Cross-Database Queries

Max Ganz II @ Redshift Observatory

9th August 2021

Abstract

Cross-database queries are not implemented by improvements in the use of the system tables such that Redshift can now access tables across databases, but rather by bringing over from the remote database to the local database a copy of the table being queried, leading to a duplicate of the table in every database issuing cross-database queries to that table. Where the remote table is not on local disk, the first query is slow, as it must wait for a local copy of the table to be made and similarly, when updates are made to the remote table, the next query after the updates is slow, as it must bring the updates over to the local copy of the table.

Contents

Introduction

Redshift in October 2020 introduced cross-database queries.

As far as actually writing SQL is concerned, this meant that the previous three part "schema.table.column" notation now supports an extra stanza, becoming "database.schema.table.column".

The very natural thought is that Redshift has been improved such that where before the system tables could only support cross-schema queries, they can now support cross-database queries. This would mean that the new functionality is provided by improvements to the use of the system tables and that cross-database queries behave in the same way as cross-schema queries.

However, there are in the official docs enumerated for cross-database queries a wide range of limitation and in particular, that this functionality is only available on RA3 type nodes. This hints that there is more to the implementation than meets the eye, and as such this document investigates cross-database queries.

Test Method

Performance Test

A two-node `ra3.xlplus` cluster is created. Three databases are created, 'local', 'remote_1' and 'remote_2'.

An identical unsorted `even` distribution table with a single raw encoded not null `float8` column is created in each database, each populated with the same number of rows, each row generated by the `random()` function and so being a value between 0.0 and 1.0, with each table fully vacuumed (to 100%) and analyzed (with threshold set to 0).

We then connect to the 'local' database, and all test queries are issued from this connection.

There is in fact only one test query, which obtains the `sum()` of the data in one of the test tables; so we need to read each row, but we have very little network traffic coming back.

We measure the time taken for the query by examining `STL_WLM_QUERY` and `SVL_COMPILE`, the former giving the total execution time, the latter giving compile time. Subtracting compile time from total execution time gives the run time of the query.

We first issue this query on the 'local' database (so not using a cross-database query, as we are connected to the 'local' database'), and then on each of the 'remote' databases.

The work of making the tables in the databases, populating them and querying them, is repeated three times, where the number of rows is varied, in the sequence 1000, 1m, 10m.

The results are the run times taken for the queries, on the local and the remote tables, for each number of rows.

Timings are taken directly from the system tables, and so mark the times internal to Redshift as to when the query execution started and ended (queuing and compile times are excluded).

Note this is investigation is not a benchmark, but rather a method to discover behaviour, and so the usual arrangement of repeating a benchmark five times, with the slowest and fastest test results being discarded, is not appropriate;

we're looking to see how behaviour *changes* over queries, rather than figuring out the usual speed of a query.

General Proofs

Having performed *ad hoc* experimentation to figure out what goes on behind the scenes with cross-database queries, I put together a sequence of queries to demonstrate the findings.

The following work is performed;

1. create two databases, 'local' and 'remote'
2. connect once to 'local', twice to 'remote'
3. create and populate a test table on 'local'
4. issue a cross-database query on session #1 connected to remote (and note total disk used on cluster)
5. issue a cross-database query on session #2 connected to remote (and note total disk used on cluster)
6. add 1m rows to the test table
7. issue a cross-database query on session #1 connected to remote (and note total disk used on cluster)
8. issue a cross-database query on session #2 connected to remote (and note total disk used on cluster)
9. add a column to the test table
10. issue a cross-database query on session #1 connected to remote (and note total disk used on cluster)
11. issue a cross-database query on session #2 connected to remote (and note total disk used on cluster)
12. disconnect session #1 to remote (and note total disk used on cluster)
13. disconnect session #2 to remote (and note total disk used on cluster)
14. drop the test table (and note total disk used on cluster)

Results

See Appendix A for the Python `pprint` dump of the results dictionary.

Performance Test

The X-axis is the number of rows in the table.

The Y-axis is the iteration number of the test query.

The individual results are in seconds.

In all cases, we are connected to the 'local' database, and are querying either the 'local' database (first set of results), or we are using a cross-database query with one of the 'remote' databases.

The first connection has its queries issued first, then we move onto the second connection.

ra3.xlplus, 2 nodes (1.0.28965)

local (connection #0)

	1000	1000000	10000000
1	0.005	0.007	0.016
2	0.005	0.005	0.016
3	0.004	0.005	0.016

local (connection #1)

	1000	1000000	10000000
1	0.005	0.007	0.021
2	0.005	0.005	0.015
3	0.005	0.007	0.015

remote_1 (connection #0)

	1000	1000000	10000000
1	0.208	0.141	0.740
2	0.005	0.005	0.020
3	0.006	0.007	0.020

remote_1 (connection #1)

	1000	1000000	10000000
1	0.006	0.006	0.016
2	0.027	0.010	0.020
3	0.005	0.006	0.017

remote_2 (connection #0)

	1000	1000000	10000000
1	0.135	0.171	0.618
2	0.005	0.011	0.021
3	0.005	0.007	0.023

remote_2 (connection #1)

	1000	1000000	10000000
1	0.005	0.007	0.020
2	0.005	0.006	0.016
3	0.005	0.006	0.019

General Proofs

ra3.xlplus, 2 nodes (1.0.28965)

Annotations have been added by the author; they are not emitted by the test code.

Note that Redshift is constantly issuing queries of its own, and so the total number of blocks used on a cluster can by this change, independently of the test run. This may account for small discrepancies in the number of blocks at any given time.

0. Cluster version = 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 1.0.28965
1. Created databases named ‘local’ and ‘remote’.
2. Connected to ‘local’ database.
3. Connected to ‘remote’ database (session #1).
4. Connected to ‘remote’ database (session #2).

5. Initial total disk used on cluster = 647 blocks.
    - MG2 : curiously, a much higher value than the previous version of Redshift
6. Made and populated test table in 'local' database.
7. Table size = 220 blocks.
8. Total disk used on cluster = 868 blocks.
    - MG2 : the off-by-one could well be from Redshift's own ongoing queries running in the background
9. Cross-database query issued in session 1 (local table `localhost@101691@175@`.
    - MG2 : this table name, found via `STL_SCAN`, cannot be found in `pg_class`, or indeed in any system table;
10. Total disk used on cluster = 1087 blocks.
    - MG2 : so, having issued a cross-database query onto the 220 block test table, we see the total disk used 11.
11. Cross-database query issued in session 2 (local table `localhost@101691@175@`.
12. Total disk used on cluster = 1087 blocks.
    - MG2 : here though in the second session to the remote database, unlike the previous version of Redshift, issuing another cross-database query does not lead to another (per-session) copy of the test table; so we see now cross-database queries when they copy the remote table are making a single database-wide copy
13. Added 1m rows to test table.
14. Table size = 240 blocks.
    - MG2 : adding 1m rows added 20 blocks to the test table
15. Total disk used on cluster = 1112 blocks.
16. Cross-database query re-issued in session 1 (local table `localhost@101691@175@`.
17. Total disk used on cluster = 1131 blocks.
    - MG2 : re-issuing the cross-database query from session 1 consumed another 20 blocks of disk, and we see the 18. Cross-database query re-issued in session 2 (local table `localhost@101691@175@`.
18. Total disk used on cluster = 1131 blocks.
    - MG2 : no changes, as expected, since there is only one local copy per database
19. Test table altered, one column added.
20. Table size now = 328 blocks.
    - MG2 : adding a new column consumed an extra 328 - 240 = 88 blocks
21. Total disk used on cluster = 1221 blocks.
22. Cross-database query re-issued in session 1 (local table `localhost@101691@204@`.
23. Total disk used on cluster = 1548 blocks.
    - MG2 : now, we added a column to the test table and re-issued the cross-database query, and we see the local copy table name has changed, and the total disk used on the cluster has increased by 1497-1169 = 326 blocks, so we can safely say adding a new column has made this a new table from the point of view of the query, and so it has been brought over in full, and also that the original copy has not been deleted
24. Cross-database query re-issued in session 2 (local table `localhost@101691@204@`.
25. Total disk used on cluster = 1548 blocks.
26. Session 1 disconnected.
27. Total disk used on cluster = 1548 blocks.

28. Session 2 disconnected.
29. Total disk used on cluster = 1548 blocks.
    - MG2 : now this is interesting; we've disconnected both sessions, so there are no connections now to the 31. Dropped test table.
30. Total disk used on cluster = 1221 blocks.
    - MG2 : and this is even more remarkable; the source table has been dropped, we recovered its 328 blocks, but both the first copy (prior to adding a column) and the second copy still persist - I have no idea when or how they go away

Discussion

There are then two sets of results to discuss; first, the performance tests, second, the general proofs.

I begin with the performance tests because the numbers from these tests immediate show something of great interest, namely that when issuing a cross-database query, the very first time the query is issued, *it is slow*, but the performance test queries after that run at normal speed.

If we look at the first query issued from database `remote_1`, using connection #1, we see that with 1k records, the first query took 0.208 seconds; the second query took 0.005 seconds.

Similarly, with the 10m records query, the first query took 0.740 seconds, the second took 0.016.

By comparison, normal queries (a *non* cross-database query) for 1k rows and 10m rows take 0.005 and 0.016 seconds, respectively.

We see also that this slow first query is on a per-database basis, as the cross-database queries after the first run query, regardless of which connection issues the query.

So what's going on?

It seems clear that whatever it is that *is* happening, cross-database queries are not implemented by improved use of the system tables, because the behaviour we're seeing now is distinctly different to cross-schema queries.

In particular, we note the time taken for the first query varies depending on the number of rows in the table, but after that, performance is identical to working with a normal, local table. It could be a *copy* of the table is being made.

To progress this further we need now to turn to the general proofs.

Now, the system table `STL_SCAN` fairly recently was enhanced so that it carries the name of the table being scanned. Before this it carried only the ID of the table being scanned, but this is no use if that table has gone away by the time you come to look up the ID.

In the general proofs, lines 8, 9 and 10, show the number of blocks used before a cross-database query, then the table name the scan step read, and then the number of blocks after the cross-database query.

We see that before the query we had 821 blocks, that the scan step read a table named `localhost@101691@175@`, and after the query we had 1040 blocks. The size of the remote table is 220 blocks.

To my utter surprise, that table cannot be found in the system tables.

It is clear that it has been brought into existence, because of the disk space which has been consumed and which continues to be consumed after the cross-database query has completed, and also because we see further cross-database queries to the same remote table work at the same speed as local copies.

What is happening then is that a cross-database query makes a local copy, on a per-database basis, of the remote table.

I am however very concerned that this table *cannot be found anywhere in the system tables.*

It appears to be completely invisible - whilst, of course, consuming disk space; and the more you use cross-database queries, the more blocks will be consumed by these invisible tables, *and you have no way of knowing how much disk they are using.*

If what I'm seeing really is so, this is extraordinarily wrong. It is a profound blunder. This directly obstructs cluster admins from being able to run their own clusters; how can you manage a cluster when you cannot know where disk is being consumed, or how that consumption is changing over time?

This issue in fact dovetails into an open question revealed in the proofs, which I will bring into play here as it is directly relevant : it is not clear when local copies are deallocated.

In fact, the local copies do not go away *even when the original table the copy is dropped.*

They must have *some* mechanism to retire them, but it not yet apparent what that mechanism is.

I think we can now move properly onto the general proofs, and draw out the details of the cross-database query mechanism.

First, the general proofs demonstrate what has been deduced from the performance tests, that a cross-database query makes an invisible local copy of the remote table, on a per-database basis.

Second, we see that changes (new rows, etc) to the remote table are propagated to copies when and only when the remote table is queried again after the changes to the remote table have occurred. So, when a remote table is changed, propagation is lazy : it occurs on a per-copy basis, when and only when the local copy actually comes to need those new rows.

Accordingly, if the remote table changes more often than you query, every time you query, data has to be brought over from the remote table; and the query necessarily will when this happens be slower than normal.

I would speculate the extra time required will be composed of a fixed part, and then a part which varies depending on how much data has to be brought over; but I have not tested this.

This leads to a critical question; can local copies and the remote table differ in how sorted they are? Imagine we have a remote table, which is badly disordered, and we issue a cross-database query. Do we get the equivalent of an `INSERT/SELECT`, in which case the local copy will be fully sorted? or do we get an exact copy of what's in the remote table?

The answer to this question is critical because it determines whether or not a merge join can occur, and without knowing that, use of remote tables is crippled by this uncertainty; the whole point of Redshift is the merge join [1], and not knowing if it will or will reliably occur by itself, regardless of all the other questions of performance when copying over large numbers of rows, partially obviates remote queries : if you can't know for sure when you will or will not get merge joins, you have to assume you'd only get hash joins.

However, given that we don't want to bringing over Big Data tables anyway, we can reasonably in fact say merge joins don't matter that much; we'd never want to be issuing a cross-database query on a large table anyway.

As ever, the lack of documentation makes it impossible to knowingly correctly design systems using Redshift. This is a critical and profound weaknesses of Redshift : almost all systems are designed by people groping in a pitch-black room while wearing blindfolds. To the utter astonishment of absolutely no-one at all, the systems they produce do not perform well, and consequently, now they finally have another option, they are exiting to Snowflake.

Moving on, we next in the general proofs see that a major change to the remote table, in this case adding a column, means that when the next cross-database query is issued, a new and full local copy is made.

My guess here is that any change which would modify the DDL of the remote table leads to this behaviour.

Somewhat eyebrow-raisingly, the local copies which existed prior to the new column being added are not deallocated, and indeed we then next see that even when we drop the remote table which is the source of the local copies, the local copies are *still* not deallocated. It is not clear when the invisible local copies are deallocated.

Cross-database queries epitomise the failure of Redshift to give people using it to make systems the knowledge necessary to knowingly design correct systems, but even more than this, Redshift very nearly actively misleads developers, by the complete silence with regard to information so salient, that *by its absence* users are being led to imagine behaviour which is not true.

If you make a car, and driving it causes it to explode, *not* telling users is to lead them into thinking it's a normal car, because you would of course, being a rational and normal person, realise you did in fact have to let the car drivers know about this behaviour.

If you make cross-database queries, and you're making local copies of the remote

[1] Redshift under the hood has three methods by which it implements all SQL joins; the merge join, the hash join and the nested loop join. Only when a merge join is used can two Big Data tables be joined in a timely manner and without hammering the cluster for all users, but merge joins have a number of often difficult to meet requirements.

table, *not* telling developers is to lead them into thinking such queries behave like normal queries. Why or how would they ever imagine for themselves the complex and unexpected edifice of invisible per-database local copies? Why should they *have* to anyway? this isn't a cryptic cross-word puzzle. You're supposed to *tell* developers what they need to know.

I know from experience if you point this out to the devs or Support, they will tell you there is no documentation because it is internal implementation and subject to change. This is certainty true - it is subject to change - *but it is what it is right now and it's so important developers needed to know.*

Moreover, I can't see any reason why there can't be a box or heading on the documentation page which explains the important properties of the current implementation, and which is updated if and when that implementation significantly changes and this change is noted with a bullet point in the release notes. Isn't that what documentation is *for*?

If we take a step back and looking at the larger picture, what we find is that the more we imagine wide-spread use of cross-database queries, the more we imagine every database being duplicated into every other database.

This is obviously not viable for wide-spread use.

So in the first place we can only think about limited use only; and then when we realize that a local copy is going to be made, we obviously also don't want to use cross-database queries on large tables.

In fact, we anyway couldn't use cross-database queries to issue joins on two Big Data tables, since we have no idea if the copies differ in how sorted they are to the remote tables, so we can't know if merge joins will be issued, and you can only join two Big Data tables with a merge join; we have to assume hash joins only, which are - in the normal case - limited to one small table and one Big Data table, but here we don't want to make a local copy of a Big Data table, so cross-database queries are limited to joining two small tables only.

(Of course, you can do what you like - you certainly can issue a cross-database query on a pair of Big Data tables. Redshift won't stop you. It won't warn you. It just won't end well, for you or anyone else on the cluster.)

Next matter of note is that we cannot track the amount of disk being consumed by copies.

Then we have also the published set of limitations;

1. Views can't use remote tables
2. No column-level privileges
3. Queries using remote tables can't go to concurrency scaling clusters
4. Results from queries using remote tables do not go into the query result cache

Note though although I've not tested this, I think with this implementation of making local copies Redshift avoids a limitation found in Postgres. In Postgres, a cross-database query can only use tables *from a single database*. (I guess it's kinda temporarily "swapping database" for the query). Redshift where it makes a local copy of the table is still only using tables from a single database, and

so I suspect/expect Redshift can in its queries use tables from multiple remote databases.

Next consideration is where cross-database queries are supported only by RA3, if you build a system which uses them, you've pinned yourself to RA3 node types; you can't swap back to DC/DS types.

Then there's a hard limit in Redshift to 60 databases, so you can't have that many of them.

Note that Redshift organizes users, groups and privileges separately from databases, so you still need to correctly arrange all your access privileges (including default privileges) for the schemas and tables you have, regardless of the database they're in; all you get really is that the "schema.table.column" notation gains an extra stanza, and becomes "database.schema.table.column", and the ability to use on a per-user basis the two privileges available on databases, the privilege to create schemas, and the privilege to create temp tables.

All in all, I can't see any reason at all, except if you find yourself in a tight spot due to blunders or unexpected demands on your data design such that a cross-database query is just the ticket and saves you a lot of costly work, to use cross-database queries.

You *certainly* don't want them as part of your normal system design.

This functionality also brings a black-box into your system design.

During the writing of the first version of this white paper, the implementation of cross-database queries changed. It went from per-session copies, so plenty of disk overhead, to per-database copies, much less overhead, but where it's now not clear how deallocation of copies occurs. To my knowledge this change is completely undocumented and no notification has been given, anywhere. I'm *not* a fan of my systems silently and without warning changing their behaviour.

All in all, I can't see why cross-database queries were introduced, and I do think very nearly almost all developers are going to be using them only by dint of being mislead, by the silence of the documentation, with regard to how they work : and given how cross-database queries *do* in fact work, what we actually have are developers, by their lack of knowledge, producing poor systems with erratic performance, then having no clue why performance is poor and erratic.

I am of the view this is true in any number of areas of Redshift use, and this is the main and direct cause of the loss of Redshift clients to Snowflake.

Conclusions

The method used to implement cross-database queries is to bring down a copy of the remote database table to a table in the local database, and then query that local table.

Naturally, the larger the remote database table, the longer this takes. On a two node ra3.xlplus cluster, the time for 1k rows is about 0.11 seconds. 10m rows, the largest tested, takes about 0.52 seconds. This compares to local access of about 0.006 and 0.017 seconds, respectively.

The local copy is on a per-database basis. It is unclear when or how local copies are deallocated; even dropping the remote table does not lead to deallocation.

Worryingly, the local copy of the remote table appears to be completely invisible in the system tables.

This directly impacts the capability of a cluster administrator to administer a cluster, because it seems it is no longer possible to see how many blocks all tables are using; it seems impossible to find out how much disk is being consumed by cross-database query table copies.

At the time a remote database table is modified, the changes are not brought over to copies of the table. Updating occurs when another cross-database query is issued on a remote database table, and so occurs independently across the remote copies.

It's not clear if the ordering of the rows in the local copy can come to differ to the ordering of the rows in the remote copy. If it can, then it can be that merge joins cannot be issued on the local copy, while they could be issued on the original remote table; where this isn't clear, it is only safe to assume only hash joins can be used. Given this, and the overhead of copying a large table over the network, cross-database queries can only be used to read or join small tables.

If a remote table undergoes a significant change (the single tested example is a new column), then when the next cross-database query occurs, the table is brought over in full. The original local copy, of the earlier version of the remote table, remains on disk. It is not clear when or how it is deallocated.

I would guess all structural changes, which is to say, changes affecting DDL, lead to a new table being brought over.

It is then that when issuing a cross-database query, there is a significant initial

delay to make a local copy of the remote table, and there are delays on later queries when-ever the remote table has been modified, and if a table changes significantly, then it is in effect as if it were a new table, and a full delay, to bring over the whole table, re-occurs.

Once a local copy exists, and is up-to-date, cross-database queries are indistinguishable in performance from local tables.

Finally, as mentioned, it is not clear when or how local copies of remote tables are deallocated. This was not seen to occur, even when the remote table was dropped.

If we take a step back and looking at the larger picture, what we find is that the more we imagine wide-spread use of cross-database queries, the more we imagine every database being duplicated in every other database.

This is not obviously viable for wide-spread use.

Unexpected Findings

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

1. `EXPLAIN` for `UPDATE` appears to give the wrong row width.

```
dev=# create table table_1
dev-# (
dev(#  column_1  float8  not null encode raw
dev(# )
dev-# diststyle even
dev-# compound sortkey( column_1 );
CREATE TABLE
dev=#
dev=# analyze table_1;
ANALYZE
dev=#
dev=# insert into table_1( column_1 ) values ( random() );
INSERT 0 1
dev=#
dev=# explain select * from table_1;
                        QUERY PLAN
-----------------------------------------------------------
 XN Seq Scan on table_1  (cost=0.00..0.01 rows=1 width=8)
(1 row)

dev=#
dev=# explain update table_1 set column_1 = 1.0;
                        QUERY PLAN
-----------------------------------------------------------
 XN Seq Scan on table_1  (cost=0.00..0.01 rows=1 width=6)
(1 row)
```

   The plan for `SELECT` indicates a width of eight (which is correct), but `UPDATE` gives a width of six.

2. Two node ra3.xlplus clusters take a lot longer - about three times as long - to come up as two node dc2.large clusters.

Further Questions

1. When are local table copies deallocated?
2. When a remote table is updated, and then a cross-database query is issued, what is the nature of the delay for bringing over the new data? is there a fixed length delay, to get things set up, followed by a variable length delay depending on the number of rows being brought over? or is it different?
3. If a source table is dis-ordered (not fully vacuumed) and a cross-database query is issued, is the local copy sorted when it is brought over?
4. If a source table is ordered when a copy is made, and then updated and so becomes fully dis-ordered, does the remote copy, when next queried and so brought up to date, also become fully dis-ordered?
5. If a source table is disordered, then copies are made, and then the source table is vacuumed, and then the copies are again queried, what happens?

Revision History

v1

- Initial release.

v2

- Redshift's internal implementation of cross-database queries changed from making local copies of the remote table in the form of temporary tables on a per-session basis, to local copies in the form of a completely invisible (not present in the system tables) per-database copy of the remote table.
- Metadata changes.

v3

- Correction in the Discussion, "0.06" changed to "0.005". Nod to Transient_Simian from reddit.

v4

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

v5

- Added "About the Author". made site name in title a link, and made each chapter start a new page.

v6

- Web-site name changed to "Redshift Observatory".
- Updated links from redshiftresearcproject.org to redshift-observatory.ch.

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': {'ra3.xlplus': {2: ['Cluster version = 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 '
                               '1.0.28965',
                               "Created databases named 'local' and 'remote'.",
                               "Connected to 'local' database.",
                               "Connected to 'remote' database (session #1).",
                               "Connected to 'remote' database (session #2).",
                               'Initial total disk used on cluster = 647 '
                               'blocks.',
                               "Made and populated test table in 'local' "
                               'database.',
                               'Table size = 220 blocks.',
                               'Total disk used on cluster = 868 blocks.',
                               'Cross-database query issued in session 1 '
                               '(local table `localhost@101691@175@`.',
                               'Total disk used on cluster = 1087 blocks.',
                               'Cross-database query issued in session 2 '
                               '(local table `localhost@101691@175@`.',
                               'Total disk used on cluster = 1087 blocks.',
                               'Added 1m rows to test table.',
                               'Table size = 240 blocks.',
                               'Total disk used on cluster = 1112 blocks.',
                               'Cross-database query re-issued in session 1 '
                               '(local table `localhost@101691@175@`.',
                               'Total disk used on cluster = 1131 blocks.',
                               'Cross-database query re-issued in session 2 '
                               '(local table `localhost@101691@175@`.',
                               'Total disk used on cluster = 1131 blocks.',
                               'Test table altered, one column added.',
                               'Table size now = 328 blocks.',
                               'Total disk used on cluster = 1221 blocks.',
                               'Cross-database query re-issued in session 1 '
```

```
                                  '(local table `localhost@101691@204@`.',
                                  'Total disk used on cluster = 1548 blocks.',
                                  'Cross-database query re-issued in session 2 '
                                  '(local table `localhost@101691@204@`.',
                                  'Total disk used on cluster = 1548 blocks.',
                                  'Session 1 disconnected.',
                                  'Total disk used on cluster = 1548 blocks.',
                                  'Session 2 disconnected.',
                                  'Total disk used on cluster = 1548 blocks.',
                                  'Dropped test table.',
                                  'Total disk used on cluster = 1221 blocks.']}},
 'tests': {'ra3.xlplus': {2: {1000: {'local': {0: [0.00512799999999999,
                                                    0.005321,
                                                    0.004414],
                                               1: [0.005342,
                                                    0.005342,
                                                    0.005017]},
                                     'remote_1': {0: [0.207598,
                                                      0.005157,
                                                      0.006149],
                                                  1: [0.0056,
                                                      0.026549,
                                                      0.004884]},
                                     'remote_2': {0: [0.134573,
                                                      0.004792,
                                                      0.005012],
                                                  1: [0.005099,
                                                      0.004914,
                                                      0.00498]}},
                              1000000: {'local': {0: [0.007007,
                                                      0.005311,
                                                      0.005395],
                                                  1: [0.007311,
                                                      0.005376,
                                                      0.007153]},
                                        'remote_1': {0: [0.140918,
                                                         0.005469,
                                                         0.006944],
                                                     1: [0.006271,
                                                         0.009979,
                                                         0.006328]},
                                        'remote_2': {0: [0.171059,
                                                         0.010971,
                                                         0.006767],
                                                     1: [0.007179,
                                                         0.005989,
                                                         0.006191]}},
                              10000000: {'local': {0: [0.015953,
                                                       0.015755,
                                                       0.016106],
```

```
                                                  1: [0.021459,
                                                      0.015404,
                                                      0.015353]},
                                          'remote_1': {0: [0.740273,
                                                           0.019952,
                                                           0.020437],
                                                       1: [0.015614,
                                                           0.019559,
                                                           0.016512]},
                                          'remote_2': {0: [0.618019,
                                                           0.021284,
                                                           0.02333],
                                                       1: [0.01993,
                                                           0.0156,
                                                           0.019368]}}}}},
'versions': {'ra3.xlplus': {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 1.0.28965'}}}
```

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.