are up to their little electronic ears in telemetry.

Aside from this, Redshift node types appear, within the resolving power of the benchmarks, to be identical across regions.

As to how this is so; a region - a vast data center - is an enormous number of servers. At the time of writing AWS has about twenty-five data centers. I cannot imagine for one second every data center is going to be bang up to date with the latest hardware - it's simply impossible logistically, and hardly makes sense financially; you do not replace a server the moment the next incrementally improved server comes out.

However, AWS offer *virtual* servers and so the specifications of the virtual server are defined in software and, where the specifications are given in high-level terms - the number of physical processors, the amount of memory, disk space, etc - they gloss over the differences in hardware between regions.

In this way, virtual servers, despite the underlying hardware difference, turn out to be very similar; although it may well be some servers are running on more modern processors, or on faster memory, the impact of these differences is normally is much smaller than the impact of the high-level specifications.

I thought it might also be useful and interesting here, since we are after all now looking at node type performance, to list, ordered by price, the cost of each node type in USD per hour across regions, so you can see where you stand with your choice of region.

This data is obtained via the `pricing` API in `boto3` and reflects prices at the time of writing (the timestamp column is the timestamp the price is valid from).

For the `dc2.large` table, there is an additional column, `note`, where I indicate the five slow regions.

Table 26: dc2.large

node type	region	price	note
dc2.large	sa-east-1	0.4	
dc2.large	ap-east-1	0.363	slow
dc2.large	af-south-1	0.357	slow

node type	region	price	note
dc2.large	ap-southeast-1	0.33	
dc2.large	ap-southeast-2	0.33	
dc2.large	ap-southeast-3	0.33	slow
dc2.large	me-south-1	0.33	slow
dc2.large	us-west-1	0.33	
dc2.large	eu-central-1	0.324	
dc2.large	eu-west-2	0.32	
dc2.large	eu-west-3	0.32	
dc2.large	ap-south-1	0.315	
dc2.large	eu-south-1	0.315	slow
dc2.large	ap-northeast-1	0.314	
dc2.large	ap-northeast-3	0.314	
dc2.large	ap-northeast-2	0.3	
dc2.large	eu-west-1	0.3	
dc2.large	us-gov-east-1	0.3	
dc2.large	us-gov-west-1	0.3	
dc2.large	eu-north-1	0.285	
dc2.large	ca-central-1	0.275	
dc2.large	us-east-1	0.25	
dc2.large	us-east-2	0.25	
dc2.large	us-west-2	0.25	
dc2.large	cn-north-1	no data	
dc2.large	cn-northwest-1	no data	
dc2.large	us-gov-secret-1	no data	
dc2.large	us-gov-topsecret-1	no data	
dc2.large	us-gov-topsecret-2	no data	

Table 27: ds2.xlarge

node type	region	price
ds2.xlarge	ap-east-1	1.375
ds2.xlarge	sa-east-1	1.36
ds2.xlarge	ap-southeast-1	1.25
ds2.xlarge	ap-southeast-2	1.25
ds2.xlarge	us-west-1	1.25
ds2.xlarge	ap-northeast-1	1.19
ds2.xlarge	ap-northeast-3	1.19
ds2.xlarge	ap-south-1	1.19
ds2.xlarge	ap-northeast-2	1.15
ds2.xlarge	af-south-1	1.1305
ds2.xlarge	me-south-1	1.05
ds2.xlarge	eu-central-1	1.026
ds2.xlarge	us-gov-east-1	1.02
ds2.xlarge	us-gov-west-1	1.02
ds2.xlarge	eu-west-2	1
ds2.xlarge	eu-west-3	1

node type	region	price
ds2.xlarge	eu-south-1	0.9975
ds2.xlarge	eu-west-1	0.95
ds2.xlarge	ca-central-1	0.935
ds2.xlarge	eu-north-1	0.9025
ds2.xlarge	us-east-1	0.85
ds2.xlarge	us-east-2	0.85
ds2.xlarge	us-west-2	0.85
ds2.xlarge	ap-southeast-3	no data
ds2.xlarge	cn-north-1	no data
ds2.xlarge	cn-northwest-1	no data
ds2.xlarge	us-gov-secret-1	no data
ds2.xlarge	us-gov-topsecret-1	no data
ds2.xlarge	us-gov-topsecret-2	no data

Table 28: ra3.xlplus

node type	region	price
ra3.xlplus	sa-east-1	1.731
ra3.xlplus	ap-southeast-1	1.303
ra3.xlplus	ap-southeast-2	1.303
ra3.xlplus	eu-central-1	1.298
ra3.xlplus	ap-northeast-1	1.278
ra3.xlplus	ap-northeast-2	1.278
ra3.xlplus	eu-west-3	1.265
ra3.xlplus	eu-west-2	1.264
ra3.xlplus	ap-south-1	1.235
ra3.xlplus	ca-central-1	1.202
ra3.xlplus	eu-west-1	1.202
ra3.xlplus	us-west-1	1.202
ra3.xlplus	eu-north-1	1.139
ra3.xlplus	ap-east-1	1.086
ra3.xlplus	us-east-1	1.086
ra3.xlplus	us-east-2	1.086
ra3.xlplus	us-gov-east-1	1.086
ra3.xlplus	us-gov-west-1	1.086
ra3.xlplus	us-west-2	1.086
ra3.xlplus	af-south-1	no data
ra3.xlplus	ap-northeast-3	no data
ra3.xlplus	ap-southeast-3	no data
ra3.xlplus	cn-north-1	no data
ra3.xlplus	cn-northwest-1	no data
ra3.xlplus	eu-south-1	no data
ra3.xlplus	me-south-1	no data
ra3.xlplus	us-gov-secret-1	no data
ra3.xlplus	us-gov-topsecret-1	no data
ra3.xlplus	us-gov-topsecret-2	no data

Table 29: dc2.8xlarge

node type	region	price
dc2.8xlarge	sa-east-1	7.68
dc2.8xlarge	ap-east-1	7.04
dc2.8xlarge	af-south-1	6.664
dc2.8xlarge	ap-southeast-1	6.4
dc2.8xlarge	ap-southeast-2	6.4
dc2.8xlarge	ap-southeast-3	6.4
dc2.8xlarge	us-west-1	6.4
dc2.8xlarge	me-south-1	6.16
dc2.8xlarge	ap-south-1	6.1
dc2.8xlarge	ap-northeast-1	6.095
dc2.8xlarge	ap-northeast-3	6.095
dc2.8xlarge	eu-central-1	6.048
dc2.8xlarge	eu-south-1	5.88
dc2.8xlarge	eu-west-2	5.88
dc2.8xlarge	eu-west-3	5.88
dc2.8xlarge	ap-northeast-2	5.8
dc2.8xlarge	us-gov-east-1	5.76
dc2.8xlarge	us-gov-west-1	5.76
dc2.8xlarge	eu-west-1	5.6
dc2.8xlarge	eu-north-1	5.32
dc2.8xlarge	ca-central-1	5.28
dc2.8xlarge	us-east-1	4.8
dc2.8xlarge	us-east-2	4.8
dc2.8xlarge	us-west-2	4.8
dc2.8xlarge	cn-north-1	no data
dc2.8xlarge	cn-northwest-1	no data
dc2.8xlarge	us-gov-secret-1	no data
dc2.8xlarge	us-gov-topsecret-1	no data
dc2.8xlarge	us-gov-topsecret-2	no data

Table 30: ds2.8xlarge

node type	region	price
ds2.8xlarge	ap-east-1	11
ds2.8xlarge	sa-east-1	10.88
ds2.8xlarge	ap-southeast-1	10
ds2.8xlarge	ap-southeast-2	10
ds2.8xlarge	us-west-1	10
ds2.8xlarge	ap-northeast-1	9.52
ds2.8xlarge	ap-northeast-3	9.52
ds2.8xlarge	ap-south-1	9.5
ds2.8xlarge	ap-northeast-2	9.05
ds2.8xlarge	af-south-1	9.044
ds2.8xlarge	me-south-1	8.36

node type	region	price
ds2.8xlarge	eu-central-1	8.208
ds2.8xlarge	us-gov-east-1	8.16
ds2.8xlarge	us-gov-west-1	8.16
ds2.8xlarge	eu-south-1	7.98
ds2.8xlarge	eu-west-2	7.98
ds2.8xlarge	eu-west-3	7.98
ds2.8xlarge	eu-west-1	7.6
ds2.8xlarge	ca-central-1	7.48
ds2.8xlarge	eu-north-1	7.22
ds2.8xlarge	us-east-1	6.8
ds2.8xlarge	us-east-2	6.8
ds2.8xlarge	us-west-2	6.8
ds2.8xlarge	ap-southeast-3	no data
ds2.8xlarge	cn-north-1	no data
ds2.8xlarge	cn-northwest-1	no data
ds2.8xlarge	us-gov-secret-1	no data
ds2.8xlarge	us-gov-topsecret-1	no data
ds2.8xlarge	us-gov-topsecret-2	no data

Table 31: ra3.4xlarge

node type	region	price
ra3.4xlarge	sa-east-1	5.195
ra3.4xlarge	ap-southeast-1	3.909
ra3.4xlarge	ap-southeast-2	3.909
ra3.4xlarge	ap-southeast-3	3.909
ra3.4xlarge	eu-central-1	3.894
ra3.4xlarge	ap-northeast-1	3.836
ra3.4xlarge	ap-northeast-2	3.836
ra3.4xlarge	eu-west-3	3.795
ra3.4xlarge	eu-west-2	3.793
ra3.4xlarge	ap-south-1	3.706
ra3.4xlarge	ca-central-1	3.607
ra3.4xlarge	eu-west-1	3.606
ra3.4xlarge	us-west-1	3.606
ra3.4xlarge	eu-north-1	3.418
ra3.4xlarge	ap-east-1	3.26
ra3.4xlarge	us-east-1	3.26
ra3.4xlarge	us-east-2	3.26
ra3.4xlarge	us-gov-east-1	3.26
ra3.4xlarge	us-gov-west-1	3.26
ra3.4xlarge	us-west-2	3.26
ra3.4xlarge	af-south-1	no data
ra3.4xlarge	ap-northeast-3	no data
ra3.4xlarge	cn-north-1	no data
ra3.4xlarge	cn-northwest-1	no data

node type	region	price
ra3.4xlarge	eu-south-1	no data
ra3.4xlarge	me-south-1	no data
ra3.4xlarge	us-gov-secret-1	no data
ra3.4xlarge	us-gov-topsecret-1	no data
ra3.4xlarge	us-gov-topsecret-2	no data

Table 32: ra3.16xlarge

node type	region	price
ra3.16xlarge	sa-east-1	20.78
ra3.16xlarge	ap-southeast-1	15.636
ra3.16xlarge	ap-southeast-2	15.636
ra3.16xlarge	eu-central-1	15.578
ra3.16xlarge	ap-southeast-3	15.363
ra3.16xlarge	ap-northeast-1	15.347
ra3.16xlarge	ap-northeast-2	15.347
ra3.16xlarge	eu-west-3	15.18
ra3.16xlarge	eu-west-2	15.174
ra3.16xlarge	ap-south-1	14.827
ra3.16xlarge	ca-central-1	14.43
ra3.16xlarge	eu-west-1	14.424
ra3.16xlarge	us-west-1	14.424
ra3.16xlarge	eu-north-1	13.675
ra3.16xlarge	ap-east-1	13.04
ra3.16xlarge	us-east-1	13.04
ra3.16xlarge	us-east-2	13.04
ra3.16xlarge	us-gov-east-1	13.04
ra3.16xlarge	us-gov-west-1	13.04
ra3.16xlarge	us-west-2	13.04
ra3.16xlarge	af-south-1	no data
ra3.16xlarge	ap-northeast-3	no data
ra3.16xlarge	cn-north-1	no data
ra3.16xlarge	cn-northwest-1	no data
ra3.16xlarge	eu-south-1	no data
ra3.16xlarge	me-south-1	no data
ra3.16xlarge	us-gov-secret-1	no data
ra3.16xlarge	us-gov-topsecret-1	no data
ra3.16xlarge	us-gov-topsecret-2	no data

Conclusions

Within the resolving power of the benchmark suite, Redshift node types are identical in performance across regions, with the single exception of the `dc2.large` node type, which is much slower in five regions; `af-south-1`, `ap-east-1`, `ap-southeast-3`, `eu-south-1`, and `me-south-1`.

Unexpected Findings

When you investigate Redshift, there are *always* unexpected findings.

1. The step-plan for the disk (read) benchmark reveals a missing step, the step-plan for the network benchmark, two. It's not clear why this is happening, but on the face of it, I can no longer fully know what work a query has done. This would be staggeringly bad; when a cluster has problems, and I need to figure out what's happening with a slow query, how can I do so when the slow step might no longer be present in the system tables?

2. A benchmark run on the three small node types over the twenty-two regions being tested fires up about 60 clusters (not all node types are available in all regions), which are two worker nodes each, and so about 180 nodes (if we include the leader node). About a hundred of so queries would then be issued each cluster. I would normally find, with this number of clusters and this number of queries, that one query, to one cluster, would hang indefinitely; this happens because a Redshift internal process (usually the WLM manager) would crash, but this crash would not cancel the query or disconnect the user, and so from the users point of view, it looks just like a normal query which is taking a long time to run. For me, using `pyscopg2`, I would call `execute`, and it would simply not return; no error occurred.

   To my thought, this then requires of the Redshift user two things, firstly, that timeouts need to be set on queries, which is a bit problematic, because you don't know how long a query will take to execute, and secondly, of course, you must structure your code so any single query hanging only affects the results from that query, and that you can easily re-run that query. For me, the original benchmark code was structured that all benchmarks were run, on all nodes on all regions, and then the results returned; this required *all* clusters to function correctly. Once I recognized this problem, the code was structured so that each cluster returned its results independently of the others, and so when a cluster hung in this way, I had the results from all the other clusters, and then re-ran the benchmark on that single cluster.

Revision History

v1

- Initial release.

v2

- No content changes; path to image file(s) changed.

v3

- Added "About the Author". made site name in title a link, and made each chapter start a new page.
- Updated links to amazonredshiftresearcproject.org to redshiftresearcproject.org.

Appendix A : Raw Data Dump

The document processing software is having trouble with this appendix, as it is about 700kb of JSON.

As such, I've had to here give a link to the appendix as a separate document, which is here.

About the Author

I am a C programmer - kernel development, high performance computing, networking, data structures and so on.

I read the C. J. Date book, the classic text on relational database theory, and having learned the principles, wrote a relational database from scratch in C, which purely by chance set me up quite nicely for what came next, moving into data engineering in late 2011, when I joined as the back-end engineer two friends in their startup.

In that startup, I began using Redshift the day it came out, in 2012 (we had been trying to get into the beta programme).

We were early, heavy users for a year and a half, and I ending up having monthly one-to-one meetings with one of the Redshift team managers, where one or two features which are in Redshift today originate from suggestions made in those meetings, such as the distribution style `ALL`.

Once that was done, after a couple of years of non-Redshift data engineering work, I returned to Redshift work, and then in about mid-2018 contracted with a publisher to write a book about Redshift.

The book was largely written but it became apparent I wanted to do a lot of things which couldn't be done with a book - republish on every new Redshift release, for example - and so in the end I stepped back from the contract and developed the web-site, where I publish investigation into, and ongoing monitoring of, Redshift.

So for many years now I've been investigating Redshift sub-systems full-time, one by one, and this site and these investigations are as far as I know the and the only source of this kind of information about Redshift.

Redshift Cluster Cost Reduction Service

I provide consultancy services for Redshift - advice, design, training, getting failing systems back on their feet pronto, the usual gamut - but in particular offer a Redshift cluster cost reduction service, where the fee is and only is one month of the savings made.

Broadly speaking, to give guidance, savings are expected fall into one of two categories; either something like 20%, or something like 80%. The former is

for systems where the business use case is such that Redshift cannot be operated correctly, and this outcome requires no fundamental re-engineering work, the latter is for systems where Redshift can be operated correctly, and usually requires fundamental re-engineering work (which you may or may not wish to engage in, despite the cost savings, in which case we're back to the 20%).

Details and contact information are on the web-site.