Materialized Views

Max Ganz II @ Redshift Observatory

5th September 2021

Abstract

Materialized views are implemented as a normal table, a normal view and a procedure, all created by the `CREATE MATERIALIZED VIEW` command, where the procedure is called by the `REFRESH MATERIALIZED VIEW` command and performs refresh. The table is created by `CREATE TABLE AS`, which is why column encodings cannot be specified. The encoding choices made by Redshift are extremely poor. A full refresh makes a new table, populates it, and uses table rename to replace the existing table. An incremental refresh uses an `insert` followed by a `delete`, using the system columns `deletexid` and `insertxid` to keep track of which rows have changed, and as such runs a full refresh when any of the tables used by the materialized view have either manually or automatically been vacuumed, as vacuum resets the values in the `deletexid` and `insertxid` columns and so invalidates the book-keeping information held by the materialized view, this information being stored in extra columns in the materialized view, one plus one for every table used in the materialized view SQL. The table underlying the materialized view is never vacuumed, except by auto-vacuum, which I suspect running so infrequently as to be inconsequential. Auto-refresh is an undocumented black box, likely subject to ongoing unannounced change, and its behaviour is unpredictable. On a small single column table on an idle cluster refresh occurred normally after about 55 seconds; with one row inserted per second, refresh occurred after between 54 to 1295 seconds (twenty-one minutes).

Contents

Introduction

Redshift added support for materialized views near the end of 2019.

Materialized views are a method for pre-computing the results of a query, so that when the results come to be used, the time and work to compute them has already been expended.

A Redshift materialized view is defined in the same way as a normal view, as an SQL statement, but unlike a normal view - where the name of the view is replaced by its SQL in the text of an SQL query issued against the view - a materialized view actually produces the rows of the SQL defining the materialized view and stores them on disk, in a table, and a query issued against the materialized view actually uses that table.

The concept of pre-computing results is useful and widespread, but I aver there are in Redshift's implementation numerous design and implementation flaws which ensure that materialized views are the *exact* same amount of development work and complexity, but with much less performance, than manually creating and maintaining your own pre-computed results, and as such, there are no circumstances where it is correct to use them.

This document then examines the internal implementation of materialized views and assesses and critiques their behaviour.

Test Method

There are quite a few tests.

Auto-Refresh

An empty single column table is created, and then a materialized view, with auto refresh on, is created, which uses that table.

The table then has ten exactly full blocks inserted, and then the materialized view is monitored, once per second, to time how long auto-refresh takes.

The test is then repeated, but now immediately before each check of the materialized view, a single row is inserted into the underlying table, on every check.

This is repeated five times, to give a range of auto-refresh times. Since this is not a performance test in the usual sense, but more of a discovery, where extremes matter, all five times form the results.

Creation and Refresh

A normal, empty table is created, and then a materialized view is created using that table. Auto refresh is off, and a window function is used in the materialized view definition to ensure a full refresh is in use. The SQL command sequence induced by creating the materialized view is captured and examined.

Then a single row is inserted into the underlying table, and the materialized view is refreshed. The SQL command sequence induced by refreshing the materialized view is captured and examined, to obtain insight into the internal implementation of materialized views.

The test is then repeated, but without a window function, to allow incremental refresh, and captures the SQL command sequence issued in this case.

Column Encodings

A table is created with one column per data type. A materialized view is created, using this table. The encodings selected by Redshift are then taken from the system tables.

Additional Columns

Two tables each with a single column is created. A full and an incremental materialized view are created using one table in their SQL, and a second full and incremental materialized view are created using both tables in their SQL. In all cases, the full list of columns in the materialized view is taken from the system tables.

VACUUM

A table with a single column is created. An incremental refresh materialized view is created using this table. A sequence of `INSERT` and `REFRESH MATERIALIZED VIEW` occur, where it is noted if the refreshes are full or incremental. The table ends up with some sorted and some unsorted rows and a `VACUUM` is issued on the table, and then one more `REFRESH`, again noting the refresh type of the final refresh.

A table with a single column is created. An incremental refresh materialized view is created using this table. A sequence of `INSERT` and `REFRESH MATERIALIZED VIEW` occur, where it is noted if the refreshes are full or incremental. The table ends up with some sorted and some unsorted rows and a `VACUUM` is issued on the materialized view, noting the number of sorted and unsorted rows before and after. Next, a `VACUUM` is issued on the table underlying the materialized view, noting the number of sorted and unsorted rows before and after, and also then issuing a `REFRESH`, to see if this `VACUUM` induced a full refresh.

Results

Test duration was 2,948 seconds.

All times are in seconds.

dc2.large, 2 nodes (1.0.29551)

Auto-Refresh Delay (Count Only)

Often the first and second iterations vary considerably, and then times settle to about 55 seconds. Note though the test fully tears down the test environment between each test, so it's a little strange we seem to see state persisting over test runs.

Iteration	Delay
0	11.06
1	54.31
2	53.53
3	55.13
4	54.03

Auto-Refresh Delay (Count With Insert)

These results are typical. Making the table busy, by inserting to it, seems to throw off the auto-refresh algorithm. On the other hand, I also saw exactly one run which looked exactly like the "Count Only" test, above; first run about 10 seconds, rest about 55 seconds.

Iteration	Delay
0	240.62
1	54.23
2	1294.94
3	542.90
4	176.52

Materialized View Column Encoding Choices

These are the encoding choices made by Redshift for a materialized view, given empty tables being used by the SQL for the materialized view.

Ordinal	Name	Data Type	Encoding
-9	deletexid	int8	raw
-8	insertxid	int8	raw
-7	tableoid	oid	raw
-6	cmax	cid	raw
-5	xmax	xid	raw
-4	cmin	cid	raw
-3	xmin	xid	raw
-2	oid	oid	raw
-1	ctid	tid	raw
1	column_01	bool	raw
2	column_02	char(256)	lzo
3	column_03	char(64)	lzo
4	column_04	date	az64
5	column_05	float4	raw
6	column_06	float8	raw
7	column_07	geometry	raw
8	column_08	hllsketch	raw
9	column_09	int2	az64
10	column_10	int4	az64
11	column_11	int8	az64
12	column_12	numeric(19,0)	az64
13	column_13	numeric(38,0)	az64
14	column_14	varchar(256)	lzo
15	column_15	time	az64
16	column_16	timestamp	az64
17	column_17	timestamptz	az64
18	column_18	timetz	az64
19	column_19	varchar(64)	lzo
20	table_1_oid	int8	az64
21	num_rec	int4	az64

VACUUM Source Table

"Materialized view mv_1 was incrementally updated successfully" is given for an incremental refresh.

"Materialized view mv_1 was recomputed successfully" is given for a full refresh.

We see here a `VACUUM` on the source table forces a full refresh.

```
1. create table table_1
           (
             column_1  bigint not null encode raw
           )
```

```
           diststyle even
           compound sortkey( column_1 );
2. create materialized view mv_1
             diststyle even
             compound sortkey( column_1 )
             auto refresh no
           as
             select
               *
             from
               table_1;
3. insert into table_1( column_1 ) values ('1'), ('2'), ('3'), ('4'), ('5'), ('6');
4. refresh materialized view mv_1;
5. INFO:  Materialized view mv_1 was incrementally updated successfully.
6. insert into table_1( column_1 ) values ('1'), ('2'), ('3'), ('4'), ('5'), ('6');
7. refresh materialized view mv_1;
8. INFO:  Materialized view mv_1 was incrementally updated successfully.
9. insert into table_1( column_1 ) values ('1'), ('2'), ('3'), ('4'), ('5'), ('6');
10. vacuum full table_1 to 100 percent;
11. refresh materialized view mv_1;
12. INFO:  Materialized view mv_1 was recomputed successfully.
```

VACUUM Materialized View Underlying Table

The key rows here are the counts of sorted and unsorted rows.

We can see on lines 9, 10 and 11 a `VACUUM` of the materialized view did nothing.

We can see on lines 12 and 13, `VACUUM` of the underlying table sorted the underlying table, and we then see on lines 14 and 15, this did not then force a full refresh.

```
1. create table table_1
           (
             column_1  bigint not null encode raw
           )
           diststyle even
           compound sortkey( column_1 );
2. create materialized view mv_1
             diststyle even
             compound sortkey( column_1 )
             auto refresh no
           as
             select
               *
             from
               table_1;
3. insert into table_1( column_1 ) values ('1'), ('2'), ('3'), ('4'), ('5'), ('6');
4. refresh materialized view mv_1;
5. INFO:  Materialized view mv_1 was incrementally updated successfully.
6. insert into table_1( column_1 ) values ('1'), ('2'), ('3'), ('4'), ('5'), ('6');
```

```
7. refresh materialized view mv_1;
8. INFO:  Materialized view mv_1 was incrementally updated successfully.
9. 6 sorted rows, 6 unsorted rows
10. vacuum full mv_1 to 100 percent;
11. 6 sorted rows, 6 unsorted rows
12. vacuum full mv_tbl__mv_1__0 to 100 percent;
13. 12 sorted rows, 0 unsorted rows
14. insert into table_1( column_1 ) values ('1'), ('2'), ('3'), ('4'), ('5'), ('6');
15. refresh materialized view mv_1;
16. INFO:  Materialized view mv_1 was incrementally updated successfully.
```

Materialized View Full Column List (One Source Table, Full Refresh)

The `row_number` column is part of the source table, it's used to force a full refresh.

We see here there are no additional columns.

Ordinal	Name	Data Type	Encoding
-9	deletexid	int8	raw
-8	insertxid	int8	raw
-7	tableoid	oid	raw
-6	cmax	cid	raw
-5	xmax	xid	raw
-4	cmin	cid	raw
-3	xmin	xid	raw
-2	oid	oid	raw
-1	ctid	tid	raw
1	row_number	int8	az64
2	column_1	int8	raw

Materialized View Full Column List (Two Source Tables, Full Refresh)

The `row_number` column is part of the source table, it's used to force a full refresh.

We see here there are no additional columns.

Ordinal	Name	Data Type	Encoding
-9	deletexid	int8	raw
-8	insertxid	int8	raw
-7	tableoid	oid	raw
-6	cmax	cid	raw
-5	xmax	xid	raw
-4	cmin	cid	raw
-3	xmin	xid	raw
-2	oid	oid	raw

Ordinal	Name	Data Type	Encoding
-1	ctid	tid	raw
1	row_number	int8	az64
2	column_1	int8	raw

Table, View and Proc Counts (B/A CMV, Full Refresh)

The numbers of tables, views and procedures before and after a `CREATE MATERIALIZED VIEW`, for a full refresh materialized view.

	Before	After
Tables	947	948
Views	547	548
Procs	0	1

Materialized View Text From `pg_views` (Full Refresh)

The SQL source in `pg_views` for a full refresh materialized view. A normal view contains only the SQL statement which forms the view.

create materialized view mv_1 diststyle even compound sortkey(column_1) auto refresh no as select row_number() over (partition by column_1 order by column_1), column_1 from table_1;

CREATE MATERIALIZED VIEW (Full Refresh)

The SQL commands induced by a `CREATE MATERIALIZED VIEW` with a full refresh materialized view.

Event Time	System Table	SQL
2021-09-05 21:28:33.306758	stl_querytext	create materialized view mv_1 diststyle even compound sortkey(column_1) auto refresh no as select row_number() over (partition by column_1 order by column_1), column_1 from table_1;

Event Time	System Table	SQL
2021-09-05 21:28:33.323018	stl_ddltext	create materialized view mv_1 diststyle even compound sortkey(column_1) auto refresh no as select row_number() over (partition by column_1 order by column_1), column_1 from table_1;
2021-09-05 21:28:33.324324	stl_ddltext	create materialized view mv_1 diststyle even compound sortkey(column_1) auto refresh no as select row_number() over (partition by column_1 order by column_1), column_1 from table_1;
2021-09-05 21:28:33.326914	stl_querytext	padb_fetch_sample: select count(*) from mv_tbl__mv_1__0

REFRESH MATERIALIZED VIEW (Full Refresh)

The SQL commands induced by a `REFRESH MATERIALIZED VIEW` with a full refresh materialized view.

Event Time	SQL
2021-09-05 21:28:35.975696	REFRESH MATERIALIZED VIEW mv_1;
2021-09-05 21:28:35.977569	CALL public.mv_sp__mv_1__1_0(7434, 7439, 1, '(0)');
2021-09-05 21:28:35.989098	CREATE TABLE public.mv_tbl__mv_1__0__tmp BACKUP YES DISTSTYLE EVEN COMPOUND SORTKEY(2)AS (SELECT ROW_NUMBER() OVER (PARTITION BY "table_1"."column_1" ORDER BY "table_1"."column_1" ASC NULLS LAST) AS "row_number", "table_1"."column_1" AS "column_1" FROM "public"."table_1" AS "table_1")

Event Time	SQL
2021-09-05 21:28:36.006137	Analyze mv_tbl__mv_1__0__tmp
2021-09-05 21:28:36.006278	padb_fetch_sample: select * from mv_tbl__mv_1__0__tmp
2021-09-05 21:28:36.082477	CREATE OR REPLACE VIEW public.mv_1 AS SELECT * FROM public.mv_tbl__mv_1__0__tmp
2021-09-05 21:28:36.084287	DROP TABLE public.mv_tbl__mv_1__0
2021-09-05 21:28:36.085588	ALTER TABLE public.mv_tbl__mv_1__0__tmp RENAME TO mv_tbl__mv_1__0
2021-09-05 21:28:36.087398	CREATE OR REPLACE VIEW public.mv_1 AS SELECT * FROM public.mv_tbl__mv_1__0
2021-09-05 21:28:36.090246	COMMIT

Procedure SQL

```
mv_sp__mv_1__1_0( recompute bool, end_xid int8, start_xid int8, finished_xid_list varchar
begin
  if recompute then
    create table public.mv_tbl__mv_1__0__tmp backup yes diststyle even compound sortkey(2)
    create or replace view  public.mv_1 as select * from public.mv_tbl__mv_1__0__tmp;
    drop table public.mv_tbl__mv_1__0;
    alter table public.mv_tbl__mv_1__0__tmp rename to mv_tbl__mv_1__0;
    create or replace view public.mv_1 as select * from public.mv_tbl__mv_1__0;
  else
    delete from public.mv_tbl__mv_1__0;
    insert into public.mv_tbl__mv_1__0( select row_number() over (partition by "table_1"."
  end if;
end;
```

Materialized View Full Column List (One Source Table, Incremental Refresh)

Incremental materialized views have additional columns. Here we see the additional `num_rec` column, and then one further column, `table_1_oid`.

Ordinal	Name	Data Type	Encoding
-9	deletexid	int8	raw
-8	insertxid	int8	raw
-7	tableoid	oid	raw
-6	cmax	cid	raw
-5	xmax	xid	raw
-4	cmin	cid	raw
-3	xmin	xid	raw
-2	oid	oid	raw

Ordinal	Name	Data Type	Encoding
-1	ctid	tid	raw
1	column_1	int8	raw
2	table_1_oid	int8	az64
3	num_rec	int4	az64

Materialized View Full Column List (Two Source Tables, Incremental Refresh)

Incremental materialized views have additional columns. Here we see the additional `num_rec` column, and then two further columns, `table_1_oid` and `table_2_oid`; in general there is one additional column per table in the SQL forming the materialized view.

Ordinal	Name	Data Type	Encoding
-9	deletexid	int8	raw
-8	insertxid	int8	raw
-7	tableoid	oid	raw
-6	cmax	cid	raw
-5	xmax	xid	raw
-4	cmin	cid	raw
-3	xmin	xid	raw
-2	oid	oid	raw
-1	ctid	tid	raw
1	column_1	int8	raw
2	table_1_oid	int8	az64
3	table_2_oid	int8	az64
4	num_rec	int4	az64

Table, View and Proc Counts (B/A CMV, Incremental Refresh)

The numbers of tables, views and procedures before and after a `CREATE MATERIALIZED VIEW`, for an incremental refresh materialized view.

	Before	After
Tables	947	948
Views	547	548
Procs	0	1

Materialized View Text From `pg_views` (Incremental Refresh)

The SQL source in `pg_views` for an incremental refresh materialized view. A normal view contains only the SQL statement which forms the view.

create materialized view mv_1 diststyle even compound sortkey(column_1) auto refresh no as select column_1 from table_1;

CREATE MATERIALIZED VIEW (Incremental Refresh)

Event Time	System Table	SQL
2021-09-05 21:28:40.468523	stl_querytext	create materialized view mv_1 diststyle even compound sortkey(column_1) auto refresh no as select column_1 from table_1;
2021-09-05 21:28:40.484168	stl_ddltext	create materialized view mv_1 diststyle even compound sortkey(column_1) auto refresh no as select column_1 from table_1;
2021-09-05 21:28:40.485893	stl_ddltext	create materialized view mv_1 diststyle even compound sortkey(column_1) auto refresh no as select column_1 from table_1;
2021-09-05 21:28:40.491842	stl_querytext	padb_fetch_sample: select count(*) from mv_tbl__mv_1__0

REFRESH MATERIALIZED VIEW (Incremental Refresh)

Event Time	SQL
2021-09-05 21:28:41.664228	REFRESH MATERIALIZED VIEW mv_1;
2021-09-05 21:28:41.665943	CALL public.mv_sp__mv_1__1_0(7475, 7480, 0, '(0)');

Event Time	SQL
2021-09-05 21:28:41.678324	INSERT INTO “public”.”mv_tbl__mv_1__0” (SELECT “table_1”.”column_1” AS “column_1”, CAST(“table_1”.”oid” AS INT8) AS “table_1_oid”, CAST(1 AS INT4) AS “num_rec” FROM “public”.”table_1” AS “table_1” WHERE ((CAST(“table_1”.”insertxid” AS INT8) > 7475) OR CAST(“table_1”.”insertxid” AS INT8) IN (0)) AND ((CAST(“table_1”.”insertxid” AS INT8) <= 7480) AND (CAST(“table_1”.”deletexid” AS INT8) > 7480)))
2021-09-05 21:28:41.772182	DELETE FROM “public”.”mv_tbl__mv_1__0” USING (SELECT “table_1”.”column_1” AS “column_1”, CAST(“table_1”.”oid” AS INT8) AS “table_1_oid”, CAST(-1 AS INT4) AS “num_rec” FROM “public”.”table_1” AS “table_1” WHERE (CAST(“table_1”.”insertxid” AS INT8) <= 7475) AND (NOT CAST(“table_1”.”insertxid” AS INT8) IN (0) AND (((CAST(“table_1”.”deletexid” AS INT8) > 7475) OR CAST(“table_1”.”deletexid” AS INT8) IN (0)) AND (CAST(“table_1”.”deletexid” AS INT8) <= 7480)))) AS “mv_tbl__mv_1__0__deletes” WHERE “mv_tbl__mv_1__0”.”table_1_oid” = “mv_tbl__mv_1__0__deletes”.”table_1_oid”
2021-09-05 21:28:41.845141	COMMIT

Procedure SQL

```
mv_sp__mv_1__1_0( recompute_mv bool, end_xid int8, start_xid int8, finished_xid_list varch
begin
  if recompute_mv then
    execute $_7275328207056490759_$ create table "public"."mv_tbl__mv_1__0_recomputed" bac
```

```
    create or replace view "public"."mv_1" as (select "derived_table1"."column_1" as "colu
    drop table "public"."mv_tbl__mv_1__0";
    alter table "public"."mv_tbl__mv_1__0_recomputed" rename to "mv_tbl__mv_1__0";
    create or replace view "public"."mv_1" as (select "derived_table1"."column_1" as "colu
  else
    execute $_7275328207056490759_$ insert into "public"."mv_tbl__mv_1__0" (select "table_
    execute $_7275328207056490759_$ delete from "public"."mv_tbl__mv_1__0" using (select "
  end if;
end;
```

Discussion

Implementation

Creation

When the `CREATE MATERIALIZED VIEW` command is issued we see in the transaction four SQL statements are issued.

On the face of it, what we see is strange. We see the create command *three* times, once in `STL_QUERYTEXT` and twice in `STL_DDLTEXT`, and then we see in `STL_QUERYTEXT` a final command of `SELECT`, which is being used for a `padb_fetch_sample`, which is a query typically issued by `ANALYZE`.

Now, I have in fact observed in the past that which is logged in these system tables is *not* strictly the text of the commands issued. This can be seen here in the final command, which is a status message followed by SQL, or in the messages logged by `VACUUM`, which behaves in the same way.

If you approach these tables expecting them to behave as they are described in the docs and as their names indicate, and to contain only DDL/SQL commands, you will be disappointed. You *will* need to parse the output, it *can* be arbitrary, and you have *no* idea what it can be until you happen to see it.

What I think is actually happening here is that the text in `STL_DDLTEXT` is a bug, and the wrong SQL text is being logged.

If we examine the counts of tables, views and procedures before and after `CREATE MATERIALIZED VIEW`, we see the command creates one table, one view and one procedure.

I think what's happening is that the first command is the `CREATE MATERIALIZED VIEW` we actually issued, which must be creating the view and since this it the first command and so the table which will be created does not yet exist, it must be this view is created with late binding; and the second and third commands are actually a `CREATE PROCEDURE` followed by a `CREATE TABLE AS`, and the final command is the automatic sampling of data performed by the inherent `ANALYZE` issued by `CREATE TABLE AS`.

A materialized view, then, is a normal view (with the name passed by the caller to `CREATE MATERIALIZED VIEW`) which points at a normal table created by Redshift, which is used to store the materialized rows, but we also have a mysterious (but to be explained) procedure.

This explains why materialized views are listed in `pg_class` with the `relkind` 'v'; what we're seeing there is the view part of the materialized view.

This is though a colossal blunder, as the only way now I can get a list of normal views from `pg_class` is to list all views, and then `EXCEPT` from that list the list of materialized view names from `STV_MV_INFO`. Whomever had this happen was asleep at the wheel.

Moreover, this is in fact also a breaking change, because I have replacement system tables which show information about normal views, and they abruptly began also showing information about materialized views; and the SQL recorded in the system tables for a normal view is not the entire `CREATE VIEW` command, but only the SQL statement given to `CREATE VIEW` to define the view, but the SQL recorded in the system tables for a materialized view is the entire `CREATE MATERIALIZED VIEW` command, so it's two breaking changes.

The same problem is also seen in `pg_views`, which is now showing materialized views as well as normal views.

When it comes to databases, where they are so central and critical as core components of larger systems - like operating system kernels - breaking changes are almost verboten. If you are going to make them, you telegraph them in advance to the user base and you announce them when they happen and you document them heavily.

Silent breaking changes, with no announcement and no documentation, are so profoundly and completely off-the-map they are utterly inconceivable. No dev team would do this, from awareness of their users' needs and because of the catastrophic loss of trust and reputation - and every now and then, I see such a change being made in Redshift. It's not a one-off thing.

I think the devs are unaware of the impact their changes are having on end-users.

Refresh

It is now the mysterious procedure comes into play.

The `REFRESH MATERIALIZED VIEW` command in fact calls the procedure, which in turn contains a bit of logic and a number of SQL commands and implements refresh, be it full or incremental.

We can see the procedure text in the Results, and see that there are two different procedures, one for materialized views with full refresh, the other for materialized views with incremental refresh.

Here's how they work, remembering that being inside a procedure, every set of SQL commands is executed inside a transaction;

Full Refresh Procedure, Recompute True

1. Create new table (with a temp name)
2. Modify view to use new table
3. Drop original table
4. Rename new table to original table name

5. Modify view to use renamed new table

Full Refresh Procedure, Recompute False

1. Delete all rows in table
2. Insert all rows (by executing the materialized view SQL statement)

Incremental Refresh Procedure, Recompute True

1. Create new table (with a temp name)
2. Modify view to use new table
3. Drop original table
4. Rename new table to original table name
5. Modify view to use renamed new table

Incremental Refresh Procedure, Recompute False

1. Insert new records into table
2. Delete old records from table

Note when `recompute` is `true` the avoidance of `truncate table` and instead the creation and rename of a new table. This is because `truncate` commits the current transaction. If issued in a procedure, it commits the current transaction and a new transaction automatically and immediately begins - which means then that the table holding the rows of materialized view is for a certain period of time empty and *seen* to be empty by users of the materialized view, which isn't going to fly.

In general `truncate table` when needed inside a transaction can be replaced by the method used here.

Now, I would say in most ETL systems, it's often possible to truncate and then insert (which also means a vacuum is not required), or simply to insert.

The implementation here though of materialized views has no access to information about the system within which it is being used, and so it has to play it safe; it must use a method - delete and insert inside a transaction (which then mandates a vacuum, because this makes a mess of the underlying table) - which is *always* going to be correct, even though that means being awfully expensive in the situations where it is in fact possible to use much more efficient methods.

The behaviour then of the procedures is all pretty clear, except there is something remarkable; the path for the Incremental Refresh where `recompute` is `false` uses the system columns found in every table, `deletexid` and `insertxid`, to figure out which rows to insert and which to delete.

This is a real surprise. I tried in the past to access these columns and access was forbidden ("column does not exist"), and indeed trying this now it still doesn't work, neither when issuing SQL directly and also not in a procedure. Note Postgres does allow access to these columns.

I suspect it may be it works in these procedures because the owner of the materialized view procedure is `rdsdb`, the über-user, the user owned by Amazon,

which is more powerful than the mere system admin user allowed to whomever created the cluster. Remember - Redshift is like Android : you are not root.

VACUUM

VACUUM Induces Full Refresh

In the documentation, there is a peculiar and entirely unexplanatory paragraph which warns about an interaction between automatic background vacuum with the auto-refresh of materialized views;

> Background vacuum operations might be blocked if materialized views aren't refreshed. After an internally defined threshold period, a vacuum operation is allowed to run. When this vacuum operation happens, any dependent materialized views are marked for recomputation upon the next refresh (even if they are incremental). For information about VACUUM, see VACUUM. For more information about events and state changes, see STL_MV_STATE. [1]

When I first read that documentation paragraph, I had absolutely no idea what was going on - why on *earth* would auto-vacuum care about whether or not materialized views were updated or not?

However, I've investigated the materialized view implementation enough that I think I can now actually penetrate the murk and explain what's going on, and this in fact reveals a stupendous and truly staggering omission from the documentation.

When a materialized view is using a full refresh, there is no problem at all. There is no interaction between vacuum, or auto-vacuum, and refresh, whether it is automatic or manual.

The quoted paragraph in fact only applies to incremental refresh materialized views.

When a materialized view is using incremental refresh, it is in fact relying upon the use of the system columns `deletexid` and `insertxid`, which are in every table, to figure out what row changes have occurred, as that information is required to perform an incremental refresh.

The problem is that when a `VACUUM` - of any kind, manual or automatic - runs, it resets the values in the `insertxid` and `deletexid` columns. This of course completely messes up the record-keeping information being used by a materialized view to perform incremental refresh - and so, perforce, a full refresh has to occur.

What the documentation so saliently fails to mention is that this need for a full refresh is induced *not* only by auto-vacuum, but by *normal*, manually issued vacuum.

[1] Now, I have to say I have over the years of investigating Redshift come fully to the view to documentation is not reviewed by technical staff. I think what happens is someone explains to the author, who is not a software engineer, who then writes down his grasp of what has been said, and I think everyone, all along the line, is trying to obfuscate everything which is not a strength.

So, to be clear : every time you `VACUUM` any table used by an incremental-refresh materialized view, the next refresh will be a full refresh.

This is also true for any auto-vacuum on any table used by an incremental-refresh materialized view, but auto-vacuum seems to run so infrequently I suspect this doesn't play much of a part.

Materialized Views Are Only Auto-Vacuumed

You cannot in fact, despite appearances, directly `VACUUM` a materialized view.

The `VACUUM` command will run, it will not throw an error, and it will return the status message `VACUUM` and so it will *look* exactly like it ran, but it does not run. This is not documented.

You can `VACUUM` the underlying table which holds the rows of the materialized view. This does work, and it does what you expect; the problem is that this is not documented, and the name of this table is quite well hidden; you need to investigate the SQL commands issued by `CREATE MATERIALIZED VIEW` to find it.

As a consequence, and this is an enormous problem, the only `VACUUM` operating on materialized views is whatever is afforded by background auto-vacuum, which is regarded by myself and a number of fellow Redshift admin as running so infrequently as to have negligible effect.

Vacuuming the underlying table does not induce a full refresh (it's `deletexid` and `insertxid` columns are not used in refresh).

Finally, note that when an incremental refresh materialized view actually has an incremental refresh, an `insert` has been issued, followed by a `delete` and so at this point, a `VACUUM` is needed. For both full refresh, and an incremental refresh which runs in full refresh mode, where a new underlying table has been created from scratch and repopulated, a `VACUUM` is not needed.

Column Encodings

When you create a materialized view, you can specify the sorting type and keys, and the distribution type (and distribution column). What you *cannot* specify are the column encodings. Redshift automatically selects encodings. The problem is, in my view, Redshift makes extremely poor encoding choices and this by itself is a fatal shortcoming.

A discussion of this assertion is long and takes us a long way from materialized views, so I have moved it to Appendix B.

Auto-Refresh

Materialized views are described in the documentation as offering an option to automatically bring themselves up to date when the underlying data changes, but, critically, there are no guarantees of when this occurs, and the algorithm

which when updates occur is an undocumented, likely complex, black box which is going to be undergoing ongoing silent changes.

A quote from the docs;

> To complete refresh of the most important materialized views with minimal impact to active workloads in your cluster, Amazon Redshift considers multiple factors. These factors include current system load, the resources needed for refresh, available cluster resources, and how often the materialized views are used.

There are tens of thousands of Redshift systems out there. Real life is infinitely more complicated than we can imagine and so design for. The idea, for example, that a materialized view which is used more often is more important is simply not true; it may be a good rule of thumb, but that's small consolation if someone applies that rule of thumb to your system when it's not true.

Moreover, any complex system has design and implementation flaws. The implementation for example of AutoWLM has to my knowledge gone through at least three major rewrites, the earlier versions being complete failures and even the current version I think remaining inherently flawed, just less blatantly and up-front problematically so (such that if you tried AutoWLM, you had to disable it, since it was killing your system, as happened in the earlier versions).

In any event, I fully and completely hold the view that it is not possible to knowingly design a correct system when that system contains black boxes controlled by a third party who silently and on an ongoing basis changes their function, let alone when those black boxes are complex and likely flawed.

(This is a major problem with Redshift these days. It's happening a lot.)

In fact, another feature of Redshift, the automatic background vacuum, has exactly the same automatic update behaviour and is described in the same way in its documentation. It's been found that the automatic background vacuum runs too infrequently to be useful.

I didn't want to get into an open-ended exploration exploration of the factors involved in auto-refresh, because there's so many that could be involved, and they vary so much (which leads to the question of how you test this functionality in the first place), so I made two simple tests, just to get the beginning of a feel for behaviour;

1. Create an empty table with a single column, create a materialized view with auto-fresh active, using that table, insert a single row into the table and then poll the materialized view once per second and see how long auto-refresh takes.
2. Create an empty table with a single column, create a materialized view with auto-fresh active, using that table, insert a single row into the table and then poll the materialized view once per second and see how long auto-refresh takes, but now on every poll, insert one record into the table.

In both cases, the cluster (two node `dc2.large`) is completely idle, and we can have a reasonable expectation that refresh times will be extended the busier a cluster becomes. Both tests were repeated five times.

For the first test case, in the test run for this document, the first refresh time was ten seconds, then went to 40 seconds, then became stable at about 55 seconds. (On earlier runs, I saw refresh times as low as 3 seconds and as high as 140 seconds). In general, the first two runs vary (maybe by as much as a minute) but the later test runs settle at 55 seconds - which is still a bit odd, given the materialized view and table are both being dropped and re-created on every test run; it looks like Redshift is maintaining state of some kind relating to refresh decisions which is not tied to the given materialized view and its table(s).

For the second test case, the delay was much more variable and often much longer, ranging from 54 seconds to 1295 seconds (twenty minutes).

The obvious question is : how do you build an ETL system when one of the stages of data propagation through that system is outside of your control, you have only a rough idea of how long it might take, where that idea could be up-ended by factors you are unaware of, and where the behaviour of that stage could also change at any time, without notice?

The answer obviously is that you cannot; you need to manually issue the refresh command at the appropriate point in the ETL process. I assert auto-refresh has no value.

Additional Columns

A materialized view which uses full refresh has no extra columns added into the underlying table.

A materialized view which uses incremental refresh has a number of extra columns, the number being 1 (`num_rec`) plus 1 for every table used by the SQL of the materialized view.

Full vs Incremental Refresh

When a materialized view is created, Redshift examines the SQL and determines if the materialized view when refreshed will experience full refresh or incremental refresh, with incremental refresh being used if possible (being preferred).

If you want a materialized view with incremental refresh, there's a wide range of restrictions on the SQL which can be used to form a materialized view - none of the following are permitted;

- OUTER JOIN (RIGHT, LEFT, or FULL).
- Set operations: UNION, INTERSECT, EXCEPT, MINUS.
- UNION ALL when it occurs in a subquery and an aggregate function or a GROUP BY clause is present in the query.
- Aggregate functions: AVG, MEDIAN, PERCENTILE_CONT, LISTAGG, STDDEV_SAMP, STDDEV_POP, APPROXIMATE COUNT, APPROXIMATE PERCENTILE, and bitwise aggregate functions (Note the COUNT and SUM aggregate functions *are* supported.)
- DISTINCT aggregate functions, such as DISTINCT COUNT, DISTINCT SUM, and so on.

- Window functions.
- A query that uses temporary tables for query optimization, such as optimizing common sub-expressions.
- Sub-queries in any place other than the FROM clause.
- External tables referenced as base tables in the query that defines the materialized view.
- Mutable functions, such as date-time functions, RANDOM and non-STABLE user-defined functions.

In addition, as we've seen, when a refresh is issued on an incremental refresh materialized view after a vacuum on any of the tables it uses, a full refresh occurs anyway, and furthermore, unless you are manually issuing `VACUUM` on the table underlying the materialized view, the only vacuum work being performed on that table is that from automatic background vacuum, which I think running so infrequently it is negligible, and finally, there's an overhead of one column plus one column per table used in the materialized view SQL.

All in all, I think incremental refresh is a complete non-starter and in fact should specifically be avoided. It is in the first place very limited in the SQL it permits, and then we also find the implementation is even more than problematic - it's harmful, because of the lack of vacuum.

System Tables

Tucked away in the documentation for one of the system tables which holds information about materialized views, `STV_MV_INFO`, we discover two columns of particular interest, the first of which is `is_stale`, an `char(1)`, which sayeth the docs;

> A `t` indicates that the materialized view is stale. A stale materialized view is one where the base tables have been updated but the materialized view hasn't been refreshed. This information might not be accurate if a refresh hasn't been run since the last restart.

In other words, this column has only ambiguous information about whether or not a materialized view is up to date; you have to perform an update to initialize the information in this column.

I wanted to write a view which shows users information about their materialized views, and of course one very useful piece of information is whether or not the materialized view is up to date. I can't show that information, because the only source is this column, and I can't know if this column is accurate or not, and I do not want to *require* and *depend* upon the users reading the docs to avoid being misled (something `STV_MV_INFO` is it seems entirely happy with).

Then, next, we have the column `state`, an `integer`, which the docs have holding the following values;

> - 0 – The materialized view is fully recomputed when refreshed.
> - 1 – The materialized view is incremental.
> - 101 – The materialized view can't be refreshed due to a dropped column. This constraint applies even if the column isn't used

in the materialized view.
- 102 – The materialized view can't be refreshed due to a changed column type. This constraint applies even if the column isn't used in the materialized view.
- 103 – The materialized view can't be refreshed due to a renamed table.
- 104 – The materialized view can't be refreshed due to a renamed column. This constraint applies even if the column isn't used in the materialized view.
- 105 – The materialized view can't be refreshed due to a renamed schema. |

We see that the information about the refresh type (full or incremental) is conflated in this single column with information about failures (underlying tables or columns being altered) and so when there is a failure, it is no longer possible to know the refresh type of the materialized view.

How am I supposed to make a view on top of this, showing the refresh type?

(Moreover, as we saw earlier, the design of materialized views is such that incremental refresh views sometimes anyway will issue full refreshes, so this can never be a reliable indicator anyway.)

Conclusions

Materialized views are implemented as a table, a normal view and a procedure.

When `CREATE MATERIALIZED VIEW` is issued, Redshift actually issues a number of SQL commands, creating the table, the view, and the procedure. The view is presented to the user as the materialized view; it simply points at the table. The procedure is used to implement refresh, with `REFRESH MATERIALIZED VIEW` calling the procedure.

The table underlying the materialized view is created by the `CREATE TABLE AS` command. This is why it is not possible with materialized views to specify column encodings. I am of the view the encoding choices made by Redshift are extremely poor (indeed, in some cases, nonsensical) and as such many columns are experiencing no compression at all.

For any given materialized view, Redshift by examining at creation time the SQL of the materialized view determines whether full refresh or incremental refresh will be used. Redshift prefers incremental and will choose it where possible, but to be possible, a very extensive list of SQL functionality cannot be used in the materialized view SQL.

When an incremental refresh occurs, the procedure uses the table system columns `deletexid` and `insertxid`, which are in Redshift not available to normal users (but they are in Postgres), to figure out which rows to delete and which rows to insert, with an additional column per table used by the materialized view being created in the underlying table to store this information. Materialized views using full refresh have no extra columns.

As such, when a `VACUUM` (manual or automatic) of any of the tables used by the materialized view occurs, it forces a full refresh, as `VACUUM` resets the values in the `deletexid` and `insertxid` columns of the vacuumed table and so messes up the book-keeping information held by the materialized view.

When a full refresh occurs, either by the refresh occurring on a materialized view using full refresh, or an incremental refresh running as a full refresh due to `VACUUM`, the procedure, which implements refresh, and here remembering that all SQL inside a procedure is inside a transaction, creates a new table, populates it, renames the old table out of the way, renames the new table to the name of the old table, re-points the view at the new table and drops the old table. A `VACUUM` is not required, since the table is brand new and then populated by a single insert.

When an incremental refresh occurs, the table underlying the materialized view experiences an `insert` and then a `delete`, and as such after refresh requires `VACUUM`.

Materialized views cannot directly be vacuumed. The vacuum command will run, it will report no errors, and it will return the normal `VACUUM` info message, but it will do no work.

The table underlying the materialized view can be vacuumed, but the name of this table is not normally available. It can only be found by inspecting the SQL commands issued by Redshift to implement `CREATE MATERIALIZED VIEW`.

The only other source of vacuum for the underlying table is whatever comes from auto-vacuum. I have not yet produced a white paper on auto-vacuum, so this is has not been investigated, but I and other admin think it runs so infrequently that it is ineffectual.

Note though that when an incremental refresh materialized view is forced to perform a full refresh due to one or more of the tables it uses being vacuumed, the full refresh will produce a new, freshly populated table, which is inherently fully ordered, and so "reset" the underlying table.

Materialized views can be refreshed manually, by issuing the `REFRESH MATERIALIZED VIEW` command, or they can be created such that Redshift takes responsible for automatically issuing refresh, as and when it sees fit.

On a completely idle two node `dc2.large` cluster, a materialized view with full refresh pointing at a one column table with exactly ten full blocks of data typically takes 55 seconds before the first auto-refresh occurs.

When the same setup has a single new row being inserted once per second, the time before the first auto-refresh varied from a low of 54 seconds to a high of 1295 seconds (twenty-one minutes).

The algorithm for refresh is not published, so the factors involved are not known, and it is likely to be undergoing ongoing undocumented, unannounced change.

With regard to all of the above, I hold the following points to be true;

1. Auto-refresh cannot be used, in part because the refresh times are so variable, but mainly because the algorithm is not defined, is likely to be complex, and where real-life is always far more complex than such an algorithm, quite possibly flawed, and, most critically, is sure to be undergoing ongoing undocumented, unannounced change; you cannot knowingly build a correct system when it contains black boxes controlled by third parties.
2. Incremental refresh, because of its weaknesses regarding vacuum, should specifically be avoided.
3. All refresh then must be full refresh, which has to fully regenerate all rows.
4. The impact upon performance of the poor encoding choices made by Redshift is very large.

This leaves then materialized views as full refresh and manual refresh only, which is identical to manually maintaining pre-computing results, except that with materialized views, column encodings cannot be chosen, leading to a very large

loss of performance, and, most critically of all, materialized views only offer on refresh full regeneration of all pre-computed results, where-as manually pre-computing results allows you to perform an incremental insert only, which is often all that is required.

The ETL work for both is the same, an initial statement to create the materialized view or the table holding manually pre-computed results, and then a statement to perform refresh, which is issued by the ETL system at the appropriate moment.

All in all, then, I can actually see *no* use case for materialized views *at all*. As things stand, they are *always* inferior to manually pre-computing results.

Unexpected Findings

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

1. I ran into so many intricate and perplexing problems with `STL_QUERYTEXT` that in the end, once I got to the third round of debugging, I abandoned the effort to perform the necessary pre-processing to the query text being searched for so that it would match the rows recorded in `STL_QUERYTEXT`.

   For now, I am for queries I need to find pre-processing the SQL *before issuing it*, to reduce all contiguous white space to one white space, remove all newlines, etc, so that I avoid most of the bugs in `STL_QUERYTEXT` and can find the queries by their text.

   This doesn't work as a proper solution - what happens if you have strings with whitespace in - but it works for the queries I need to search for in this script. I intend and need now to produce a white paper on `STL_QUERYTEXT` to figure out all the problems and solutions to them, although I have a nasty suspicion right now it may in fact be impossible to find all queries in that table (I suspect you end up with different query texts which are indistinguishable from each other once they're been placed into `STL_QUERYTEXT`).

2. A `VACUUM`, manual or automatic, on any table used by an incremental refresh materialized view, caused the next refresh on that materialized view to be a full refresh.

3. The implementation of materialized views uses the per-table system columns `insertxid` and `deletexid`, access to which is - but by no means at all should or needs to be (Postgres allows access) - forbidden to users.

4. Normal views in `PG_VIEWS` for their SQL store the SQL of the statement which forms the view. This statement must be mated with a `CREATE VIEW` command to create a view.

   Materialized views, however, which are also shown in `PG_VIEWS`, and which have on the face of it cannot be distinguished from normal views, for their SQL store the entire 'CREATE MATERIALIZED VIEW' command.

   Views and materialized views then must be handled differently, are however conflated into the same view, and require painful SQL to distinguish between (using `STV_MV_INFO`, which only lists materialized views, to figure out which rows in `PG_VIEWS` are views and which are materialized views).

5. If an object name is too long (such as a table or view name), I think Redshift used to throw an error. Now Redshift, with an info message, truncates the overly-long name to the maximum supported length.

   I think this is staggeringly vast mistake. If something is wrong, it's infinitely better to alert the user and stop. If you carry on, the user is likely not to be expecting this (since they had no idea there were making a mistake in the first place) and is likely to get caught out later on - which is the more expensive route to discovering error.

Revision History

v1

- Initial release.

v2

- Fixed a spilling errar.

v3

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

v4

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

v5

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

Appendix A : Raw Data Dump

The document processing software is having trouble with this appendix, as it is mainly JSON.

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

Appendix B : AZ64 Encoding

I've not yet produced a white paper investigating the encoding choices Redshift makes (there are three or so different times and places where Redshift makes such choices, but to provide some evidence here, the Results contain a table which has one column for every data type, and a materialized view which is based on this table, and what we see is;

Data Types	Encoding
bool, float4, float8	raw
char, varchar	lzo
everything else	az64

(Note `geometry` and `hllsketch` are both `raw` because that's the only encoding they support.)

There's no reason for any data type to be `raw`. It's simply a lost opportunity to reduce disk space use.

I've made, but not yet published, a white paper which benchmarks encoding and decoding, and the information I provide now comes from that research.

Regarding `char` and `varchar`, there are two general purpose encoders, `lzo` and `zstd`, and a general purpose encoder is indeed the correct choice for arbitrary strings.

Of these two however, `lzo` is fast to encode and slow to decode, which is not what you want, where-as `zstd` is slow to encode but fast to decode, which *is* what you want, and compresses considerably more than `lzo`.

In short, `lzo` should only be used when you have data which is written more often than it is read, which is almost never going to be the case with a sorted column-store database; it is the *wrong choice* as the default, and in fact it is basically obsolete. It has been superseded by `zstd`, which is not too surprising since the basis of `lzo` was developed in 1977 and `zstd` was developed in 2015, and `zstd` should be the encoder used - but it is not.

Finally, we come to `AZ64`, about which I am really quite angry.

Each encoder implements a different compression method and each method works well with and only with data which possesses certain characteristics, and works badly and often fabulously badly with data which does not possess those characteristics.

For example, run length encoding works well on data where there are many repeating values, row after row, and works very badly when this is not the case.

In short, to pick an encoder, *you must know how it works*.

Amazon have *never* published how `az64` works and as such, quite simply, *you can never use it*, because you have no idea if it is going to work well or not given the data you have. It is utter lunacy users are placed in this position.

The only information Amazon ever published was a blog post, written by someone who was either incompetent or deliberately misleading readers.

The post compares `az64` to `raw`, `lzo` and `zstd` and makes various claims about how `az64` is much faster and compresses much better.

I quote;

> - Compared to `RAW` encoding, `AZ64` consumed 60–70% less storage, and was 25–30% faster.
> - Compared to `LZO` encoding, `AZ64` consumed 35% less storage, and was 40% faster.
> - Compared to `ZSTD` encoding, `AZ64` consumed 5–10% less storage, and was 70% faster.

These figures are true for and only for extremely carefully selected data, and so are absolutely misleading to be given as generally true; the problem is that `az64` looks to me in my research to be a runlength-type encoder [2], while the other two are general purpose encoders, and as such, these encoders *cannot be compared* - it's an apples and oranges situation - and to do so, without explaining that `az64` is *not* a general purpose encoder, is either flatly incompetent or culpable.

Indeed, by making this comparison, readers are led to believe `az64` *is* a general purpose encoder, because no one in their right mind would directly compare different types of encoders without informing the reader.

To put it explicitly, consider a runlength type encoder. It works *staggeringly* well when data consists of long runs of the same value, and *staggeringly* badly - making the data larger than the original, in fact - when the data does *not* consist of long runs of the same value.

Now consider a general purpose encoder; it will normally produce about 30% to 40% compression, always, no matter what the data.

You cannot make - as has been done - a blanket statement that the runlength encoder compresses better and faster than the general purpose encoders, because it is simply *not true*. The performance of each encoder depends profoundly on

[2] A Redshift dev informed me `az64` is in fact delta based. I've not yet conducted the testing to prove this, so all I can say is from the testing I have done so far on encodings it looks runlength-like, but that testing did not exclude the possibility of being delta-like.

the data being compressed, and with some data one encoder is better or much better, and with other data, the other encoder is better or much better.

Coming back then to the encoding choices made by Redshift, the use of `az64` as the default for all non-string columns, whether it is runlength or delta, is in my view *absolutely and categorically wrong*, because the data being compressed is an unknown, except that the majority of columns are not going to be sorted, and so will *not* possess the characteristics needed for either of these two methods to work well (and indeed, by not being sorted, will in fact possess the characteristics needed for these two methods to perform badly).

Just to make it clear how flawed the encoding selection algorithm is, note both runlength and delta encoders should *never* be used on random data - it's the worst kind of data for them both - but Redshift uses `az64` as described above, for all non-string columns, *on interleaved sorted tables*.

Interleaved sorting makes the values in columns essentially random, and `az64` I see from my own testing as expected in this situation produces *zero* compression. This isn't just the wrong choice, it's dribbling madness. No one is thinking about this. No one is checking the outcome of these choices. These are not well informed, carefully selected choices made by an expert.

I may be wrong, but what I suspect has happened is that where `az64` is proprietary, an in-house AWS encoder, internal politics have pushed for it to be used, and so the wrong technical choice has been taken for political reasons.

We then come to see a later blog post, where we see that;

> Five months after launch, the `AZ64` encoding has become the fourth most popular encoding option in Amazon Redshift with millions of columns.

I suspect this is true, but it's because `az64` has been made the by far the most common default encoding choice made by Redshift, and because people have been misled by the blog post comparing `az64` to general purpose encoders, and so all of the people using it, either by default or by their own choice, are very likely using the wrong encoder for their data, quite likely are getting zero compression, and this is contributing to making performance poor and erratic, and this is contributing to people leaving for Snowflake.

Coming back then from this journey into encoding choices, I argue then that the encoding choices made by Redshift are no good, and these choices are used for materialized views, and *this alone* is enough to mandate that materialized views are never used, because encoding choices are critical for performance.

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.