

# Pseudo-Random Number Generation

Max Ganz II @ [Redshift Observatory](#)

18th October 2021

### **Abstract**

Redshift provides two PRNGs, one on the leader node and one on the worker nodes. The leader node PRNG is correctly implemented. The worker node PRNG is fundamentally flawed. Each worker slice produces a linear, minutely incrementing number sequence (the “non-PRNG number sequence”) which cycles between 0.0 and 1.0, where when a query is issued, each slice consumes the current number from that sequence and uses it as the seed for a PRNG, which in turn uses it to generate a random number sequence for and only for that query. The first random number emitted by a query on each slice is the number from the non-PRNG number sequence on that slice. The following numbers are from the PRNG. Low-entropy non-PRNG number sequence numbers, occurring when the numbers contain many zero bits, lead to correlation between the initial numbers produced by a PRNG for a query, and also between the initial values produced between queries.

# Contents

<b>Introduction</b>	<b>3</b>
<b>Redshift Internal Architecture</b>	<b>4</b>
<b>Test Method</b>	<b>7</b>
<b>Results</b>	<b>10</b>
dc2.large, 2 nodes (1.0.30840) . . . . .	10
Five Queries, One Row Each, One Number per Row, Leader Node	10
One Query, Five Rows, One Number per Row, Leader Node . . .	10
One Query, One Row, Five Numbers per Row, Leader Node . . .	11
PRNG Seed/Number Sequence per-Session or Global, Leader Node	11
First Five Numbers per Worker Slice . . . . .	11
First Five Numbers per Worker Slice (with extra query) . . . . .	11
One Query, Five Rows, One Number per Row, Worker Slice 0 . .	11
Five Queries, One Row per Query, Five Numbers per Row,	
Worker Slice 0 . . . . .	12
One Query, One Row, One Number per Slice, Worker Nodes . . .	12
PRNG Seed/Number Sequence per-Session or Global, Worker Node	12
Slice Hopping and PRNG Consumption . . . . .	12
<b>Discussion</b>	<b>14</b>
<b>Conclusions</b>	<b>23</b>
<b>Unexpected Findings</b>	<b>25</b>
<b>Revision History</b>	<b>26</b>
v1 . . . . .	26
v2 . . . . .	26
v3 . . . . .	26
v4 . . . . .	26
v5 . . . . .	26
v6 . . . . .	26
v7 . . . . .	27
v8 . . . . .	27
<b>Appendix A : Raw Data Dump</b>	<b>28</b>

<b>Appendix B : The Curious Step-Plan</b>	<b>31</b>
<b>Appendix C : Dieharder Results</b>	<b>34</b>
Core Code . . . . .	34
Python 3.7.3 . . . . .	35
Redshift Leader Node (dc2.large, 2 nodes, 1.0.30840) . . . . .	38
Redshift Worker Slice 0 (dc2.large, 2 nodes, 1.0.30840) . . . . .	40
Summary . . . . .	43
<b>Redshift Observatory Slack</b>	<b>44</b>

# Introduction

This paper is an investigation into the pseudo-random number generator (PRNG) in Redshift, and credit here must be given to another software engineer.

I began to investigate this subject in early 2021, but was immediately directed by a fellow Redshift admin to a [post](#) on the official Amazon Redshift Developer Forums, made by Murat Tasan ([LinkedIn](#)), which describes major design flaws in the PRNG.

The origin of the investigation in this paper is his findings, which I have investigated, determined the causes, formalized with a script to provide reproducibility, and then extended by investigating further aspects of behaviour in different situations.

In general, then, this white paper explores the behaviour of the pseudo-random number generators (there's more than one, as you will see) in Redshift.

# Redshift Internal Architecture

This white paper intended to explore random number generation, then in doing so ran into what I think is a change made something like a year ago in the internal architecture of Redshift, and which I've been observing in various places since then but have yet to actually investigate.

Note though this is based on my recollections of behaviour prior to this time, as I have no earlier investigations into this matter, so I can't compare what happens now with a formal record of what used to happen - so I could be wrong.

Nevertheless, the crux of this change, as I understand it, must be described in this white paper, and at this point, for without it the white paper will make no sense at all - even the test methods cannot be explained - even though here in this paper I am not providing proofs.

So, turning to the subject in hand : when it comes to Redshift clusters, it is never possible to run older versions. You can only run what's available now (and maybe one previous release). So I can always exactly determine what Redshift does *now*, but for anything in the past where I lack formal proofs, I'm relying on memory.

I *think* I remember things were different - but having no formal records of this behaviour, I can't be certain, so there is a possibility here of error on my part.

So, having said that, what I think is this : that it used to be each slice held a portion of the rows in each normal Redshift table, and that only the slice holding the rows could directly access those rows. For other slices to access those rows, the owning slice would need to read them, and then transmit them over the network, as part of the actual work being performed by a query (a broadcast or distribute step).

As such, if a block was stored on a slice and you accessed that block, it would necessarily be that slice which read that block.

Well, to quote Mr. Dylan, "[things have changed](#)".

Now, I've not actually sat down and investigated what's going on, so what I relate now are the behaviours I've noticed as I've been doing other work and especially with the investigation work for this white paper.

The key change is that it looks now that every slice on a node can directly access the rows of every other slice on that node.

As such, when a query is issued, say for example a query which uses rows only held by one slice, it is no longer possible to know which slice that query will execute upon; it could execute on any of the slices in that node.

If only one slice on a node holds blocks for a table, there seems to be a strong preference for a query to execute on that slice. If all slices on a node hold blocks, it seems query distribution is pretty evenly spread.

Most queries of course execute on all slices, and in that, the usual case, it won't matter (assuming that slices are wholly symmetric in performance), but here in this white paper for example, with random number generation investigation, I need to ensure queries are running on the same single slice, always. Before, I could do this, by controlling which slice blocks went to and then issuing queries which used blocks on a single slice only. Now I cannot - what I now do in fact is loop over each query *until it executes on the slice I want it to execute on*. As with certain other changes in recent years (auto-vacuum, system tables becoming views and the underlying tables no longer being accessible, etc), this obstructs investigation.

I suspect this change relate to another change, which occurred between between 2019-09-27 and 2020-10-03 (I have partial dumps of the system tables' DDL and SQL on those dates). We can observe between these times a new column was added to STV\_SLICES, `type`, which is a `char(1)` where the value is either `C` or `D`, which we reasonably can assume mean 'compute' and 'data', respectively.

I've only ever been able to get compute slices to show up on `ra3` type nodes, and they are brought into existence by an elastic resize.

(Brief refresher on elastic resize : the number of slices remains the same, but the number of nodes changes, and the slices are redistributed over the nodes. So if you add nodes, you end up with more hardware, but less slices per node, and so reduced chances for efficiency by exploiting parallelism. If you remove nodes, you have less hardware, and so save money, but now have more slices on each node, and if you do that too much, the overheads involved in each slice begin to be unduly costly. Modus omnibus in rebus.)

Here's STV\_SLICES from a brand new two node `ra3.xlplus`;

node	slice	localslice	type
0	0	0	D
0	1	1	D
1	2	0	D
1	3	1	D

And this is what we see after an elastic re-size to four nodes.

node	slice	localslice	type
0	0	0	D
0	4	1	C
1	2	0	D

1	5	1   C
2	1	0   D
2	6	1   C
3	3	0   D
3	7	1   C

The number of nodes has doubled, the data slices have been distributed between them, and the other slice on each node is a compute slice.

Compare this to the same re-size for a two node `dc2.large`;

node	slice	localslice	type
0	0	0   D	
0	1	1   D	
1	2	0   D	
1	3	1   D	

And after a re-size to four nodes;

node	slice	localslice	type
0	0	0   D	
1	2	0   D	
2	1	0   D	
3	3	0   D	

Here we see the number of slices remains the same.

This new behaviour is immediately interesting in one way, in that I think it solves, for `ra3` nodes, a drawback (as ever, undocumented) with elastic re-size, which is that you get a maximum of ten Redshift Spectrum workers per slice, so if you performed an elastic re-size, you'd be paying for more nodes but you wouldn't be getting more throughput with Redshift Spectrum, because the number of slices was fixed. Now we see elastic re-size actually giving more slices, at least on `ra3` - it doesn't on the other node types, not as far as my brief investigation into re-sizing has found.

Speculatively, I would guess the compute slices have no SSD storage of their own. They're on the same node, though, so it doesn't matter now; they can directly access to the blocks held by other slices.

One obvious question is if there are performance issues.

My gut feeling is that only one slice at a time can access the blocks held by a slice. This is something in particular I will look into when I investigate all this new behaviour.

So, finally coming back now to this white paper, where random number generation is investigated, this is why there are in the Python script used to generate evidence loops around queries, which check to see the slice the query executed upon, to ensure results are only from queries which executed on a given slice.



# Test Method

There are basically two types of test in this white paper.

The first type of test produces only a few numbers from the PRNG, and presents them in a table, along with an explanation of what they must mean.

The second type produces an *image*, a small grey-scale bitmap, 256 by 256 pixels in size, by converting each random number to a grey-scale pixel; Redshift emits random numbers between 0.0 and 1.0, where 0.0 becomes black and 1.0 becomes white.

A reference image is generated using the Python 3.7.3 PRNG, which I understand is a [Mersenne Twister](#), which is about the best non-cryptographic PRNG out there, and very nice it is too.

The use of an image is a rough and ready method to check if a PRNG is emitting random numbers; if it is not, patterns can be seen in the bitmap. *Any* patterning visible to the plain human eye means the PRNG is emitting garbage.

To properly test a random number generator, pseudo or real, a large battery of statistical tests are performed on its output, and one utility for this is a command line tool known as **dieharder**.

This is available on my Debian system from the Debian repository, but it's not necessarily available on other systems, and so although I have included in [Appendix C](#) its results on the Python 3.7.3 PRNG, the leader-node PRNG and a worker node slice PRNG (the leader node has a different PRNG to the worker nodes), I have not used it generally, since the script which produces evidence for each white paper is specifically intended to be used by readers, so they can verify the results are true for their system, and to check in the future if the results are still true or if Redshift has changed.

Turning directly now to the tests, note all tests on a worker node are run, unless specifically indicated otherwise, on node 0 slice 0. All tests check that their queries actually did run on that slice, and run them again (resetting the seed, of course), if they did not.

All tests are usual are issued on a newly spun-up cluster. For my run of the script, I used a two node **dc2.large** cluster.

The tests are;

1. Five queries are issued, which produce one row each, with one random number per row, on the leader node.

2. One query is issued, which produces five rows, with one random number per row, on the leader node.
3. One query is issued, which produces one row, with five random numbers, on the leader node.
4. We now check if the PRNG seed/number sequence, on the leader node, is per-session or across all sessions. This is done by opening two connections to the cluster and setting the seed on both to 0, then issuing one query on the first connection taking one row of three random numbers from the leader node, and then the same query, but on the second connection.

If the three random numbers are the same for both queries, the seed must be per-session.

5. For information purposes, this test generates the first five random numbers from each worker slice.
6. We then repeat the previous test, but now with a superfluous query, which does not generate random numbers, between every query.
7. One query is issued, which generates five rows, with one random number per row, on worker slice 0.
8. Five queries are issued, generating one row each, with five random numbers per row, on worker slice 0.
9. One query is issued, with one row, which has one random number per slice, with the query generating one row on each slice.
10. We now check if the PRNG seed/number sequence, on the worker node, is per-session or across all sessions. This is done by opening two connections to the cluster and setting the seed on both to 0, then issuing one query on the first connection taking one row of one random number from the leader node, and then the same query, but on the second connection.

If the random numbers are the same for both queries, the seed must be per-session.

11. This test generates a bitmap of the results from a single query, generating all rows, where each row has a single random number, on the leader node.
12. This test generates a bitmap of the results from a single query, generating all rows, where each row has a single random number, on worker slice 0.
13. This test generates a bitmap of the results from a  $2^{16}$  queries, generating one row each, where each row has a single random number, on the leader node.
14. This test generates a bitmap of the results from a  $2^{16}$  queries, generating one row each, where each row has a single random number, on worker slice 0.
15. This test generates a bitmap of the results from a  $2^{16}$  queries, generating one row each, where each row generates five random numbers, on worker slice 0.

16. This test generates a bitmap of the results from a  $(2^{16})5$  queries, generating one row each, where each row generates five random numbers, on worker slice 0; the bitmap is generated from the second\* random number from each query. This demonstrates cross-query correlation.
17. This test generates the first five random numbers from worker slice 0, and then generates the same random numbers again except that this second set is generated repeatedly until exactly one of the numbers, between the second and second-to-last, slice hops. The two sets of results demonstrate what happens to the number sequence when slice hopping occurs.

# Results

The results are given here for ease of reference, but they are also presented, piece by piece along with explanation, in the Discussion, and it is there you should read about them to understand them.

See [Appendix A](#) for the Python `pprint` dump of the results dictionary.

Download [random\\_number\\_generation.py](#), the script used to generate the evidence for this white paper.

Usage : `./pseudo_random_number_generation.py one-shot`

The script works in `eu-central-1`, creates a VPC, configures everything that needs, runs the cluster specified in the test, executes the tests, and dumps the results on screen, then cleans everything up.

The script, like all ARRP scripts, requires `boto3` version 1.17.34 or later.

Note this script, where it generates images, makes a directory named `images`, which it populates, and requires `numpy` to be installed; I have no idea which version introduced the image generation functionality I'm using. I should think anything non-archaic will be fine.

Total script execution time, including cluster bring-up and shut-down, was 4,792 seconds.

## dc2.large, 2 nodes (1.0.30840)

**Five Queries, One Row Each, One Number per Row, Leader Node**

Query	Number
1	0.840187716763467
2	0.394382926635444
3	0.783099223393947
4	0.798440033104271
5	0.911647357512265

**One Query, Five Rows, One Number per Row, Leader Node**

Row	Number
1	0.840187716763467
2	0.394382926635444
3	0.783099223393947
4	0.798440033104271
5	0.911647357512265

### One Query, One Row, Five Numbers per Row, Leader Node

Number #1	Number #2	Number #3	Number #4	Number #5
0.840188	0.394383	0.783099	0.798440	0.911647

### PRNG Seed/Number Sequence per-Session or Global, Leader Node

Session	Query	random() #1	random() #2	random() #3
1	1	0.840187716763467	0.394382926635444	0.783099223393947
2	1	0.840187716763467	0.394382926635444	0.783099223393947

### First Five Numbers per Worker Slice

Query	Slice #0	Slice #1	Slice #2	Slice #3
1	0.0000000000000000	0.3366925144291638	0.3333850288583238	0.3100775432874837
2	0.0000895813340953	0.3670147256256947	0.33939869917293	0.100865014208892
3	0.0001791626681536	0.3671043069597497	0.34029451251349	0.100954595542948
4	0.0002687440022036	0.3671938882938057	0.34119032585404	0.101044176877004
5	0.0003583253362636	0.3672834696278617	0.34208613919460	0.101133758211059

### First Five Numbers per Worker Slice (with extra query)

Query	Slice #0	Slice #1	Slice #2	Slice #3
1	0.0000000000000000	0.3366925144291638	0.3333850288583238	0.3100775432874837
2	0.0001791626681536	0.3671043069597497	0.34029451251349	0.100954595542948
3	0.0003583253362636	0.3672834696278617	0.34208613919460	0.101133758211059
4	0.0005374880043723	0.3674626322959727	0.34387776587571	0.101312920879170
5	0.0007166506724843	0.3676417949640837	0.34566939255682	0.101492083547281

### One Query, Five Rows, One Number per Row, Worker Slice 0

Row	Number
1	0.0000000000000039
2	0.000985394674650
3	0.041631001594613
4	0.176642642542916
5	0.364602248390607

### Five Queries, One Row per Query, Five Numbers per Row, Worker Slice 0

Number #1	Number #2	Number #3	Number #4	Number #5
0.0000000000000039	0.000985394674650	0.041631001594613	0.176642642542916	0.364602248390607
0.000089581334095	0.731953177121920	0.872086605317186	0.375548061256342	0.794304826910761
0.000179162668150	0.462920959569189	0.702542209039759	0.574453479969769	0.224007405430914
0.000268744002206	0.193888742016458	0.532997812762332	0.773358898683195	0.653709983951067
0.000358325336261	0.924856524463728	0.363453416484905	0.972264317396622	0.083412562471221

### One Query, One Row, One Number per Slice, Worker Nodes

Slice #0	Slice #1	Slice #2	Slice #3
0.0000000000000039	0.366925144291638	0.733850288583238	0.100775432874837

### PRNG Seed/Number Sequence per-Session or Global, Worker Node

Session	Query	random() #1
1	1	0.0000000000000039
2	1	0.000089581334095

### Slice Hopping and PRNG Consumption

Pinned to Slice 0

Query	Slice	random()
1	0	0.0000000000000039
2	0	0.000089581334095
3	0	0.000179162668150
4	0	0.000268744002206
5	0	0.000358325336261

### Unpinned to Slice 0

Query	Slice	random()
1	0	0.0000000000000039
2	1	0.367014725625694
3	0	0.000179162668150
4	0	0.000268744002206
5	0	0.000358325336261

# Discussion

We begin with the leader node. Remember that between every test, seed is set back to zero.

First, we issue five queries, each producing one row, with one random number.

Query	Number
1	0.840187716763467
2	0.394382926635444
3	0.783099223393947
4	0.798440033104271
5	0.911647357512265

These random numbers are then the first five numbers in the leader node PRNG sequence.

Next, we issue one query, which produces five rows, each with one random number.

Row	Number
1	0.840187716763467
2	0.394382926635444
3	0.783099223393947
4	0.798440033104271
5	0.911647357512265

Here we see the same five numbers.

We then issue one query, which produces one row, but that row has five columns, each a random number.

Value #1	Value #2	Value #3	Value #4	Value #5
0.840188	0.394383	0.783099	0.798440	0.911647

Here again we see the same five numbers.

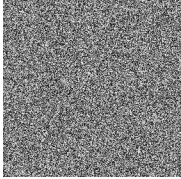
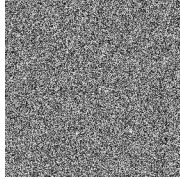


Finally, we check to see if the PRNG seed/number sequence is per-session, or global across all sessions.

To do this, we open two sessions, set the seed to 0 in both, and issue in both sessions one query which produces one row of three random numbers.

Session #	Query #	random() #1	random() #2	random() #3
1	1	0.840187716763467	0.394382926635444	0.783099223393947
2	1	0.840187716763467	0.394382926635444	0.783099223393947

If the numbers produced from both queries are identical, then the PRNG seed/number sequence must be on a per-session basis, and this is what found.

Python	Leader
	

Here we see the reference bitmap from the Python 3.7.3 PRNG, and the bitmap produced by the leader node PRNG with one query producing all rows, with each row holding one random number. All the variations on generating random numbers (many queries with one row each with each row holding one number, many queries with many rows each with each row holding one number, etc) all produce similar, properly random output.

What we see then is that the leader node PRNG has a single source of random numbers, and every call to `random()` consumes one number from this sequence, regardless of whether the calls are made over many queries, or many rows per query, or many calls per row. We also see the seed/position in the number sequence is held on a per-session basis.

One of the properties of a PRNG is *reproducibility*, which is to say, that the sequence of numbers produced is always the same, so that once we have picked a starting point in that sequence by setting the seed, we will always get the same sequence of numbers. This is necessary for debugging.

The leader node provides reproducibility.

All in all what we see from the leader node is what we would expect from the leader node, as it was formerly a single-node relational database, and when you only have a single node to deal with, life is much simpler.

Now we turn to worker nodes.

To begin with, we issue *for each slice*, five queries, each producing one row, each row holding one random number. This gives us the first five numbers in the PRNG number sequence for each slice.

Query	Slice #0	Slice #1	Slice #2	Slice #3
1	0.000000000000039	0.366925144291638	0.733850288583238	0.100775432874837
2	0.000089581334095	0.367014725625694	0.733939869917293	0.100865014208892
3	0.000179162668150	0.367104306959749	0.734029451251349	0.100954595542948
4	0.000268744002206	0.367193888293805	0.734119032585404	0.101044176877004
5	0.000358325336261	0.367283469627861	0.734208613919460	0.101133758211059

What we observe is that each slice, for seed 0, starts at a different location in the number sequence, and that each number being produced is in all cases an increase of 0.000089581334055 on the previous number. When the number exceeds 1.0, it loops back to 0.0.

This is not in fact a PRNG, being from here on referred to as the “non-PRNG number sequence”, and so we must now investigate further to find how random number are being generated.

First, if we repeat these queries, but this time issuing after each query a simple `select timeofday() from table 1 limit 1`, we see that the numbers produced *change*.

Query	Slice #0	Slice #1	Slice #2	Slice #3
1	0.000000000000039	0.366925144291638	0.733850288583238	0.100775432874837
2	0.000179162668150	0.367104306959749	0.734029451251349	0.100954595542948
3	0.000358325336261	0.367283469627861	0.734208613919460	0.101133758211059
4	0.000537488004372	0.367462632295972	0.734387776587571	0.101312920879170
5	0.000716650672484	0.367641794964083	0.734566939255682	0.101492083547281

In fact, what’s happening is the extra query is causing every slice to jump forward by one number in its number sequence, even though the extra query does not generate any random numbers. The second set of results are the same as the first, but with the lines from queries #2 and #4 removed (and the later rows in the second set are not seen in the first set, as both sets only have five rows).

We will come back to this a little later.

Next, if we issue a single query, to worker slice 0, which produces five rows, each with one random number, we see the first number is unchanged, but the succeeding numbers are different to those generated when five individual queries are issued. We are now seeing something new, and which looks more like PRNG output.

Row	Number
1	0.000000000000039
2	0.000985394674650
3	0.041631001594613
4	0.176642642542916
5	0.364602248390607

This behaviour becomes clearer when we issue five queries, all to slice 0, with seed set to 0 once, at the beginning of the set of five queries, with each query producing one row, where each row has five random numbers.

Number #1	Number #2	Number #3	Number #4	Number #5
0.000000000000039	0.000985394674650	0.041631001594613	0.176642642542916	0.364602248390607
0.000089581334095	0.731953177121920	0.872086605317186	0.375548061256342	0.794304826910761
0.000179162668150	0.462920959569189	0.702542209039759	0.574453479969769	0.224007405430914
0.000268744002206	0.193888742016458	0.532997812762332	0.773358898683195	0.653709983951067
0.000358325336261	0.924856524463728	0.363453416484905	0.972264317396622	0.083412562471221

First, we see the five random numbers from the first query match the five random numbers produced when we issue single queries which produces one row with one random number each (not the table immediately above with the five columns, but the table above that).

Second, if we look back at the table showing the first five random numbers produced on each slice, we see that the *first* number from each query (remembering the queries here all run on slice 0) is reproducing the first five random numbers from slice 0.

What seems then to be happening is that with the slice begins with the first number in its non-PRNG number sequence (0.000000000000039), and then it generates a sequence of random numbers (the next four random numbers in the row), using a PRNG, derived from that initial number.

Then, when we come to the second query, the slice moves on to the second number in its non-PRNG number sequence (0.000089581334095), and then, as before, the slice generates numbers from a PRNG number sequence derived from that initial number.

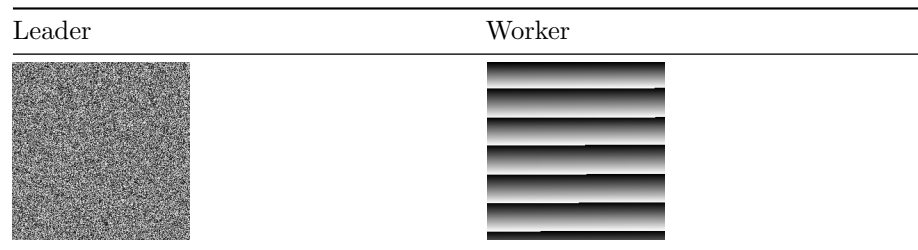
It seems then that every query - and if we think back to the earlier results where we saw queries which were not generating random numbers causing progression down the non-PRNG Number sequence - consumes the current non-PRNG number from the slice(s) it is running on and uses that as the seed for an actual PRNG, to generate the random numbers for that query.

Slice #0	Slice #1	Slice #2	Slice #3
0.000000000000039	0.366925144291638	0.733850288583238	0.100775432874837

This is what we see here, with a query which generates one row, with one random number per slice. Each slice emits the current number from its own non-PRNG number sequence.

This leads to the immediate observation that any user issuing single queries which produce a single random number is fully exposed to the minutely incrementing non-PRNG number sequence for whatever slice produces the row for that query, because that number is always given as the first random number for a query.

This is a fundamental design flaw; these are not random numbers and `random()` should provide random numbers no matter how you call it.



On the left, the bitmap produced by the leader node, when we generate pixels by issuing a single query which produces a single row with a single random number. On the right, worker slice 0.

Comparing here the leader node to the worker node is a little unfair, since the leader node is a single computer rather than a cluster and so faces simpler design challenges, but the worker node PRNG can perfectly well be designed so it works correctly, so it's not that unfair.

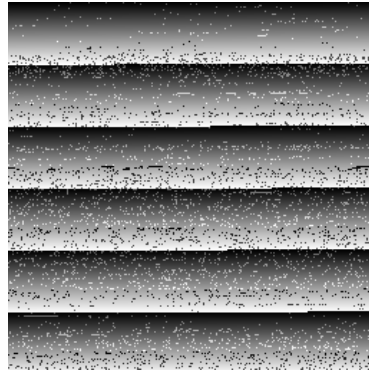


Figure 1: Unpinned Worker Slice 0

As an aside, figure 1 shows the bitmap produced when no steps are taken to use only numbers generated by slice 0. The speckles mark numbers which were produced on other slices, where their non-PRNG number sequence is different to that on slice 0. Normally I think queries run evenly over slices, but where these numbers are being produced by a select which is using a table which only has blocks on slice 0 (e.g. `select slice_num(), random() from table_1 limit 1`), there seems to be a strong preference for execution on slice 0.




Designs where a PRNG is seeded from another PRNG are not unknown, but the general advice is that the two PRNGs should use fundamentally different mathematical approaches. In this case, though, there is one non-PRNG number sequence and one PRNG, and this is not going to work correctly.

The problem is superficially ameliorated by each slice being at a different point of the non-PRNG number sequence, but this is a billion miles from being an actual, sound solution; in fact what it does mainly is obscure the problem, which is not what you want.

The reason I think of for a design like this is performance. The idea is that each query only needs to touch the master entropy state once, and then independently produces its own random numbers; if all queries were using the same entropy state to produce all their random numbers you could well end up with quite a bottleneck.

Turning to the secondary PRNG, examination on worker slice 0 demonstrates strong correlation between the initial few numbers created for each query, both simply between themselves but also between the numbers generated by successive queries, where by this I mean to say the first number generated by a query is correlated to the first number generated in the following queries, the second number generated is correlated to the second number generated in the following queries, and so on.

That's not random; random numbers do not correlate to other random numbers, *at all*, regardless of the query they're in.

Python	Single Query	Cross Query
		

Here we see a reference bitmap, generated from the Python 3.7.3 PRNG, then a bitmap produced by issuing many queries, which produce one row each, where that row has five random numbers (constructing the bitmap, left-to-right, then top-to-bottom, by concatenating each queries five random numbers), and finally, a bitmap produced by issuing many queries, which produce on row each, where that row has five random numbers, and this bitmap is composed of and only of the *second* random number from each query (and so here to produce a bitmap of the same size, five times as many queries had to be generated; note also the first number from these queries is the non-PRNG number sequence, and so is meaningless, which is why it's not used - we're here looking at the output from the worker slice PRNG, not the non-PRNG number sequence).

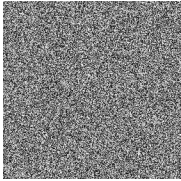
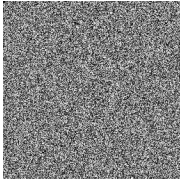
Remember; it may not seem like much, but anything you can actually notice with the plain human eye means output is profoundly broken.

I think the reason for both these problems is that in the test code, where I'm issuing test queries one after the other, the numbers taken from the non-PRNG number sequence are always very similar, and also I suspect being from slice 0, which begins its non-PRNG number sequence almost at zero, have very few bits set and so very little entropy, and it takes the PRNG some iterations to recover from this handicap and get to the point where it's producing decent random numbers.

However, we must note the non-PRNG number sequences loop between 0.0 and 1.0, so all slices will be for extended and contiguous periods be emitting low-entropy numbers.

Aside from this, we must remember that the first number for each query is always non-random, since it is taken from the minutely incrementing non-PRNG number sequence on each slice.

What we see then is that only safe way to use the worker node PRNG is to issue queries which produce enough random numbers to get past the initial non-random numbers. We see here the bitmap produced by worker slice 0 when a single query produces all rows, with one number per row. The first number is not random, as it is the non-PRNG number, then there's a series of low quality random numbers, and then the PRNG gets into its stride. The random numbers produced when worker slice 0 is used in this way perform well in the **dieharder** tests, the non-random initial numbers being overwhelmed by the many tens of thousands of random numbers which follow.

Python	Worker
	

Here we see the output from worker slice 0 when operated safely.

One thought then when using worker node PRNG is to issue prophylactic calls to `random()`, to generate the initial non-random or poorly random numbers and throw them away.

We can now turn back to two outstanding questions, whether the seed for worker nodes is per-session or global, and the question of reproducibility.

As before, the test for per-session seed is to make two connections, set seed to zero on both, and then issue a select into each session where the select generates a random number, although with the worker slices we obtain only one random number, rather than the three from the leader nodes (which I did just to show some extra numbers to make things more obvious), because as we've seen, with worker slices random numbers after the first come from the PRNG, which is using the first number as its seed, and so they are purely dependent on the first number and so having the first number is enough.

Session #	Query #	random() #1
1	1	0.0000000000000039
2	1	0.000089581334095

This test demonstrates that the non-PRNG seed/number sequence position is global, not per-session. This means every time *any* slice moves down the non-PRNG number sequence, *all* slices do so, throwing away the numbers they skip over.

Query	Slice	random()
1	0	0.0000000000000039
2	0	0.000089581334095
3	0	0.000179162668150
4	0	0.000268744002206
5	0	0.000358325336261

Query	Slice	random()
1	0	0.0000000000000039
2	1	0.367014725625694
3	0	0.000179162668150

Query	Slice	random()
4	0	0.000268744002206
5	0	0.000358325336261

This behaviour combines with slice hopping to produce a rather unexpected outcome. Each slice has its own non-PRNG number sequence, and normally queries which produce a single number will produce that sequence; we see this in the first of the two tables above. However, when a query hops to another slice, and runs there, that query will return a non-PRNG number sequence number from that other slice, *and* the slice the query normally runs on will move forward by one in its non-PRNG number sequence - permanently throwing away a number, even though we have not consumed it. We see this in the second of the two tables above.

Regarding reproducibility, it is obvious and inherent that Redshift worker nodes do not offer reproducibility.

This is because where Redshift runs on a cluster, reproducibility is in principle problematic. A query will execute, normally, on every slice in the cluster, and there is no telling in which order rows will be produced by the slices; it might be one slice is going slow, or another fast, depending on what other load is at the time present on those slices.

As such, even if the worker node PRNG were producing the same sequence of numbers across all slices, you would *still* see random numbers turning up in different orders in your rows every time you run a query, because the rows would naturally be generated in different orders every time the query runs.

Finally, note that Murat Tasan, the engineer who noticed the PRNG was behaving very oddly, did so back in December 2016.

That was five years ago. Redshift came out in early 2013, three years before then. My guess is the PRNG has been as it is found now since the beginning.



# Conclusions

There are two PRNGs in Redshift; one on the leader node, one on the worker nodes.

The leader node PRNG works correctly, always.

The worker node PRNG is fundamentally flawed.

The worker node PRNG design has each worker node slice producing a linear number series (the “non-PRNG number sequence”) where each number is 0.000089581334055 larger than the previous number, where each query, whether it produces random numbers or not, consumes the current number from the non-PRNG number sequence on each slice and uses this as the seed for a per-slice PRNG, which then generates random numbers for that query from the number sequence produced by that seed.

Each query then is generating its own, individual, per-slice sequence of PRNG numbers, where the PRNG is initialized by the number taken from the non-PRNG number sequence on each slice.

The first random number generated by a query, on each slice, is the number taken from the non-PRNG number sequence on that slice. These numbers change very slowly, incrementing each time a query is issued, and so this first number is usually similar or very similar to the previously produced first number; any user issuing consecutive queries which generate random numbers will receive a for their first number on each slice direct copy of the non-PRNG number sequence on each slice.

When queries which generate random numbers are issued in any kind of close succession, the numbers taken from the non-PRNG number sequences are very similar. Those sequences loop between 0.0 and 1.0, in tiny increments, and so for considerable and contiguous periods produce numbers with low entropy - many zero bits. When these two events combine, the PRNGs are seeded with numbers which are very similar and low-entropy, and this presents difficulties to the PRNG, which can take a certain number of iterations before it is generating high quality random numbers, and during this time the initial numbers emerging from the PRNG are low quality, such that these numbers are correlated both with themselves (the numbers being generated by a single query) and also with the numbers generated by successive queries.

It is advised then that when using `random()`, prophylactic calls are made, to generate and then throw away initial low quality random numbers. At the very

least the first random number from each slice must be disposed of, since it is from the non-PRNG number sequence.

The leader node maintains the PRNG seed and number sequence on a per-session basis, and provide reproducibility, always generating the same PRNG number sequence for any seed.

The worker nodes maintain the PRNG seed and number sequence globally, over all sessions, although the position of each slice in the number sequence is different. When one slice consumes a number from its non-PRNG number sequence, *all* slices move down their sequence by one number.

Where the slices in the worker nodes depending on load produce rows at different rates, it is inherently impossible for Redshift to provide reproducibility; even if the PRNG always emitted, across all slices, the same sequence of numbers, where the slices vary in how quickly they produce rows, the ordering of those numbers would always differ between runs of a query.

# Unexpected Findings

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

1. I seemed to sometimes run into problems dropping tables which had received large numbers (16,384) of sixteen row single inserts. There would be no queries running, but issuing a drop on the table leads to the drop just sitting there, nothing happening; I had to shut down the cluster. It might be if I waited a long enough time, the drop would work, but I never tried this.
2. If you issue a query in a procedure, Redshift will mess with the text you issue, upper-casing the letters of the SQL command and removing any trailing colon. If the query is executed via a cursor, i.e. `open cursor for ...`, a space is prepended to the query text, but the SQL command is now not capitalized, and in some cases any trailing semi-colon is also removed. Additionally, the text is messed with differently in `STL_QUERY` compared to `STL_QUERYTEXT`.
3. This may well be a standard PL/pgSQL behaviour, but I've not seen it documented anywhere and it's new to me : opening a cursor with a query does not cause query to execute. You have to call the first fetch to make the query occur.
4. In the low-level system tables, such as `STL_SCAN`, the row count number looks to be off-by-one some of the time. For example, if you scan five rows from `pg_class`, `STL_SCAN` reports you scanned six.

# Revision History

## **v1**

- Initial release.

## **v2**

- Re-arranged one or two sentences in the internal architecture material, where an old sentence from previous writing had inadvertently remained in the text.

## **v3**

- Fixed an error in the text where I instructed users to look at another table, but told them the wrong table - I said look up two tables, should have been one.

## **v4**

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

## **v5**

- Changed to Redshift Research Project (AWS have a copyright on “Amazon Redshift”).

## **v6**

- Added “About the Author”. made site name in title a link, and made each chapter start a new page.
- Updated links to [amazonredshiftresearchproject.org](http://amazonredshiftresearchproject.org) to [redshiftresearchproject.org](http://redshiftresearchproject.org).
- Reduced font size in tables where the numbers in the columns were overlapping.

## **v7**

- Web-site name changed to “Redshift Observatory”.
- Updated links from redshiftresearchproject.org to redshift-observatory.ch.

## **v8**

- Removed “About The Author”.
- Added Slack join URL.

# Appendix A : Raw Data Dump

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

```
{'proofs': {'dc2.large': {2: {'first_five_values_per_slice': {0: [3.90798504668055e-14,
8.95813340946461e-05,
0.000179162668150212,
0.000268744002205779,
0.000358325336261345],
1: [0.366925144291638,
0.367014725625694,
0.367104306959749,
0.367193888293805,
0.367283469627861],
2: [0.733850288583238,
0.733939869917293,
0.734029451251349,
0.734119032585404,
0.73420861391946],
3: [0.100775432874837,
0.100865014208892,
0.100954595542948,
0.101044176877004,
0.101133758211059]},
'first_five_values_per_slice_with_extra_query': {0: [3.90798504668055e-14,
0.000179162668150212,
0.000358325336261345,
0.000537488004372477,
0.00071665067248361],
1: [0.366925144291638,
0.367104306959749,
0.367283469627861,
0.367462632295972,
0.367641794964083],
2: [0.733850288583238,
0.734029451251349,
0.73420861391946,
0.734387776587571,
0.734566939255682],
3: [0.100775432874837,
0.100954595542948,
0.101133758211059,
0.10131292087917,
0.101492083547281]},
'five_queries__one_row__one_value_per_row__leader': [0.8401877167634666,
0.39438292663544416,
0.7830992233939469,
0.7984400331042707,
0.9116473575122654],
'five_queries__one_row_per_query__five_values_per_row__worker': [[0,
3.90798504668055e-14,
0.000985394674650308,
0.0416310015946131,
0.176642642542916,
0.364602248390607],
[0,
8.95813340946461e-05,
0.73195317712192,
0.872086605317186,
0.375548061256342,
0.794304826910761],
[0,
0.000179162668150212,
0.462920959569189,
0.702542209039759,
0.574453479969769,
0.224007405430914],
[0,
0.000268744002205779,
0.193888742016458,
```

```

0.532997812762332,
0.773358898683195,
0.653709983951067],
[0,
0.000358325336261345,
0.924856524463728,
0.363453416484905,
0.972264317396622,
0.0834125624712208]],
'many_queries__one_row_per_query__five_values_per_row__worker_node': [[3.90798504668055e-14,
0.000985394674650308,
0.0416310015946131,
0.176642642542916,
0.364602248390607],
[8.95813340946461e-05,
0.73195317712192,
0.872086605317186,
0.375648061266342,
0.794304826910761],
[0.000179162668150212,
0.462920959569189,
0.702542209039759,
0.574453479969769,
0.224007405430914],
[0.000268744002205779,
0.193888742016458,
0.532997812762332,
0.773358898683195,
0.653709983951067],
[0.000358325336261345,
0.924856524463728,
0.363453416484905,
0.972264317396622,
0.0834125624712208],
[0.000447906670316911,
0.655824306910997,
0.193909020207478,
0.171169736110048,
0.513115140991374],
[0.000537488004372477,
0.386792089358266,
0.024364623930051,
0.370075154823475,
0.942617719511527],
[0.000627069338428043,
0.117759871805536,
0.854820227652624,
0.568980573536901,
0.372520298031681],
[0.00071665067248361,
0.8487277654252805,
0.685275831375197,
0.767885992250328,
0.802222876551834],
[0.000806232006539176,
0.579685436700074,
0.51573143509777,
0.966791410963754,
0.231925455071988]],
'one_query__five_rows__one_value_per_row__leader': [[0.8401877167634666],
[0.39438292663544416],
[0.7830992233939469],
[0.7984400331042707],
[0.9116473575122654]],
'one_query__five_rows__one_value_per_row__worker': [[0,
3.90798504668055e-14],
[0,
0.000985394674650308],
[0,
0.0416310015946131],
[0,
0.176642642542916],
[0,
0.364602248390607]],
'one_query__one_row__five_values_per_row__leader': [[0.8401877167634666,
0.39438292663544416,
0.7830992233939469,
0.7984400331042707,
0.9116473575122654]],
'one_query__one_row__one_value_per_slice__worker': [[0,
3.90798504668055e-14],
[1,
0.366925144291638],
[2,
0.733850288583238],
[3,
0.100775432874837]],
'pinned_slice_0': [(0, 3.90798504668055e-14),
(0, 8.95813340946461e-05),
(0, 0.000179162668150212),
(0, 0.000268744002205779),
(0, 0.000358325336261345)],
'prng_session_or_global_leader_1': [[0.8401877167634666,
0.39438292663544416,
0.7830992233939469]],
'prng_session_or_global_leader_2': [[0.8401877167634666,
0.39438292663544416,
0.7830992233939469]],
'prng_session_or_global_worker_1': 3.90798504668055e-14,

```

```

'prng_session_or_global_worker_2': 8.95813340946461e-05,
'unpinned_slice_0': [(0, 3.90798504668055e-14),
(1, 0.367014725625694),
(0, 0.000179162668150212),
(0, 0.000268744002205779),
(0,
0.000358325336261345)]}],
'tests': {'dc2.large': {2: {}}},
'versions': {'dc2.large': {2: 'PostgreSQL 8.0.2 on i686-pc-linux-gnu, '
'compiled by GCC gcc (GCC) 3.4.2 20041017 (Red '
'Hat 3.4.2-6.fc3), Redshift 1.0.30840'}}}

```



# Appendix B : The Curious Step-Plan

So, the existing system tables are impossible to use and as such I've been developing a set of replacement system tables for a couple of years.

One of the views produces what I term a "step-plan". It's a bit like an `EXPLAIN`, except it tells the truth, but can only be used on a query which has completed. It's a list, in stream, segment and step order, per slice, of the steps which occurred, and some information about each one. So it's an actual, real listing of what a query really did. You'll see one shortly, in this appendix.

While I was working away on the investigation, one of the queries I issued is this;

```
insert into table_1 ( distribution_key, random_value )
select 0, random() from pg_class limit 5;
```

The idea is to get one leader-node query producing five rows, each with one random number, which then is written to a worker-node table.

Well, turns out that doesn't do what I thought it would, because *any* interaction with a worker-node means the worker nodes do the work, and so in this case, you end up with random numbers from the worker node PRNGs.

But the step-plan for this query is really remarkable, so much so I've made this appendix for it.

So here's what happens from the start up to the query;

```
dev=# create table table_1
dev=# (
dev(#   distribution_key   int2       not null encode raw distkey,
dev(#   random_value      float8    not null encode raw
dev(# )
dev=# diststyle key;
CREATE TABLE
dev=#
dev=# set seed = 0;
SET
dev=# insert into table_1 ( distribution_key, random_value ) select 0, random() from pg_cl
INSERT 0 5
```

```
dev=#
dev=# select random_value from table_1;
      random_value
-----
0.892646380714737
0.579246389527981
0.462220476811851
0.743841628806479
0.363176232529966
(5 rows)
```

Now for easy reference, here's the first ten numbers from the leader node PRNG, so we can see what we have up there in that table is from a worker node PRNG.

Value
0.840187716763467
0.394382926635444
0.783099223393947
0.798440033104271
0.911647357512265
0.197551369201392
0.335222755558789
0.768229594454169
0.277774710673839
0.553969955537468

Now the step-plan.

dev=# select * from sf_queries_step_plan where qid = 520;													
qid	stream	segment	step	node_id	slice_id	step_type	rows	bytes	start_time	duration	schematable_name	notes	
520	0	0	0	0	12813	scan	1835	0	2021-10-11 10:01:15	0.000718	pg_catalog.pg_class		
520	0	0	1		12813	project	1835		2021-10-11 10:01:15	0.000718			
520	0	0	2		12813	distribute	1835	0	2021-10-11 10:01:15	0.000718			
520	0	1	0	0	0	scan	6	96	2021-10-11 10:01:15	0.001774		scan data from network to temp table	
520	0	1	0	0	1	scan	6	96	2021-10-11 10:01:15	0.001599		scan data from network to temp table	
520	0	1	0	1	2	scan	6	96	2021-10-11 10:01:15	0.001671		scan data from network to temp table	
520	0	1	0	1	3	scan	6	96	2021-10-11 10:01:15	0.001495		scan data from network to temp table	
520	0	1	1	0	0	limit	6		2021-10-11 10:01:15	0.001774			
520	0	1	1	0	1	limit	6		2021-10-11 10:01:15	0.001599			
520	0	1	1	1	2	limit	6		2021-10-11 10:01:15	0.001671			
520	0	1	1	1	3	limit	6		2021-10-11 10:01:15	0.001495			
520	0	1	2	0	0	return	5	80	2021-10-11 10:01:15	0.001774			
520	0	1	2	0	1	return	5	80	2021-10-11 10:01:15	0.001599			
520	0	1	2	1	2	return	5	80	2021-10-11 10:01:15	0.001671			
520	0	1	2	1	3	return	5	80	2021-10-11 10:01:15	0.001495			
520	0	2	0		12813	scan	6	96	2021-10-11 10:01:15	0.000171		scan data from network to temp table	
520	0	2	1		12813	limit	6		2021-10-11 10:01:15	0.000171			
520	0	2	2		12813	project	5		2021-10-11 10:01:15	0.000171			
520	0	2	3		12813	project	5		2021-10-11 10:01:15	0.000171			
520	0	2	6		12813	distribute	5	0	2021-10-11 10:01:15	0.000171			
520	0	3	0	0	0	scan	0	0	2021-10-11 10:01:15	0.004097		scan data from network to temp table	
520	0	3	0	0	1	scan	0	0	2021-10-11 10:01:15	0.004096		scan data from network to temp table	
520	0	3	0	1	2	scan	0	0	2021-10-11 10:01:15	0.00508		scan data from network to temp table	
520	0	3	0	1	3	scan	5	80	2021-10-11 10:01:15	0.00519		scan data from network to temp table	
520	0	3	1	0	0	project	0		2021-10-11 10:01:15	0.004097			
520	0	3	1	0	1	project	0		2021-10-11 10:01:15	0.004096			
520	0	3	1	1	2	project	0		2021-10-11 10:01:15	0.00508			
520	0	3	1	1	3	project	5		2021-10-11 10:01:15	0.00519			
520	0	3	2	0	0	insert	0		2021-10-11 10:01:15	0.004097	public.table_1		
520	0	3	2	0	1	insert	0		2021-10-11 10:01:15	0.004096	public.table_1		
520	0	3	2	1	2	insert	0		2021-10-11 10:01:15	0.00508	public.table_1		
520	0	3	2	1	3	insert	5		2021-10-11 10:01:15	0.00519	public.table_1		
520	0	3	3	0	0	aggregate	1	8	2021-10-11 10:01:15	0.004097		ungrouped, scalar aggregation in memory	
520	0	3	3	0	1	aggregate	1	8	2021-10-11 10:01:15	0.004096		ungrouped, scalar aggregation in memory	
520	0	3	3	1	2	aggregate	1	8	2021-10-11 10:01:15	0.00508		ungrouped, scalar aggregation in memory	
520	0	3	3	1	3	aggregate	1	8	2021-10-11 10:01:15	0.00519		ungrouped, scalar aggregation in memory	
520	1	4	0	0	0	scan	1	8	2021-10-11 10:01:15	8.4e-05		scan data from temp table	
520	1	4	0	0	1	scan	1	8	2021-10-11 10:01:15	0.000108		scan data from temp table	
520	1	4	0	1	2	scan	1	8	2021-10-11 10:01:15	8.3e-05		scan data from temp table	
520	1	4	0	1	3	scan	1	8	2021-10-11 10:01:15	7.8e-05		scan data from temp table	

```

520 | 1 | 4 | 1 | 0 | 0 | return | 1 | 8 | 2021-10-11 10:01:15 | 8.4e-05 |
520 | 1 | 4 | 1 | 0 | 1 | return | 1 | 8 | 2021-10-11 10:01:15 | 0.000108 |
520 | 1 | 4 | 1 | 1 | 2 | return | 1 | 8 | 2021-10-11 10:01:15 | 8.3e-05 |
520 | 1 | 4 | 1 | 1 | 3 | return | 1 | 8 | 2021-10-11 10:01:15 | 7.8e-05 |
520 | 1 | 5 | 0 | | 12813 | scan | 4 | 32 | 2021-10-11 10:01:15 | 0.001215 |
520 | 1 | 5 | 1 | | 12813 | aggregate | 1 | 16 | 2021-10-11 10:01:15 | 0.001215 |
520 | 2 | 6 | 0 | | 12813 | scan | 1 | 16 | 2021-10-11 10:01:15 | 4.3e-05 |
520 | 2 | 6 | 1 | | 12813 | return | 0 | 0 | 2021-10-11 10:01:15 | 4.3e-05 |
(48 rows)

```

scan data from network to temp table  
ungrouped, scalar aggregation in memory  
scan data from temp table

Now, the first thing to note is the leader node has slice\_id 12813 (at least in this query - it used to be leader node slice\_id was always 6411, but now it seems (I've not investigated, so just offhand observations) to be a value starting at 12811 and varying by query).

So what happens first is the leader node reads all 1835 rows from `pg_class`, does not apply the limit clause, and distributes the rows to the worker nodes - but the number of bytes read is 0, and that makes sense, because no columns are read from `pg_class`.

All of the worker slices then scan six rows from the network (the distribution from the leader node in the previous step), and these six rows are 96 bytes in length.

This is fascinating.

First, why six? why not five? bug in the data, or is it really happening?

Second, we can see each row is 16 bytes in length, which is correct - one 8 byte `timestamp`, one 8 byte `float8` - so it looks like the leader node has actually produced the `timeofday()` and the `random()` number. If it has, then it's *done* the work, and it's sent the rows out the worker slices - but we know from inspecting the numbers in the table (as we know what numbers the leader node produces) we have PRNG output from the worker slices, not the leader node! and this implies the worker slice later *overwrote* the work done by the leader node!

The worker slices then apply the limit clause, and each returns 5 rows to the leader node (limit has to work this way, because none of the slices can know what the others returned, and all the others might have returned zero rows; but it's still a really useful thing to do because although the number of rows is still the limit value for every slice, that's still very likely to be a hell of a lot less than the *all* the rows).

We now return to the leader node, which scans - apparently - *six* rows from the network to a temp table.

I don't get this at all. I'd expect it to read 20 - 5 from each worker slice. Maybe it knows enough to apply the limit here? and the six is an off-by-one error?

We then see the actual limit step, getting us down to 5 rows, and then the project steps are for organizing columns, so not important for us here, and then the leader node again distributes the rows it has to the worker slices, this time for the insert.

The SQL has been arranged so all rows are going to be held on slice 3, so we see slices 0 to 2 have nothing, and slice 3 has all 5 rows and 80 bytes.

The insert then happens, and after that with the aggregate, scan and return, and the leader node processing, we're seeing the computation of how many rows each worker slice inserted, with this information being returned to the user.

# Appendix C : Dieharder Results

There is a utility, **dieharder**, which applies a large range of statistical tests to PRNG output.

This is available on my Debian system from the Debian repository, but it's not necessarily available on other systems, and since the script which produces evidence for each white paper is specifically intended to be used by readers, I have not made it a dependency and so I have not included in the script the code I used to generate data for it.

I have however included the core code here (everything except the code which sets up a cluster and makes a connection).

I generated 10,000,000 32 bit values from the Python 3, leader node and worker slice 0 PRNGs, all from seed 0.

This was of course with a single query producing all values, as this is the working use case for Redshift.

To do with Redshift involves a slight element of uncertainty. The PRNG generates a value between 0.0 and 1.0, so it's not possible to know how many bits of random data are actually being generated by each call. I am then assuming it's at least 32 bits. If it's less, then the dieharder report will be invalid (as it is, given the PRNGs did well, it looks like it was okay).

## Core Code

```
import array

print( 'Python PRNG data' )

random.seed( 0 )
data = []
for loop in range( 0, 10000000 ):
    data.append( random.randint(0, 4294967296) )
bf = array.array( 'L', data )
diskfile = open( 'prng_output_python.dat', 'wb' )
bf.tofile( diskfile )
```

```

diskfile.close()

print( 'Leader node PRNG data' )

issue_sql( connection_state, 'set seed = 0;' )
sql = 'select random() from pg_class as p1, pg_class as p2, pg_class as p3 limit 10000000;'
rows, row_count = issue_sql( connection_state, sql )
data = []
for row in rows:
    data.append( int(row[0] * float(4294967296)) )
bf = array.array( 'L', data )
diskfile = open( 'prng_output_leader.dat', 'wb' )
bf.tofile( diskfile )
diskfile.close()

print( 'Worker node PRNG data' )

finished = False

while finished == False:
    issue_sql( connection_state, 'set seed = 0;' )
    sql = 'select slice_num(), random() from table_1 limit 10000000;'
    rows, row_count = issue_sql( connection_state, sql )
    if rows[0][0] == 0:
        data = []
        for row in rows:
            data.append( int(row[1] * float(4294967296)) )
        bf = array.array( 'L', data )
        diskfile = open( 'prng_output_worker.dat', 'wb' )
        bf.tofile( diskfile )
        diskfile.close()
        finished = True

```

## Python 3.7.3

```

=====#
#           dieharder version 3.31.1 Copyright 2003 Robert G. Brown           #
#=====#
    rng_name      |           filename           |rands/second|
    mt19937|           prng_output_python.dat|  4.18e+07  |
#=====#
    test_name      |ntup| tsamples |psamples|  p-value |Assessment
#=====#
    diehard_birthdays|  0|    100|    100|0.91021316|  PASSED
    diehard_operm5|  0| 1000000|    100|0.62391805|  PASSED
    diehard_rank_32x32|  0|   40000|    100|0.86047849|  PASSED
    diehard_rank_6x8|  0|   10000|    100|0.98245643|  PASSED
    diehard_bitstream|  0|  2097152|    100|0.11901796|  PASSED
    diehard_opso|  0|  2097152|    100|0.61501742|  PASSED

```

diehard_oqso	0	2097152	100 0.92247033	PASSED
diehard_dna	0	2097152	100 0.98797513	PASSED
diehard_count_1s_str	0	256000	100 0.56215375	PASSED
diehard_count_1s_byt	0	256000	100 0.83967377	PASSED
diehard_parking_lot	0	12000	100 0.18735442	PASSED
diehard_2dsphere	2	8000	100 0.83203791	PASSED
diehard_3dsphere	3	4000	100 0.92342130	PASSED
diehard_squeeze	0	100000	100 0.48740781	PASSED
diehard_sums	0	100	100 0.76343976	PASSED
diehard_runs	0	100000	100 0.54612314	PASSED
diehard_runs	0	100000	100 0.81644432	PASSED
diehard_craps	0	200000	100 0.48726638	PASSED
diehard_craps	0	200000	100 0.87719203	PASSED
marsaglia_tsang_gcd	0	10000000	100 0.10068775	PASSED
marsaglia_tsang_gcd	0	10000000	100 0.68567671	PASSED
sts_monobit	1	100000	100 0.38639892	PASSED
sts_runs	2	100000	100 0.03197373	PASSED
sts_serial	1	100000	100 0.47083513	PASSED
sts_serial	2	100000	100 0.09539431	PASSED
sts_serial	3	100000	100 0.91055561	PASSED
sts_serial	3	100000	100 0.37027202	PASSED
sts_serial	4	100000	100 0.85957342	PASSED
sts_serial	4	100000	100 0.74293426	PASSED
sts_serial	5	100000	100 0.26417245	PASSED
sts_serial	5	100000	100 0.98590100	PASSED
sts_serial	6	100000	100 0.50015357	PASSED
sts_serial	6	100000	100 0.23763476	PASSED
sts_serial	7	100000	100 0.27050504	PASSED
sts_serial	7	100000	100 0.06887525	PASSED
sts_serial	8	100000	100 0.37025966	PASSED
sts_serial	8	100000	100 0.45902758	PASSED
sts_serial	9	100000	100 0.36453740	PASSED
sts_serial	9	100000	100 0.47091905	PASSED
sts_serial	10	100000	100 0.13823639	PASSED
sts_serial	10	100000	100 0.41825651	PASSED
sts_serial	11	100000	100 0.72056127	PASSED
sts_serial	11	100000	100 0.60510631	PASSED
sts_serial	12	100000	100 0.16750470	PASSED
sts_serial	12	100000	100 0.08681430	PASSED
sts_serial	13	100000	100 0.62165433	PASSED
sts_serial	13	100000	100 0.78142778	PASSED
sts_serial	14	100000	100 0.13083346	PASSED
sts_serial	14	100000	100 0.48723282	PASSED
sts_serial	15	100000	100 0.69722467	PASSED
sts_serial	15	100000	100 0.49267886	PASSED
sts_serial	16	100000	100 0.92524386	PASSED
sts_serial	16	100000	100 0.56477191	PASSED
rgb_bitdist	1	100000	100 0.65634586	PASSED
rgb_bitdist	2	100000	100 0.49532436	PASSED
rgb_bitdist	3	100000	100 0.47939635	PASSED

rgb_bitdist	4	100000	100 0.21084338	PASSED
rgb_bitdist	5	100000	100 0.08389079	PASSED
rgb_bitdist	6	100000	100 0.98785894	PASSED
rgb_bitdist	7	100000	100 0.06729915	PASSED
rgb_bitdist	8	100000	100 0.99625180	WEAK
rgb_bitdist	9	100000	100 0.31705621	PASSED
rgb_bitdist	10	100000	100 0.72738052	PASSED
rgb_bitdist	11	100000	100 0.74987130	PASSED
rgb_bitdist	12	100000	100 0.70277477	PASSED
rgb_minimum_distance	2	10000	1000 0.20711437	PASSED
rgb_minimum_distance	3	10000	1000 0.95847441	PASSED
rgb_minimum_distance	4	10000	1000 0.73968906	PASSED
rgb_minimum_distance	5	10000	1000 0.26091900	PASSED
rgb_permutations	2	100000	100 0.26802939	PASSED
rgb_permutations	3	100000	100 0.06202086	PASSED
rgb_permutations	4	100000	100 0.48429432	PASSED
rgb_permutations	5	100000	100 0.15929449	PASSED
rgb_lagged_sum	0	1000000	100 0.99843521	WEAK
rgb_lagged_sum	1	1000000	100 0.40708004	PASSED
rgb_lagged_sum	2	1000000	100 0.52568156	PASSED
rgb_lagged_sum	3	1000000	100 0.84796544	PASSED
rgb_lagged_sum	4	1000000	100 0.13148988	PASSED
rgb_lagged_sum	5	1000000	100 0.55538232	PASSED
rgb_lagged_sum	6	1000000	100 0.55227698	PASSED
rgb_lagged_sum	7	1000000	100 0.91975633	PASSED
rgb_lagged_sum	8	1000000	100 0.84070991	PASSED
rgb_lagged_sum	9	1000000	100 0.48917896	PASSED
rgb_lagged_sum	10	1000000	100 0.22535359	PASSED
rgb_lagged_sum	11	1000000	100 0.14795671	PASSED
rgb_lagged_sum	12	1000000	100 0.96211921	PASSED
rgb_lagged_sum	13	1000000	100 0.96644383	PASSED
rgb_lagged_sum	14	1000000	100 0.48127863	PASSED
rgb_lagged_sum	15	1000000	100 0.28678048	PASSED
rgb_lagged_sum	16	1000000	100 0.94131471	PASSED
rgb_lagged_sum	17	1000000	100 0.98791186	PASSED
rgb_lagged_sum	18	1000000	100 0.52883737	PASSED
rgb_lagged_sum	19	1000000	100 0.98808888	PASSED
rgb_lagged_sum	20	1000000	100 0.74873563	PASSED
rgb_lagged_sum	21	1000000	100 0.49714591	PASSED
rgb_lagged_sum	22	1000000	100 0.46740536	PASSED
rgb_lagged_sum	23	1000000	100 0.17966965	PASSED
rgb_lagged_sum	24	1000000	100 0.95621705	PASSED
rgb_lagged_sum	25	1000000	100 0.53383031	PASSED
rgb_lagged_sum	26	1000000	100 0.07479719	PASSED
rgb_lagged_sum	27	1000000	100 0.75256734	PASSED
rgb_lagged_sum	28	1000000	100 0.63078770	PASSED
rgb_lagged_sum	29	1000000	100 0.94394862	PASSED
rgb_lagged_sum	30	1000000	100 0.85699054	PASSED
rgb_lagged_sum	31	1000000	100 0.24538772	PASSED
rgb_lagged_sum	32	1000000	100 0.55014932	PASSED

rgb_kstest_test	0	10000	1000	0.08125982	PASSED
dab_bytedistrib	0	51200000	1	0.21319946	PASSED
dab_dct	256	50000	1	0.11947066	PASSED
Preparing to run test	207.	ntuple = 0			
dab_filltree	32	15000000	1	0.53754123	PASSED
dab_filltree	32	15000000	1	0.46607264	PASSED
Preparing to run test	208.	ntuple = 0			
dab_filltree2	0	5000000	1	0.08515787	PASSED
dab_filltree2	1	5000000	1	0.57370950	PASSED
Preparing to run test	209.	ntuple = 0			
dab_monobit2	12	65000000	1	0.88376526	PASSED

## Redshift Leader Node (dc2.large, 2 nodes, 1.0.30840)

```

=====#
#               dieharder version 3.31.1 Copyright 2003 Robert G. Brown               #
=====#
  rng_name      |      filename      |rands/second|
  mt19937|      prng_output_leader.dat|  8.64e+07  |
=====#
  test_name      |ntup| tsamples |psamples|  p-value |Assessment
=====#
  diehard_birthdays| 0|    100|    100|0.56283496| PASSED
  diehard_operm5| 0|  1000000|    100|0.78028660| PASSED
  diehard_rank_32x32| 0|   40000|    100|0.08944208| PASSED
  diehard_rank_6x8| 0|   10000|    100|0.00439802|  WEAK
  diehard_bitstream| 0| 2097152|    100|0.13079436| PASSED
  diehard_opso| 0| 2097152|    100|0.02301681| PASSED
  diehard_oqso| 0| 2097152|    100|0.20625063| PASSED
  diehard_dna| 0| 2097152|    100|0.70311401| PASSED
  diehard_count_1s_str| 0|  256000|    100|0.88127953| PASSED
  diehard_count_1s_byt| 0|  256000|    100|0.94238111| PASSED
  diehard_parking_lot| 0|   12000|    100|0.77995133| PASSED
  diehard_2dsphere| 2|    8000|    100|0.99544227|  WEAK
  diehard_3dsphere| 3|    4000|    100|0.97261935| PASSED
  diehard_squeeze| 0|  100000|    100|0.24671327| PASSED
  diehard_sums| 0|    100|    100|0.24126608| PASSED
  diehard_runs| 0|  100000|    100|0.92369114| PASSED
  diehard_runs| 0|  100000|    100|0.28035254| PASSED
  diehard_craps| 0|  200000|    100|0.62306358| PASSED
  diehard_craps| 0|  200000|    100|0.65024299| PASSED
  marsaglia_tsang_gcd| 0| 10000000|    100|0.34694024| PASSED
  marsaglia_tsang_gcd| 0| 10000000|    100|0.30239745| PASSED
  sts_monobit| 1|  100000|    100|0.82007359| PASSED
  sts_runs| 2|  100000|    100|0.84952560| PASSED
  sts_serial| 1|  100000|    100|0.61632763| PASSED
  sts_serial| 2|  100000|    100|0.25207783| PASSED

```



sts_serial	3	100000	100 0.99652792	WEAK
sts_serial	3	100000	100 0.34499257	PASSED
sts_serial	4	100000	100 0.83698688	PASSED
sts_serial	4	100000	100 0.54069181	PASSED
sts_serial	5	100000	100 0.13307143	PASSED
sts_serial	5	100000	100 0.00435777	WEAK
sts_serial	6	100000	100 0.45527377	PASSED
sts_serial	6	100000	100 0.01378227	PASSED
sts_serial	7	100000	100 0.86954268	PASSED
sts_serial	7	100000	100 0.92394266	PASSED
sts_serial	8	100000	100 0.96815681	PASSED
sts_serial	8	100000	100 0.73415977	PASSED
sts_serial	9	100000	100 0.59889026	PASSED
sts_serial	9	100000	100 0.83489705	PASSED
sts_serial	10	100000	100 0.26949493	PASSED
sts_serial	10	100000	100 0.04740705	PASSED
sts_serial	11	100000	100 0.66772942	PASSED
sts_serial	11	100000	100 0.84818719	PASSED
sts_serial	12	100000	100 0.47073317	PASSED
sts_serial	12	100000	100 0.21470601	PASSED
sts_serial	13	100000	100 0.20460068	PASSED
sts_serial	13	100000	100 0.47596139	PASSED
sts_serial	14	100000	100 0.45279396	PASSED
sts_serial	14	100000	100 0.63986527	PASSED
sts_serial	15	100000	100 0.86743415	PASSED
sts_serial	15	100000	100 0.99189999	PASSED
sts_serial	16	100000	100 0.38905282	PASSED
sts_serial	16	100000	100 0.92489209	PASSED
rgb_bitdist	1	100000	100 0.16886848	PASSED
rgb_bitdist	2	100000	100 0.61718315	PASSED
rgb_bitdist	3	100000	100 0.95878298	PASSED
rgb_bitdist	4	100000	100 0.15758441	PASSED
rgb_bitdist	5	100000	100 0.68642455	PASSED
rgb_bitdist	6	100000	100 0.98846650	PASSED
rgb_bitdist	7	100000	100 0.77423360	PASSED
rgb_bitdist	8	100000	100 0.07418954	PASSED
rgb_bitdist	9	100000	100 0.27655315	PASSED
rgb_bitdist	10	100000	100 0.36952839	PASSED
rgb_bitdist	11	100000	100 0.90331503	PASSED
rgb_bitdist	12	100000	100 0.76369571	PASSED
rgb_minimum_distance	2	10000	1000 0.90089954	PASSED
rgb_minimum_distance	3	10000	1000 0.71580309	PASSED
rgb_minimum_distance	4	10000	1000 0.89880503	PASSED
rgb_minimum_distance	5	10000	1000 0.38681646	PASSED
rgb_permutations	2	100000	100 0.38359949	PASSED
rgb_permutations	3	100000	100 0.99333476	PASSED
rgb_permutations	4	100000	100 0.84905096	PASSED
rgb_permutations	5	100000	100 0.38148715	PASSED
rgb_lagged_sum	0	1000000	100 0.06409286	PASSED
rgb_lagged_sum	1	1000000	100 0.80057461	PASSED

rgb_lagged_sum	2	1000000	100 0.09333652	PASSED
rgb_lagged_sum	3	1000000	100 0.91841324	PASSED
rgb_lagged_sum	4	1000000	100 0.99895328	WEAK
rgb_lagged_sum	5	1000000	100 0.49388161	PASSED
rgb_lagged_sum	6	1000000	100 0.93171221	PASSED
rgb_lagged_sum	7	1000000	100 0.37124762	PASSED
rgb_lagged_sum	8	1000000	100 0.20428193	PASSED
rgb_lagged_sum	9	1000000	100 0.57891575	PASSED
rgb_lagged_sum	10	1000000	100 0.99323346	PASSED
rgb_lagged_sum	11	1000000	100 0.17333651	PASSED
rgb_lagged_sum	12	1000000	100 0.46028373	PASSED
rgb_lagged_sum	13	1000000	100 0.27928022	PASSED
rgb_lagged_sum	14	1000000	100 0.00015476	WEAK
rgb_lagged_sum	15	1000000	100 0.16088120	PASSED
rgb_lagged_sum	16	1000000	100 0.82912224	PASSED
rgb_lagged_sum	17	1000000	100 0.76751496	PASSED
rgb_lagged_sum	18	1000000	100 0.09242822	PASSED
rgb_lagged_sum	19	1000000	100 0.81326560	PASSED
rgb_lagged_sum	20	1000000	100 0.98568170	PASSED
rgb_lagged_sum	21	1000000	100 0.86151519	PASSED
rgb_lagged_sum	22	1000000	100 0.28338323	PASSED
rgb_lagged_sum	23	1000000	100 0.41254617	PASSED
rgb_lagged_sum	24	1000000	100 0.85047392	PASSED
rgb_lagged_sum	25	1000000	100 0.76482908	PASSED
rgb_lagged_sum	26	1000000	100 0.24530833	PASSED
rgb_lagged_sum	27	1000000	100 0.09792350	PASSED
rgb_lagged_sum	28	1000000	100 0.98778393	PASSED
rgb_lagged_sum	29	1000000	100 0.10214293	PASSED
rgb_lagged_sum	30	1000000	100 0.36574781	PASSED
rgb_lagged_sum	31	1000000	100 0.71726766	PASSED
rgb_lagged_sum	32	1000000	100 0.08671454	PASSED
rgb_kstest_test	0	10000	1000 0.26474364	PASSED
dab_bytedistrib	0	51200000	1 0.81035173	PASSED
dab_dct	256	50000	1 0.90913523	PASSED
Preparing to run test	207.	ntuple = 0		
dab_filltree	32	15000000	1 0.73063369	PASSED
dab_filltree	32	15000000	1 0.42227581	PASSED
Preparing to run test	208.	ntuple = 0		
dab_filltree2	0	5000000	1 0.08031804	PASSED
dab_filltree2	1	5000000	1 0.46052720	PASSED
Preparing to run test	209.	ntuple = 0		
dab_monobit2	12	65000000	1 0.32862265	PASSED

## Redshift Worker Slice 0 (dc2.large, 2 nodes, 1.0.30840)

```
#=====#
# dieharder version 3.31.1 Copyright 2003 Robert G. Brown #
```

```

=====#
rng_name | filename | rands/second|
mt19937 | prng_output_worker.dat | 7.35e+07 |
=====#
test_name | ntup | tsamples | psamples | p-value | Assessment
=====#
diehard_birthdays | 0 | 100 | 100 | 0.98552888 | PASSED
diehard_operm5 | 0 | 1000000 | 100 | 0.23622484 | PASSED
diehard_rank_32x32 | 0 | 40000 | 100 | 0.41156151 | PASSED
diehard_rank_6x8 | 0 | 100000 | 100 | 0.26737892 | PASSED
diehard_bitstream | 0 | 2097152 | 100 | 0.28538664 | PASSED
diehard_opso | 0 | 2097152 | 100 | 0.84712054 | PASSED
diehard_oqso | 0 | 2097152 | 100 | 0.87412791 | PASSED
diehard_dna | 0 | 2097152 | 100 | 0.85570330 | PASSED
diehard_count_1s_str | 0 | 256000 | 100 | 0.37831511 | PASSED
diehard_count_1s_byt | 0 | 256000 | 100 | 0.70217420 | PASSED
diehard_parking_lot | 0 | 12000 | 100 | 0.98448969 | PASSED
diehard_2dsphere | 2 | 8000 | 100 | 0.83764169 | PASSED
diehard_3dsphere | 3 | 4000 | 100 | 0.16493064 | PASSED
diehard_squeeze | 0 | 100000 | 100 | 0.46299607 | PASSED
diehard_sums | 0 | 100 | 100 | 0.02395836 | PASSED
diehard_runs | 0 | 100000 | 100 | 0.72829432 | PASSED
diehard_runs | 0 | 100000 | 100 | 0.90137611 | PASSED
diehard_craps | 0 | 200000 | 100 | 0.02756788 | PASSED
diehard_craps | 0 | 200000 | 100 | 0.76981361 | PASSED
marsaglia_tsang_gcd | 0 | 10000000 | 100 | 0.92486587 | PASSED
marsaglia_tsang_gcd | 0 | 10000000 | 100 | 0.82768046 | PASSED
sts_monobit | 1 | 100000 | 100 | 0.64653275 | PASSED
sts_runs | 2 | 100000 | 100 | 0.30846507 | PASSED
sts_serial | 1 | 100000 | 100 | 0.88152740 | PASSED
sts_serial | 2 | 100000 | 100 | 0.99979017 | WEAK
sts_serial | 3 | 100000 | 100 | 0.58904454 | PASSED
sts_serial | 3 | 100000 | 100 | 0.61237276 | PASSED
sts_serial | 4 | 100000 | 100 | 0.10318355 | PASSED
sts_serial | 4 | 100000 | 100 | 0.23471200 | PASSED
sts_serial | 5 | 100000 | 100 | 0.14776957 | PASSED
sts_serial | 5 | 100000 | 100 | 0.94426624 | PASSED
sts_serial | 6 | 100000 | 100 | 0.43325597 | PASSED
sts_serial | 6 | 100000 | 100 | 0.32473662 | PASSED
sts_serial | 7 | 100000 | 100 | 0.46231799 | PASSED
sts_serial | 7 | 100000 | 100 | 0.57226154 | PASSED
sts_serial | 8 | 100000 | 100 | 0.75570832 | PASSED
sts_serial | 8 | 100000 | 100 | 0.63532223 | PASSED
sts_serial | 9 | 100000 | 100 | 0.41982537 | PASSED
sts_serial | 9 | 100000 | 100 | 0.02958411 | PASSED
sts_serial | 10 | 100000 | 100 | 0.55335691 | PASSED
sts_serial | 10 | 100000 | 100 | 0.29778251 | PASSED
sts_serial | 11 | 100000 | 100 | 0.89033787 | PASSED
sts_serial | 11 | 100000 | 100 | 0.69445886 | PASSED
sts_serial | 12 | 100000 | 100 | 0.70028280 | PASSED

```

sts_serial	12	100000	100 0.37050033	PASSED
sts_serial	13	100000	100 0.82209080	PASSED
sts_serial	13	100000	100 0.19545550	PASSED
sts_serial	14	100000	100 0.25804521	PASSED
sts_serial	14	100000	100 0.80358025	PASSED
sts_serial	15	100000	100 0.89065155	PASSED
sts_serial	15	100000	100 0.98312073	PASSED
sts_serial	16	100000	100 0.27731748	PASSED
sts_serial	16	100000	100 0.51880516	PASSED
rgb_bitdist	1	100000	100 0.20214026	PASSED
rgb_bitdist	2	100000	100 0.17699662	PASSED
rgb_bitdist	3	100000	100 0.69509415	PASSED
rgb_bitdist	4	100000	100 0.65004993	PASSED
rgb_bitdist	5	100000	100 0.99160017	PASSED
rgb_bitdist	6	100000	100 0.81048325	PASSED
rgb_bitdist	7	100000	100 0.97492388	PASSED
rgb_bitdist	8	100000	100 0.58948307	PASSED
rgb_bitdist	9	100000	100 0.46751274	PASSED
rgb_bitdist	10	100000	100 0.89638556	PASSED
rgb_bitdist	11	100000	100 0.52860331	PASSED
rgb_bitdist	12	100000	100 0.03558371	PASSED
rgb_minimum_distance	2	10000	1000 0.37767141	PASSED
rgb_minimum_distance	3	10000	1000 0.16219517	PASSED
rgb_minimum_distance	4	10000	1000 0.88737620	PASSED
rgb_minimum_distance	5	10000	1000 0.04465040	PASSED
rgb_permutations	2	100000	100 0.99236354	PASSED
rgb_permutations	3	100000	100 0.52719628	PASSED
rgb_permutations	4	100000	100 0.76115210	PASSED
rgb_permutations	5	100000	100 0.73836345	PASSED
rgb_lagged_sum	0	1000000	100 0.29463994	PASSED
rgb_lagged_sum	1	1000000	100 0.10403868	PASSED
rgb_lagged_sum	2	1000000	100 0.79554102	PASSED
rgb_lagged_sum	3	1000000	100 0.99019672	PASSED
rgb_lagged_sum	4	1000000	100 0.87160049	PASSED
rgb_lagged_sum	5	1000000	100 0.63526097	PASSED
rgb_lagged_sum	6	1000000	100 0.78164429	PASSED
rgb_lagged_sum	7	1000000	100 0.01414177	PASSED
rgb_lagged_sum	8	1000000	100 0.91120273	PASSED
rgb_lagged_sum	9	1000000	100 0.56019077	PASSED
rgb_lagged_sum	10	1000000	100 0.88819428	PASSED
rgb_lagged_sum	11	1000000	100 0.13060075	PASSED
rgb_lagged_sum	12	1000000	100 0.28264461	PASSED
rgb_lagged_sum	13	1000000	100 0.44541582	PASSED
rgb_lagged_sum	14	1000000	100 0.29494481	PASSED
rgb_lagged_sum	15	1000000	100 0.33588815	PASSED
rgb_lagged_sum	16	1000000	100 0.55206878	PASSED
rgb_lagged_sum	17	1000000	100 0.75623644	PASSED
rgb_lagged_sum	18	1000000	100 0.80892139	PASSED
rgb_lagged_sum	19	1000000	100 0.98600606	PASSED
rgb_lagged_sum	20	1000000	100 0.32999864	PASSED

rgb_lagged_sum	21	1000000	100 0.60481492	PASSED
rgb_lagged_sum	22	1000000	100 0.91392204	PASSED
rgb_lagged_sum	23	1000000	100 0.93116828	PASSED
rgb_lagged_sum	24	1000000	100 0.09761773	PASSED
rgb_lagged_sum	25	1000000	100 0.94670973	PASSED
rgb_lagged_sum	26	1000000	100 0.14857654	PASSED
rgb_lagged_sum	27	1000000	100 0.17125207	PASSED
rgb_lagged_sum	28	1000000	100 0.44112337	PASSED
rgb_lagged_sum	29	1000000	100 0.37278694	PASSED
rgb_lagged_sum	30	1000000	100 0.02877012	PASSED
rgb_lagged_sum	31	1000000	100 0.56022772	PASSED
rgb_lagged_sum	32	1000000	100 0.20013696	PASSED
rgb_kstest_test	0	10000	1000 0.42363767	PASSED
dab_bytedistrib	0	51200000	1 0.32558216	PASSED
dab_dct	256	50000	1 0.40829491	PASSED
Preparing to run test	207.	ntuple = 0		
dab_filltree	32	15000000	1 0.15439468	PASSED
dab_filltree	32	15000000	1 0.97294404	PASSED
Preparing to run test	208.	ntuple = 0		
dab_filltree2	0	5000000	1 0.65211516	PASSED
dab_filltree2	1	5000000	1 0.00399712	WEAK
Preparing to run test	209.	ntuple = 0		
dab_monobit2	12	65000000	1 0.49219072	PASSED

## Summary

Python was weak for the following tests;

rgb_bitdist	8	100000	100 0.99625180	WEAK
rgb_lagged_sum	0	1000000	100 0.99843521	WEAK

The leader node was weak for the following tests;

diehard_rank_6x8	0	100000	100 0.00439802	WEAK
diehard_2dsphere	2	8000	100 0.99544227	WEAK
sts_serial	3	100000	100 0.99652792	WEAK
sts_serial	5	100000	100 0.00435777	WEAK
rgb_lagged_sum	4	1000000	100 0.99895328	WEAK
rgb_lagged_sum	14	1000000	100 0.00015476	WEAK

Worker slice 0 was weak for the following tests;

sts_serial	2	100000	100 0.99979017	WEAK
dab_filltree2	1	5000000	1 0.00399712	WEAK

# Redshift Observatory Slack

I've started up a Redshift Slack.

Join URL is;

[https://join.slack.com/t/redshiftobservatory/shared\\_invite/zt-2vm3deqis-he6h4GMDcG6Gs7~IECQNuQ](https://join.slack.com/t/redshiftobservatory/shared_invite/zt-2vm3deqis-he6h4GMDcG6Gs7~IECQNuQ)