MERGE

Max Ganz II @ Redshift Research Project

27th April 2023

Abstract

The `MERGE` command is implemented as a wrapper around existing SQL commands, calling `CREATE TEMP TABLE`, `UPDATE` and `INSERT` or `DELETE`. The SQL standard specifies the `MERGE` command will either match (and so update) or not match (and so insert) each source row. When manually issuing an `UPDATE` followed by an `INSERT`, it *is* possible for a row to match and not match, when the change performed by the update causes a row which did match to no longer match. This problem is solved by using a temp table, to store all rows which are to be inserted, prior to running the update. However, in the case when the data is such this problem does not occur, the temp table is not necessary, and is pure overhead. As such, `MERGE` is a kind of worst-case implementation; it has to always work correctly, so it always uses a temp table, even when it is not necessary. Finally, `MERGE` mandates the use of and only of a table for merge source rows, and I can see no reason for this, as none of the wrapped commands require it, and all accept a sub-query or view.

Contents

Introduction

As of April 2023, AWS have introduced into Redshift a new SQL command, `MERGE` (which originated in the SQL 2003 standard and is pretty much universal across relational databases).

This new SQL command provides an `INSERT`, combined with either a `DELETE` or an `UPDATE`.

The way it works is that you have your target table, which is the table you wish to modify, and a source table, which contains the rows you wish to use to modify the target table.

The `MERGE` command iterates over the source table, and for each row, if that row exists in the target table, that row in the target table will be either updated or deleted, and if that row does not exist in the target table, it will be inserted.

The choice of whether a row is updated or deleted is not on a per-row basis, but on a per-merge command basis; all rows which match, for a given merge command, will either be updated, or deleted.

All rows which do not match are always inserted; this is not optional (if it was, you'd just use a normal update or delete).

Finally, the source table actually has to be a table; not a sub-query or a CTE. You have to actually create and populate a table. Often this will be a staging table, so it's not so bad, but in the cases when it is not, that's an overhead.

Now, from a syntactical point of view, `MERGE` is a very nice improvement, over writing separate `INSERT` and `UPDATE` or `DELETE` commands.

From a performance point of view, the question obviously is whether `MERGE` is the same as, or better than, or worse than, separate `INSERT` and `UPDATE` or `DELETE` commands, and if there are any complications or gotchas to be aware of.

This question in turn in fact requires us to define what "better" and "worse" actually mean.

To my eye, there are three considerations.

Firstly, the number of new and modified blocks.

Does a `MERGE` produce the same, or more, or fewer, new and modified blocks, as the separate `INSERT` and `UPDATE` or `DELETE` queries, which would perform the same work?

Secondly, how does the work performed by the `MERGE` query compared to the work performed by separate `INSERT` and `UPDATE` or `DELETE` queries?

At the very highest level, we can look to see whether or not `MERGE` is a single query rather than the two for an `INSERT` and `UPDATE` or `DELETE`, but more meaningfully, we can look at the step plans for a `MERGE`, and the step plans for the equivalent `INSERT` and `UPDATE` or `DELETE`, and see what work is actually being done under the hood.

Thirdly, a more general question, which is what's actually going on in general under the hood, and are there are issues we need to be aware of and look out for?

Investigation

To begin with, I create a new two node `dc2.large` in `us-east-1` and configure the cluster like so;

```
set enable_result_cache_for_session to off;
set analyze_threshold_percent to 0;
set mv_enable_aqmv_for_session to false;
```

I then create two test tables, `target_table` and `source_table` both with one column and one row, like so;

```
create table target_table
(
  column_1  int8  not null encode raw distkey
)
diststyle key
compound sortkey( column_1 );

insert into
  target_table( column_1 )
values
  ( 1 );

vacuum full target_table to 100 percent;
analyze target_table;

create table source_table
(
  column_1  int8  not null encode raw distkey
)
diststyle key
compound sortkey( column_1 );

insert into
  source_table( column_1 )
values
  ( 2 );
```

```
vacuum full source_table to 100 percent;
analyze source_table;
```

So now we have `target_table` with a single row with the value `1`, and `source_table` with a single row and the value `2`.

`MERGE` with `UPDATE`

To begin with, to start getting an idea of what Redshift is going to do, I issue an `EXPLAIN` on the following `MERGE`, where I expect a single row, with the value `2`, to be inserted into `target_table`.

```
explain
merge into target_table
using source_table on target_table.column_1 = source_table.column_1
when matched then update set column_1 = source_table.column_1
when not matched then insert values ( source_table.column_1 );
```

What I get is this;

```
                                QUERY PLAN
---------------------------------------------------------------------------
 XN Merge Join DS_DIST_NONE  (cost=0.00..0.04 rows=1 width=8)
   Merge Cond: ("outer".column_1 = "inner".column_1)
   ->  XN Seq Scan on source_table  (cost=0.00..0.01 rows=1 width=8)
   ->  XN Seq Scan on target_table  (cost=0.00..0.01 rows=1 width=8)

 XN Hash Join DS_DIST_NONE  (cost=0.01..0.04 rows=1 width=14)
   Hash Cond: ("outer".column_1 = "inner".column_1)
   ->  XN Seq Scan on target_table  (cost=0.00..0.01 rows=1 width=14)
   ->  XN Hash  (cost=0.01..0.01 rows=1 width=8)
         ->  XN Seq Scan on source_table  (cost=0.00..0.01 rows=1 width=8)

 XN Seq Scan on merge_tt_5f9f1e92e6878  (cost=0.00..0.01 rows=1 width=8)
(12 rows)
```

Now that's a pretty unusual query plan. It looks like three separate queries.

Looking at the query history, what I see is quite surprising and wholly novel.

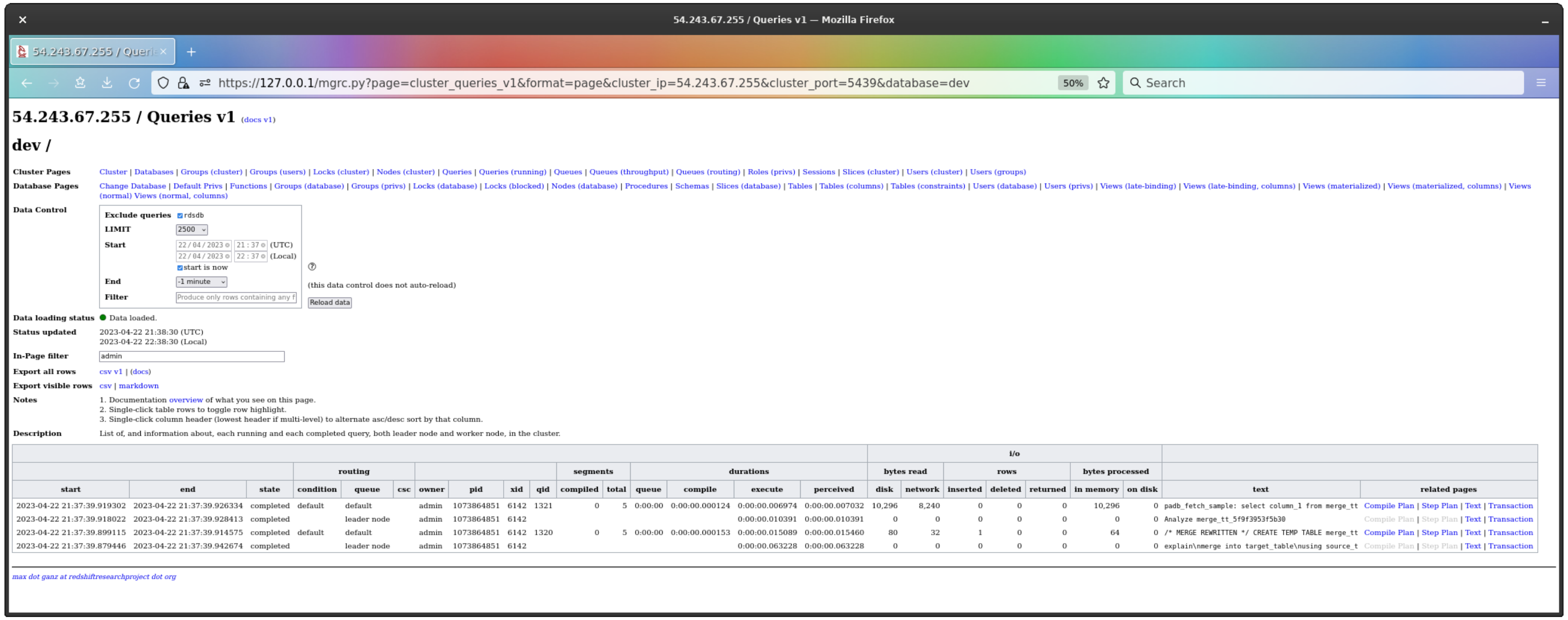


Figure 1: Query Text History Overview

Now, the query history page is pretty wide - there's lots of columns - so the first screenshot is the whole page (which requires 50% zoom), but the second is at 100% zoom and moved over to the right so we can see the column showing the first 48 characters of query text.

	i/o								
	bytes read		rows			bytes processed			
ceived	disk	network	inserted	deleted	returned	in memory	on disk	text	related pages
0.007032	10,296	8,240	0	0	0	10,296	0	`padb_fetch_sample: select column_1 from merge_tt`	Compile Plan | Step Plan | Text | Transaction
0.010391	0	0	0	0	0	0	0	`Analyze merge_tt_5f9f3953f5b30`	Compile Plan | Step Plan | Text | Transaction
0.015460	80	32	1	0	0	64	0	`/* MERGE REWRITTEN */ CREATE TEMP TABLE merge_tt`	Compile Plan | Step Plan | Text | Transaction
0.063228	0	0	0	0	0	0	0	`explain\nmerge into target_table\nusing source_t`	Compile Plan | Step Plan | Text | Transaction

Figure 2: Query Text History Closeup

The order of queries is oldest at the bottom. The `EXPLAIN` command is highlighted, and is at the bottom, and we can its text - it begins with "explain\n".

To my considerable surprise, the `EXPLAIN` has *created a temp table* and analyzed it (the analyze leads to the final command, the `padb_fetch_sample`).

This is new - I've never seen an `EXPLAIN` do actual work before, and this is interesting, because it means if you ran this `EXPLAIN` on a serious table, a Big Data table, you might find its doing a *lot* of work, and temp table persist until a session ends, so every `EXPLAIN` you issue is going to consume additional disk space (until your session ends).

The next question then is what is in the temp table.

To find this out, we look at the text of the `CREATE TEMP TABLE` command.

Figure 3: Query Text

The text has been issued as a single line of text with no newlines, so I clearly need to add an option to format the text - as it is, I have copy-and-pasted the text, and manually formatted the text, below;

```
/* MERGE REWRITTEN */
CREATE TEMP TABLE
  merge_tt_5f9f2b11d9736
AS
  (
    SELECT
```

```
    "public"."source_table"."column_1" AS "column_1"
  FROM
    "public"."source_table"
  WHERE
    NOT EXISTS
    (
      SELECT
        CAST(1 AS INT4)
      FROM
        "public"."target_table"
      WHERE
        "public"."target_table"."column_1" = "public"."source_table"."column_1"
    )
)
```

Wowsers. Three queries already - the create temp itself, and two sub-queries.

Looking at the code, what the temp table consists of is all rows which are in `source_table` but *not* in `target_table` - in other words, all the rows which would be inserted by the merge.

So, indeed - if this was done on tables of any size, that could perfectly well be a lot of rows.

To my surprise again however, the temp tables being produced appear to be silently deleted immediately after the `EXPLAIN`. As far as I can tell, they are *not* extant after the query, but I can see no `DROP TABLE`, and I'm still in the same session.

The temp table has no specified encodings, so Redshift is using its default encodings, which are really no good at all, and also `auto` for distribution and sorting. This is an inefficient table. At least where it's a temp table, it does not participate in k-safety, which will make it faster, and where it's a single insert into a new table, the rows will be sorted (but since it's an `auto` table, and the table is new, it will start as `ALL`, then if enough data is inserted become `EVEN` (I don't know when the redistribution happens - I never use `AUTO` - I suspect it might be immediately after the query completes, so the table could then having been created, in effect be created again, as a new copy has to be made with the new distribution style), and both `ALL` and `EVEN` preclude merge joins. I've no idea what sort key `AUTO` will choose for a brand new table - unsorted, perhaps? Redshift in principle can change this, but I believe this takes some time, Redshift has to build up some history of queries on the table, so I do not expect it to change from whatever the initial default is.)

Let's try now actually executing the `MERGE`, and let's see what we get.

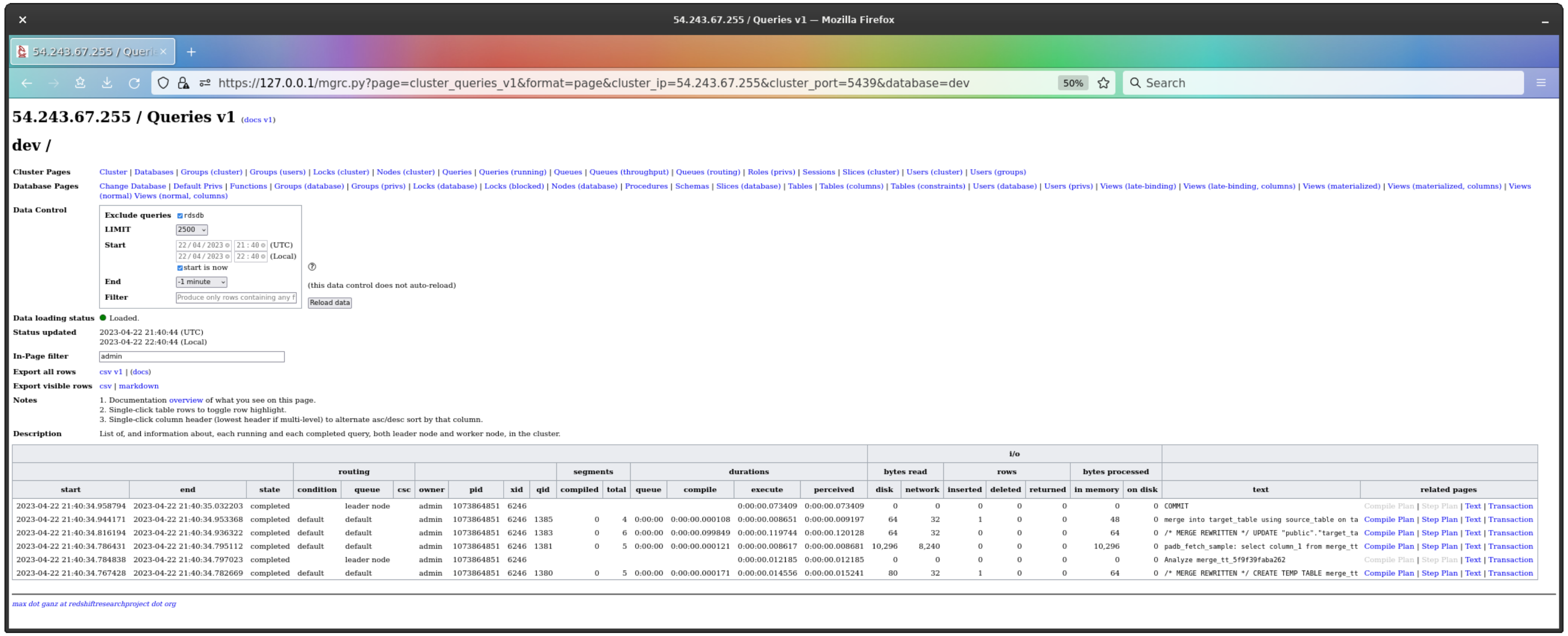
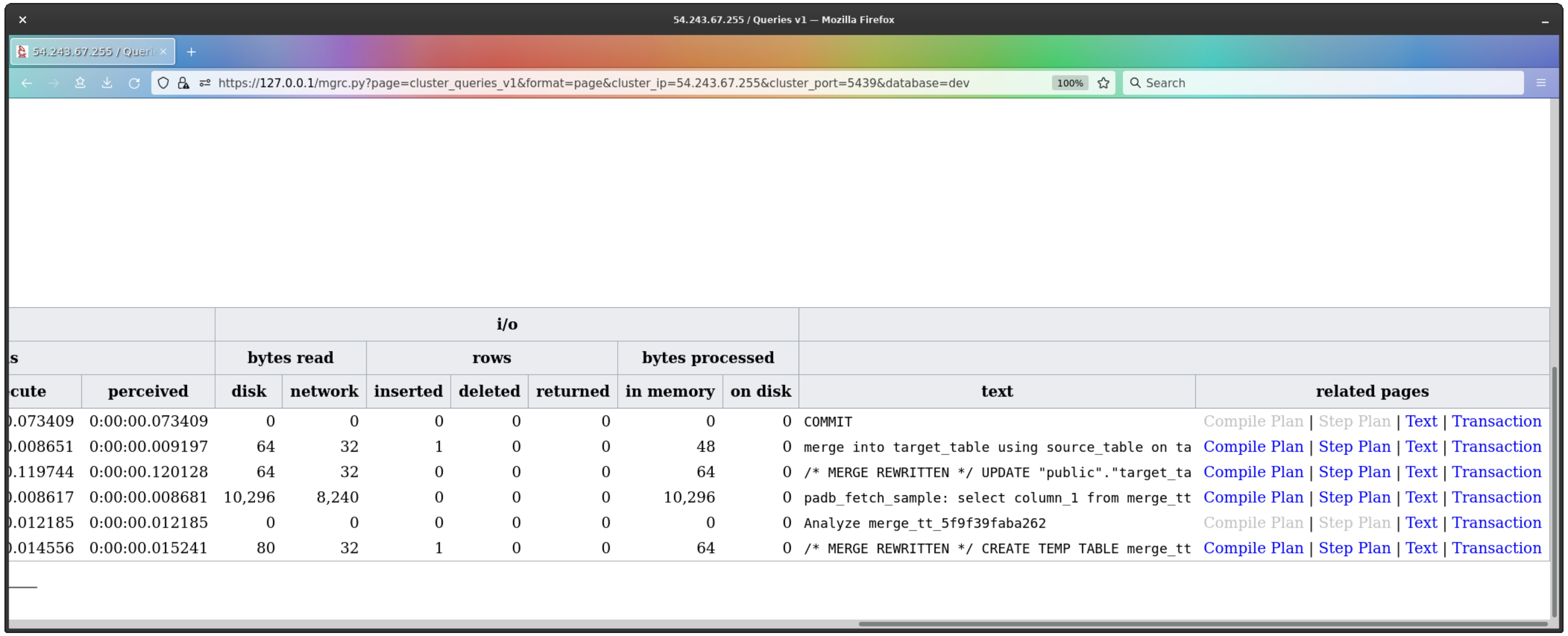


Figure 4: Query Text History Overview

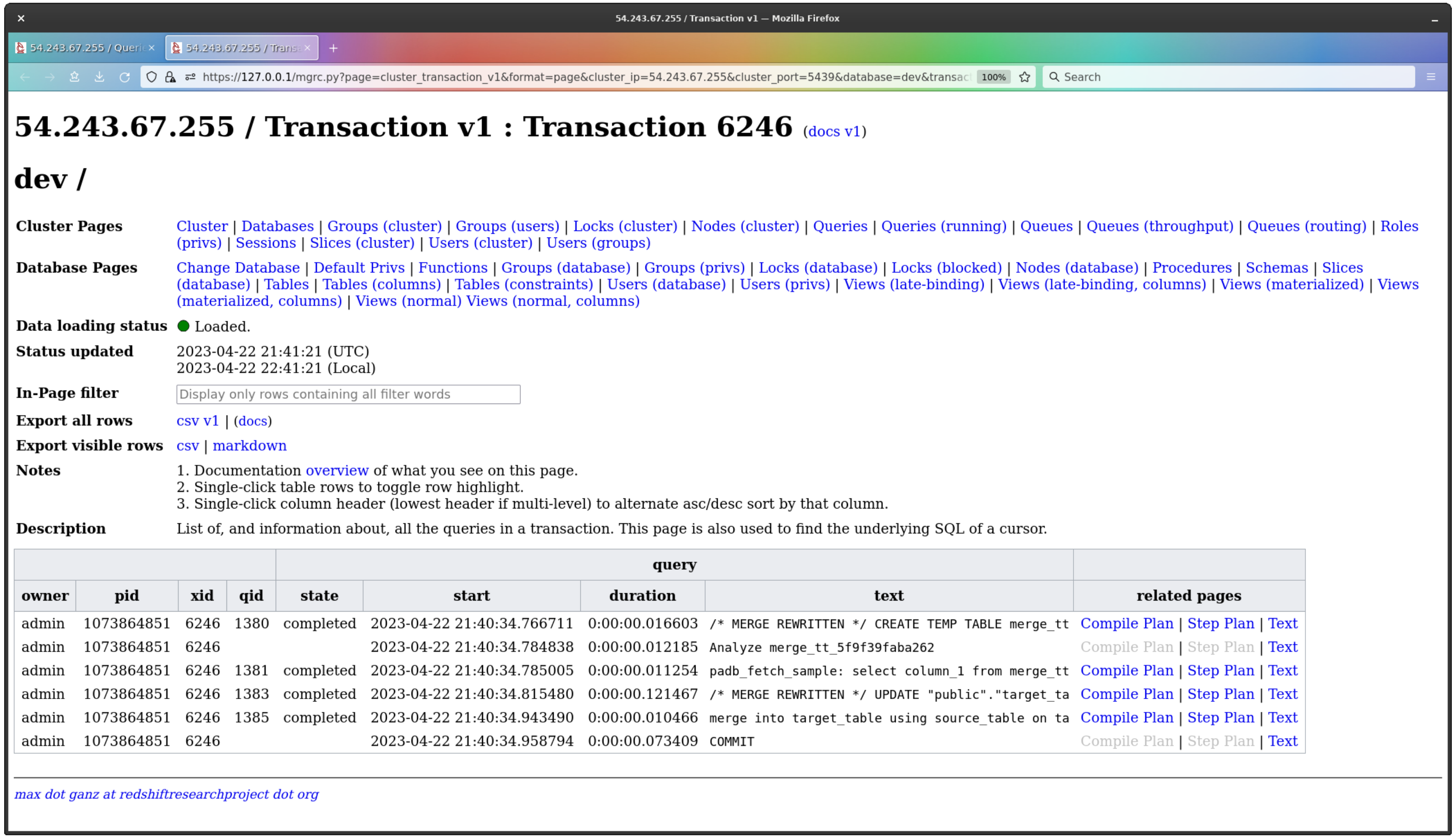


		i/o									
s		bytes read		rows			bytes processed				
cute	perceived	disk	network	inserted	deleted	returned	in memory	on disk	text	related pages
).073409	0:00:00.073409	0	0	0	0	0	0	0	COMMIT	Compile Plan | Step Plan | Text | Transaction
).008651	0:00:00.009197	64	32	1	0	0	48	0	merge into target_table using source_table on ta	Compile Plan | Step Plan | Text | Transaction
).119744	0:00:00.120128	64	32	0	0	0	64	0	/* MERGE REWRITTEN */ UPDATE "public"."target_ta	Compile Plan | Step Plan | Text | Transaction
).008617	0:00:00.008681	10,296	8,240	0	0	0	10,296	0	padb_fetch_sample: select column_1 from merge_tt	Compile Plan | Step Plan | Text | Transaction
).012185	0:00:00.012185	0	0	0	0	0	0	0	Analyze merge_tt_5f9f39faba262	Compile Plan | Step Plan | Text | Transaction
).014556	0:00:00.015241	80	32	1	0	0	64	0	/* MERGE REWRITTEN */ CREATE TEMP TABLE merge_tt	Compile Plan | Step Plan | Text | Transaction

Figure 5: Query Text Closeup

Well well well. Isn’t this interesting.

To make it easier to see, and to show this is a multi-query transaction, I’ve brought up the transaction which holds the merge.



54.243.67.255 / Transaction v1 — Mozilla Firefox

https://127.0.0.1/mgrc.py?page=cluster_transaction_v1&format=page&cluster_ip=54.243.67.255&cluster_port=5439&database=dev&transac

54.243.67.255 / Transaction v1 : Transaction 6246 (docs v1)

dev /

Cluster Pages Cluster | Databases | Groups (cluster) | Groups (users) | Locks (cluster) | Nodes (cluster) | Queries | Queries (running) | Queues | Queues (throughput) | Queues (routing) | Roles (privs) | Sessions | Slices (cluster) | Users (cluster) | Users (groups)

Database Pages Change Database | Default Privs | Functions | Groups (database) | Groups (privs) | Locks (database) | Locks (blocked) | Nodes (database) | Procedures | Schemas | Slices (database) | Tables | Tables (columns) | Tables (constraints) | Users (database) | Users (privs) | Views (late-binding) | Views (late-binding, columns) | Views (materialized) | Views (materialized, columns) | Views (normal) Views (normal, columns)

Data loading status Loaded.

Status updated 2023-04-22 21:41:21 (UTC)
2023-04-22 22:41:21 (Local)

In-Page filter Display only rows containing all filter words

Export all rows csv v1 | (docs)

Export visible rows csv | markdown

Notes
1. Documentation overview of what you see on this page.
2. Single-click table rows to toggle row highlight.
3. Single-click column header (lowest header if multi-level) to alternate asc/desc sort by that column.

Description List of, and information about, all the queries in a transaction. This page is also used to find the underlying SQL of a cursor.

				query				
owner	pid	xid	qid	state	start	duration	text	related pages
admin	1073864851	6246	1380	completed	2023-04-22 21:40:34.766711	0:00:00.016603	/* MERGE REWRITTEN */ CREATE TEMP TABLE merge_tt	Compile Plan | Step Plan | Text
admin	1073864851	6246			2023-04-22 21:40:34.784838	0:00:00.012185	Analyze merge_tt_5f9f39faba262	Compile Plan | Step Plan | Text
admin	1073864851	6246	1381	completed	2023-04-22 21:40:34.785005	0:00:00.011254	padb_fetch_sample: select column_1 from merge_tt	Compile Plan | Step Plan | Text
admin	1073864851	6246	1383	completed	2023-04-22 21:40:34.815480	0:00:00.121467	/* MERGE REWRITTEN */ UPDATE "public"."target_ta	Compile Plan | Step Plan | Text
admin	1073864851	6246	1385	completed	2023-04-22 21:40:34.943490	0:00:00.010466	merge into target_table using source_table on ta	Compile Plan | Step Plan | Text
admin	1073864851	6246			2023-04-22 21:40:34.958794	0:00:00.073409	COMMIT	Compile Plan | Step Plan | Text

max dot ganz at redshiftresearchproject dot org

Figure 6: Transaction

The transaction has queries listed from top to bottom as first to last in the transaction.

What we see are;

1. `CREATE TEMP TABLE` (all rows which would be inserted).
2. `ANALYZE` of the temp table.
3. The `padb_fetch_sample` for the analyze.
4. An actual explicit `UPDATE` command.
5. A `MERGE` command.
6. The final commit.

Now, with regard to the temp table, the work being done in the query is being done because `SELECT` in Redshift does not support the `ALL` argument to `EXCEPT`.

If it did, you'd produce the set of rows by excepting all from `source_table` all rows in `target_table`, which is sweet, simple and easy to read, but without `ALL`, you get one row only of each distinct row, regardless of how many of that row exist in `source_table`, which is not what you want.

In the first version of this document, I did not understand why the temp table was in use. After publication, a Slack user by the name of mauro explained it to me, and so I can now explain it to you.

The reason for the temp table is that the specification of the `MERGE` command mandates each source row either matches or does not match, but never both; an `UPDATE` and `INSERT` pair *without* a temp table *can* allow a source row to match *and* not match - this happens when the source row matches, but the update command that is then as such issued converts the row into a row which no longer matches, and then of course the `INSERT`, reading the source table a second time, will go right ahead and insert.

In merges where the `UPDATE` does *not* cause rows to match, the temp table is indeed unnecessary, so the implementation of `MERGE` in Redshift is a kind of worst-case implementation - it has to be like that, to actually meet the specification of `MERGE`, but whenever the merge you're issuing would not cause matched rows to no longer match, the temp table is not needed, and then it's pure overhead.

So now we a solid overview of what's going on; `MERGE` is not a new command as such, but a macro.

We can now though look at the query texts, and the step plans, of the queries, to see what they're doing.

The `CREATE TEMP TABLE` turns out not surprisingly to be identical to that emitted by the `EXPLAIN` command.

The `ANALYZE` is just doing what `ANALYZE` does, so that's not of particular interest to us.

The `UPDATE` is interesting, and we see this, which again is a single line of text, which I have taken out and formatted for easy reading;

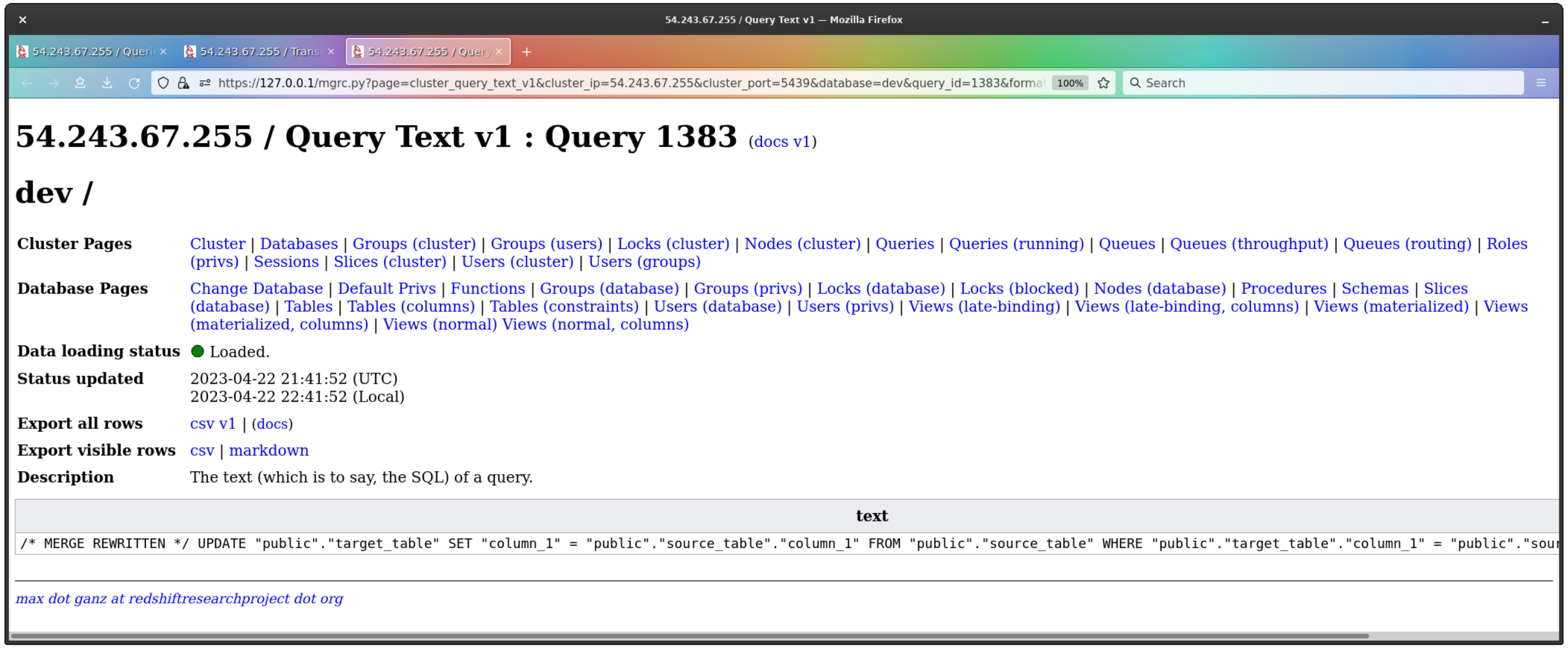


Figure 7: UPDATE Query Text

```
/* MERGE REWRITTEN */
UPDATE
  "public"."target_table"
SET
  "column_1" = "public"."source_table"."column_1"
FROM
  "public"."source_table"
WHERE
  "public"."target_table"."column_1" = "public"."source_table"."column_1"
```

This does what we'd expect; it updates every row in `target_table` which can be found in `source_table`.

Now we come to the mysterious MERGE.

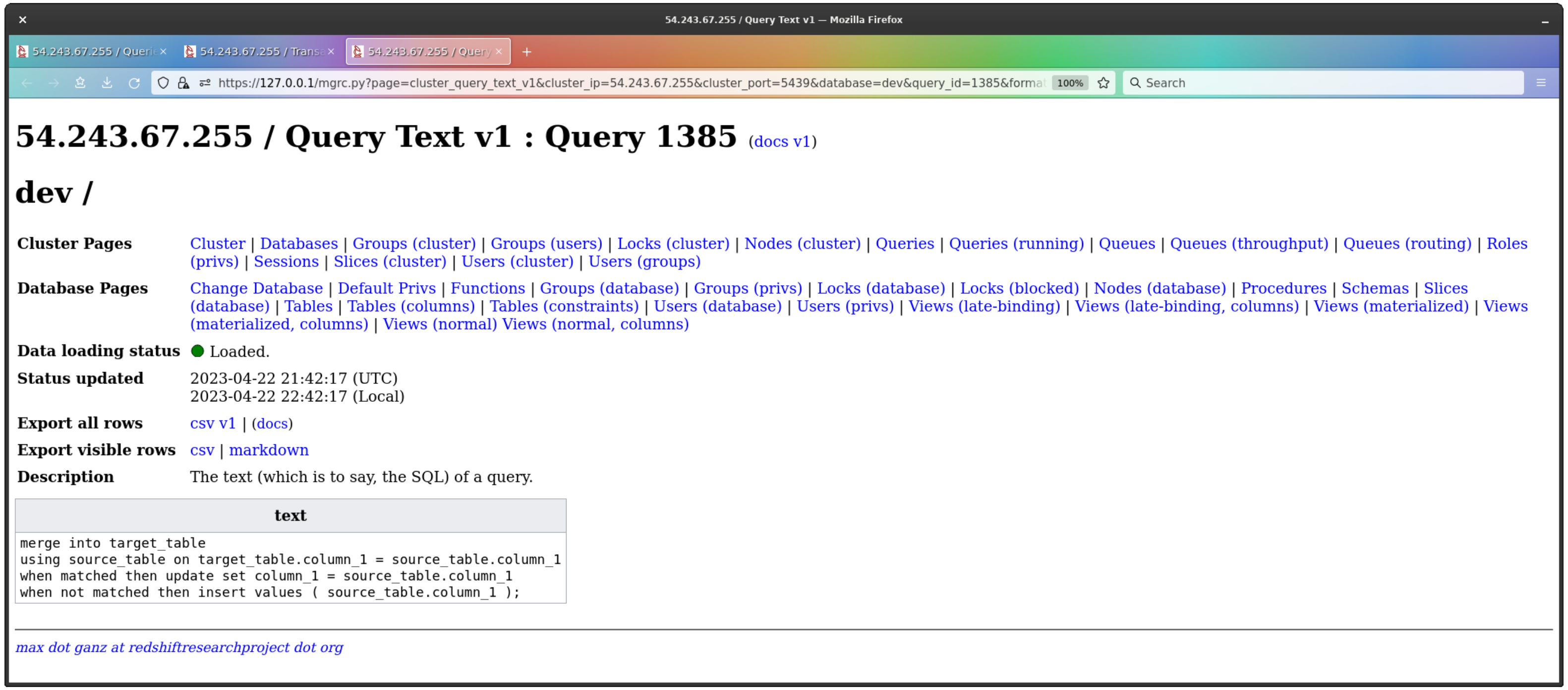
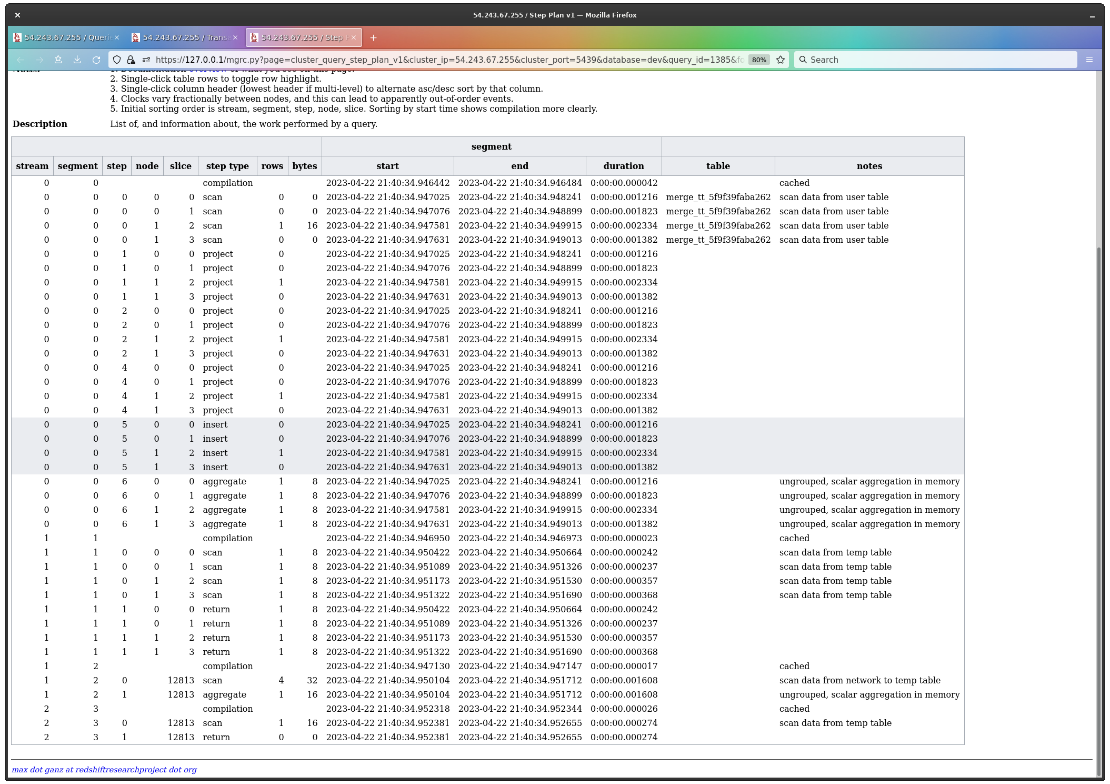


Figure 8: MERGE Query Text

As we can see, the text is exactly that of the original MERGE command, which tells us nothing.

However, we know we're missing the insert, so my guess is what's really going on here is the insert - and we can check by looking at the step plan.



54.243.67.255 / Step Plan v1 — Mozilla Firefox

https://127.0.0.1/mgrc.py?page=cluster_query_step_plan_v1&cluster_ip=54.243.67.255&cluster_port=5439&database=dev&query_id=1385&f

2. Single-click table rows to toggle row highlight.
3. Single-click column header (lowest header if multi-level) to alternate asc/desc sort by that column.
4. Clocks vary fractionally between nodes, and this can lead to apparently out-of-order events.
5. Initial sorting order is stream, segment, step, node, slice. Sorting by start time shows compilation more clearly.

Description List of, and information about, the work performed by a query.

								segment				
stream	**segment**	**step**	**node**	**slice**	**step type**	**rows**	**bytes**	**start**	**end**	**duration**	**table**	**notes**
0	0				compilation			2023-04-22 21:40:34.946442	2023-04-22 21:40:34.946484	0:00:00.000042		cached
0	0	0	0	0	scan	0	0	2023-04-22 21:40:34.947025	2023-04-22 21:40:34.948241	0:00:00.001216	merge_tt_5f9f39faba262	scan data from user table
0	0	0	0	1	scan	0	0	2023-04-22 21:40:34.947076	2023-04-22 21:40:34.948899	0:00:00.001823	merge_tt_5f9f39faba262	scan data from user table
0	0	0	1	2	scan	1	16	2023-04-22 21:40:34.947581	2023-04-22 21:40:34.949915	0:00:00.002334	merge_tt_5f9f39faba262	scan data from user table
0	0	0	1	3	scan	0	0	2023-04-22 21:40:34.947631	2023-04-22 21:40:34.949013	0:00:00.001382	merge_tt_5f9f39faba262	scan data from user table
0	0	1	0	0	project	0		2023-04-22 21:40:34.947025	2023-04-22 21:40:34.948241	0:00:00.001216		
0	0	1	0	1	project	0		2023-04-22 21:40:34.947076	2023-04-22 21:40:34.948899	0:00:00.001823		
0	0	1	1	2	project	1		2023-04-22 21:40:34.947581	2023-04-22 21:40:34.949915	0:00:00.002334		
0	0	1	1	3	project	0		2023-04-22 21:40:34.947631	2023-04-22 21:40:34.949013	0:00:00.001382		
0	0	2	0	0	project	0		2023-04-22 21:40:34.947025	2023-04-22 21:40:34.948241	0:00:00.001216		
0	0	2	0	1	project	0		2023-04-22 21:40:34.947076	2023-04-22 21:40:34.948899	0:00:00.001823		
0	0	2	1	2	project	1		2023-04-22 21:40:34.947581	2023-04-22 21:40:34.949915	0:00:00.002334		
0	0	2	1	3	project	0		2023-04-22 21:40:34.947631	2023-04-22 21:40:34.949013	0:00:00.001382		
0	0	4	0	0	project	0		2023-04-22 21:40:34.947025	2023-04-22 21:40:34.948241	0:00:00.001216		
0	0	4	0	1	project	0		2023-04-22 21:40:34.947076	2023-04-22 21:40:34.948899	0:00:00.001823		
0	0	4	1	2	project	1		2023-04-22 21:40:34.947581	2023-04-22 21:40:34.949915	0:00:00.002334		
0	0	4	1	3	project	0		2023-04-22 21:40:34.947631	2023-04-22 21:40:34.949013	0:00:00.001382		
0	0	5	0	0	insert	0		2023-04-22 21:40:34.947025	2023-04-22 21:40:34.948241	0:00:00.001216		
0	0	5	0	1	insert	0		2023-04-22 21:40:34.947076	2023-04-22 21:40:34.948899	0:00:00.001823		
0	0	5	1	2	insert	1		2023-04-22 21:40:34.947581	2023-04-22 21:40:34.949915	0:00:00.002334		
0	0	5	1	3	insert	0		2023-04-22 21:40:34.947631	2023-04-22 21:40:34.949013	0:00:00.001382		
0	0	6	0	0	aggregate	1	8	2023-04-22 21:40:34.947025	2023-04-22 21:40:34.948241	0:00:00.001216		ungrouped, scalar aggregation in memory
0	0	6	0	1	aggregate	1	8	2023-04-22 21:40:34.947076	2023-04-22 21:40:34.948899	0:00:00.001823		ungrouped, scalar aggregation in memory
0	0	6	1	2	aggregate	1	8	2023-04-22 21:40:34.947581	2023-04-22 21:40:34.949915	0:00:00.002334		ungrouped, scalar aggregation in memory
0	0	6	1	3	aggregate	1	8	2023-04-22 21:40:34.947631	2023-04-22 21:40:34.949013	0:00:00.001382		ungrouped, scalar aggregation in memory
1	1				compilation			2023-04-22 21:40:34.946950	2023-04-22 21:40:34.946973	0:00:00.000023		cached
1	1	0	0	0	scan	1	8	2023-04-22 21:40:34.950422	2023-04-22 21:40:34.950664	0:00:00.000242		scan data from temp table
1	1	0	0	1	scan	1	8	2023-04-22 21:40:34.951089	2023-04-22 21:40:34.951326	0:00:00.000237		scan data from temp table
1	1	0	1	2	scan	1	8	2023-04-22 21:40:34.951173	2023-04-22 21:40:34.951530	0:00:00.000357		scan data from temp table
1	1	0	1	3	scan	1	8	2023-04-22 21:40:34.951322	2023-04-22 21:40:34.951690	0:00:00.000368		scan data from temp table
1	1	1	0	0	return	1	8	2023-04-22 21:40:34.950422	2023-04-22 21:40:34.950664	0:00:00.000242		
1	1	1	0	1	return	1	8	2023-04-22 21:40:34.951089	2023-04-22 21:40:34.951326	0:00:00.000237		
1	1	1	1	2	return	1	8	2023-04-22 21:40:34.951173	2023-04-22 21:40:34.951530	0:00:00.000357		
1	1	1	1	3	return	1	8	2023-04-22 21:40:34.951322	2023-04-22 21:40:34.951690	0:00:00.000368		
1	2				compilation			2023-04-22 21:40:34.947130	2023-04-22 21:40:34.947147	0:00:00.000017		cached
1	2	0		12813	scan	4	32	2023-04-22 21:40:34.950104	2023-04-22 21:40:34.951712	0:00:00.001608		scan data from network to temp table
1	2	1		12813	aggregate	1	16	2023-04-22 21:40:34.950104	2023-04-22 21:40:34.951712	0:00:00.001608		ungrouped, scalar aggregation in memory
2	3				compilation			2023-04-22 21:40:34.952318	2023-04-22 21:40:34.952344	0:00:00.000026		cached
2	3	0		12813	scan	1	16	2023-04-22 21:40:34.952381	2023-04-22 21:40:34.952655	0:00:00.000274		scan data from temp table
2	3	1		12813	return	0	0	2023-04-22 21:40:34.952381	2023-04-22 21:40:34.952655	0:00:00.000274		

max dot ganz at redshiftresearchproject dot org

Figure 9: MERGE Step Plan

Bingo.

The four highlighted lines are the insert step (one line per slice).

The actual SQL for this query must be that of an `INSERT`, but my guess is it is being replaced by the `MERGE` SQL by Redshift, perhaps so people looking at the query history can actually see the command they issued.

What can I say? on one hand, I do want to see the actual commands being issued, on the other, I also want to see what's actually going on. System table design needs to be improved, rather than having stuff shoe-horned in.

MERGE with DELETE

So, we start from scratch; we drop the tables we made, and re-create them, exactly as we did for merge with update.

We then issue the following command;

```
merge into target_table
using source_table on target_table.column_1 = source_table.column_1
when matched then delete
when not matched then insert values ( source_table.column_1 );
```

As before, the screenshot of the query history for the MERGE, and the close up of the query texts;

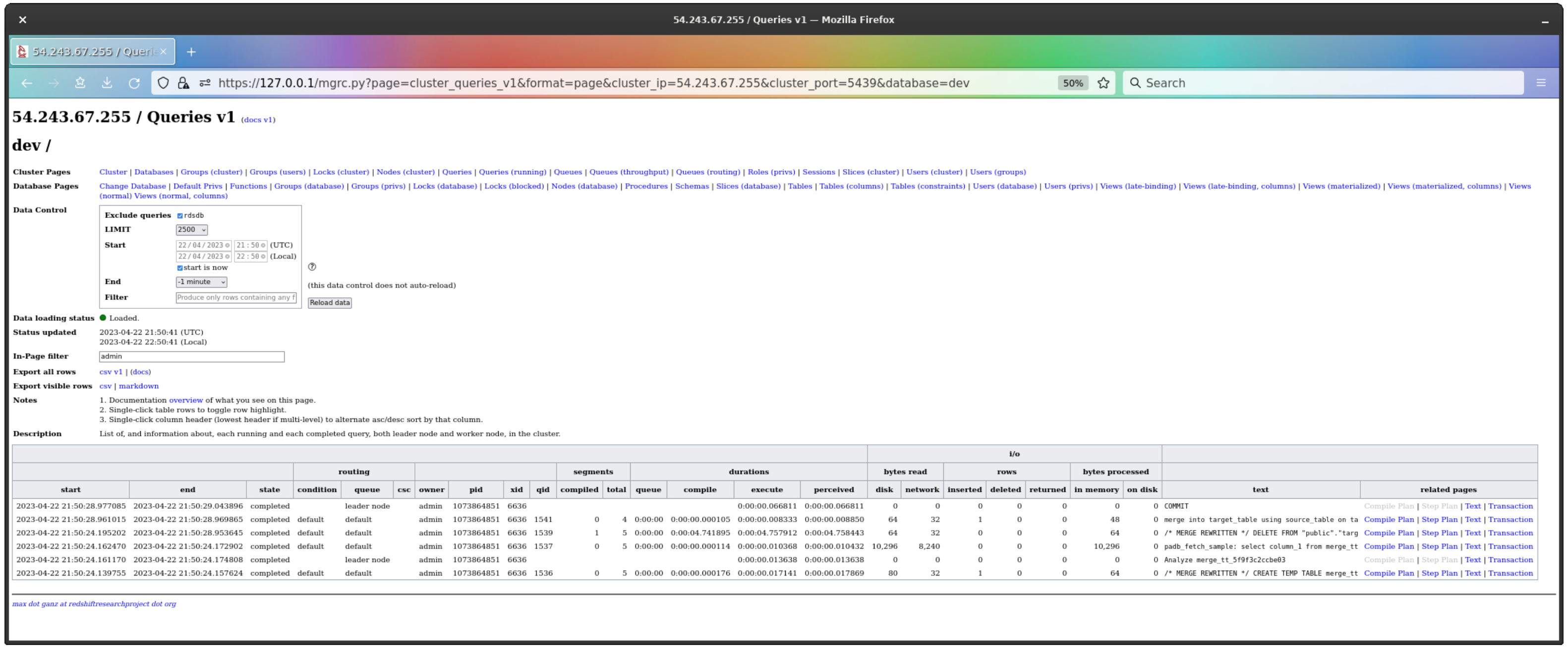
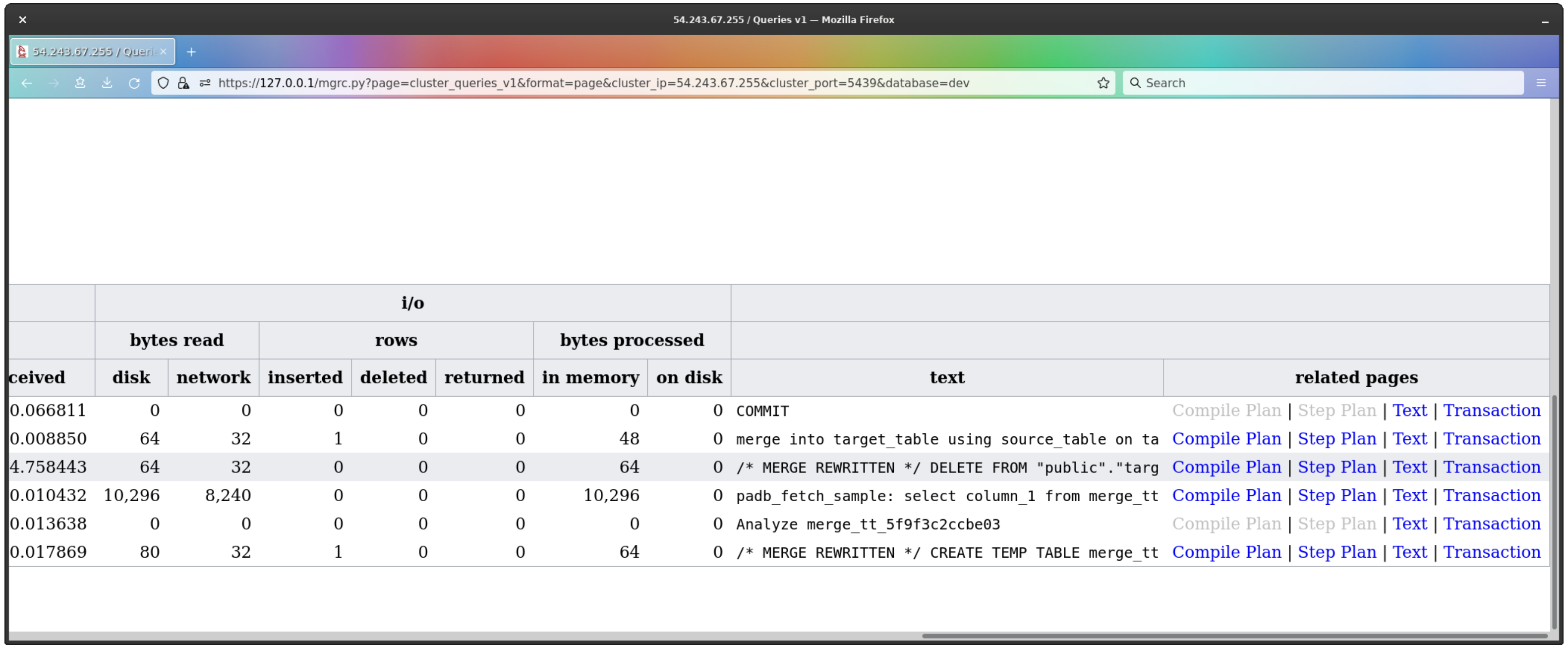


Figure 10: Query Text History Overview



	i/o								
	bytes read		rows			bytes processed			
ceived	disk	network	inserted	deleted	returned	in memory	on disk	text	related pages
0.066811	0	0	0	0	0	0	0	COMMIT	Compile Plan | Step Plan | Text | Transaction
0.008850	64	32	1	0	0	48	0	merge into target_table using source_table on ta	Compile Plan | Step Plan | Text | Transaction
4.758443	64	32	0	0	0	64	0	/* MERGE REWRITTEN */ DELETE FROM "public"."targ	Compile Plan | Step Plan | Text | Transaction
0.010432	10,296	8,240	0	0	0	10,296	0	padb_fetch_sample: select column_1 from merge_tt	Compile Plan | Step Plan | Text | Transaction
0.013638	0	0	0	0	0	0	0	Analyze merge_tt_5f9f3c2ccbe03	Compile Plan | Step Plan | Text | Transaction
0.017869	80	32	1	0	0	64	0	/* MERGE REWRITTEN */ CREATE TEMP TABLE merge_tt	Compile Plan | Step Plan | Text | Transaction

Figure 11: Query Text Closeup

The text of the `CREATE TEMP TABLE` command is unchanged, which populates the temp table with all the rows which would be inserted, and we can see the `UPDATE` has been replaced by `DELETE` (the highlighted line).

Finally, we have the text of the `DELETE`;

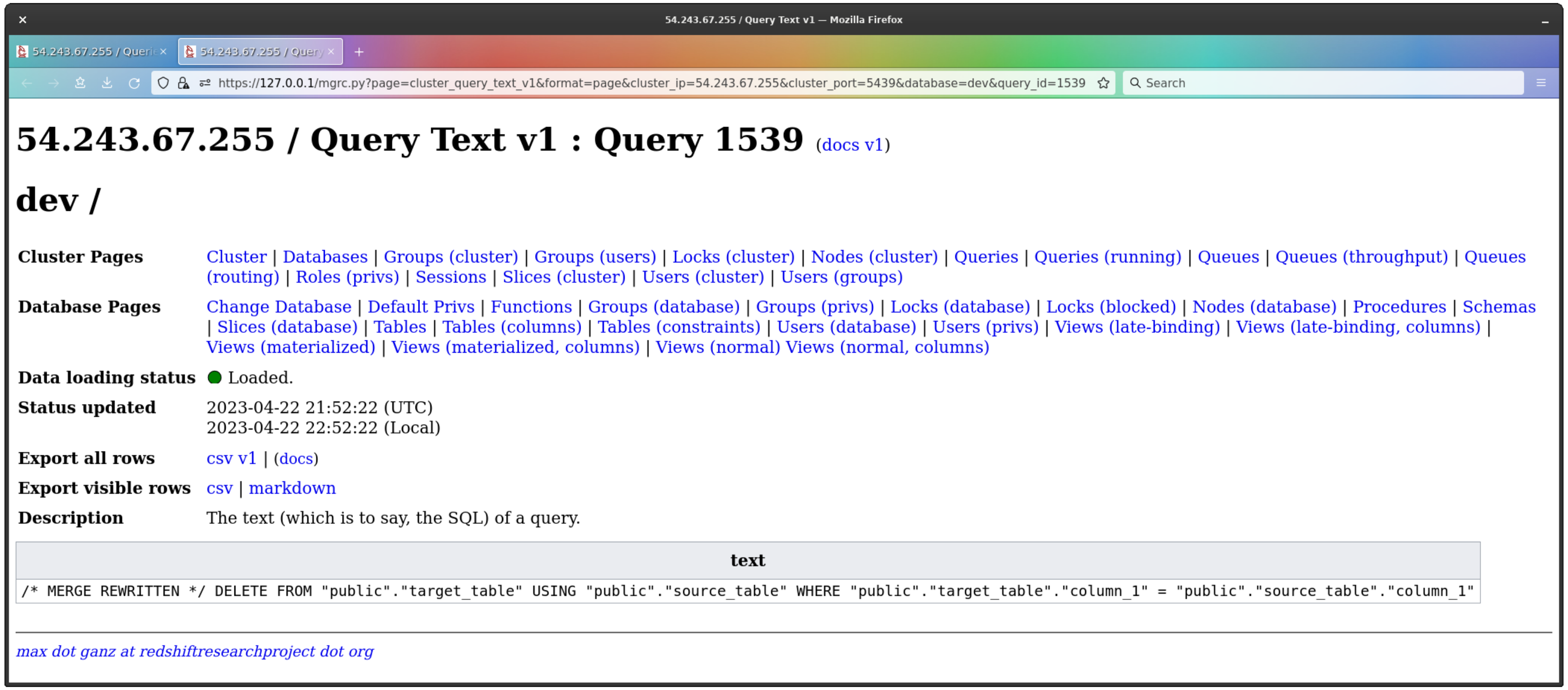


Figure 12: DELETE Query Text

```
/* MERGE REWRITTEN */
DELETE FROM
  "public"."target_table"
USING
  "public"."source_table"
WHERE
```

```
  "public"."target_table"."column_1" = "public"."source_table"."column_1"
```

As expected, here we see that any rows which are present in `source_table` and also present in `target_table` are now deleted, and the finally we have the disguised `INSERT`, which puts all rows from the temp table into `target_table`.

Benchmarks

The benchmarks were made before I understood what the temp table was for, and were intended to show the performance penalty of the temp table.

As it is, the temp table is by no means always necessary, and so the benchmarks are still useful in that they show the overhead of using the temp table (which `MERGE` will always use) against the situation where you issue an `UPDATE` and `INSERT` yourself, without the temp table.

The test method is to create `target_table` which has two columns, the first always holding the same value and being the distribution key (so all work is on a single slice), the second being an auto-incrementing integer but which we can explicitly over-ride.

We use the auto-increment functionality to generate 3,263,442 rows which each have a unique value.

We then create `source_table` in exactly the same way, but over-ride auto-increment to add 3,263,442 rows with the same value.

Both tables are then fully vacuumed and analyzed.

We then perform a `MERGE`, timing the result, and then repeat the setup, but next performing a transaction with an `UPDATE` and then an `INSERT`, and we sum the times for those two queries.

All times are taken using `\timing` in the `psql` client, so they include network time, but I expect the differences to be large, so I'm not worried about this.

The SQL for all this is given in Appendix A : Full Setup and Test SQL.

So, after all our setup work, we have `target_table` with 3,263,442 rows of unique integers, and `source_table` with 3,263,442 rows of the same unique integers, and also 3,263,442 rows of the value 999,999,999.

All rows are on a single slice, to help simplify what's going on, and all encodings are raw.

This is the `MERGE` query;

```
merge into target_table
using source_table on target_table.column_2 = source_table.column_2
when matched then update set column_2 = source_table.column_2
when not matched then insert values ( 1, source_table.column_2 );
```

And these are the `UPDATE` + `INSERT` queries, which are identical to those emitted by `MERGE`, except I'm not using the temporary table;

```
begin;
```

```
update
  target_table
set
  column_2 = source_table.column_2
from
  source_table
where
  target_table.column_2 = source_table.column_2;

insert into
  target_table( column_1, column_2 )
select
  1,
  source_table.column_2
from
  source_table
where
  not exists
  (
    select
      cast( 1 as int4 )
    from
      target_table
    where
      target_table.column_2 = source_table.column_2
  );

commit;
```

So, as noted, my manual version doesn't bother with the temp table, because there's no need for it.

After the `UPDATE` + `INSERT` has run, we can confirm `target_table` has been updated, as follows;

```
dev=# select count(*) from target_table;
  count
---------
```

```
 6526884
(1 row)

dev=# select count(*) from source_table;
  count
---------
 6526884
(1 row)

dev=# select sum(column_2) from target_table;
       sum
------------------
 3284742091057521
(1 row)

dev=# select sum(column_2) from source_table;
       sum
------------------
 3284742091057521
(1 row)

dev=# select
  count(*)
from
  (
    select
      column_1,
      column_2
    from
      target_table

    except

    select
      column_1,
```

```
      column_2
    from
      source_table
  );
 count
-------
     0
(1 row)
```

Using the code in Appendix A, I ran the full table setup code and the test code for `MERGE`, and the full table setup code and the test code for `UPDATE` + `INSERT` tests five times each (both after initial runs to get query compilation out of the way).

MERGE
3.239s
3.379s
3.077s
3.392s
3.991s

Discarding the slowest and fastest result, mean is 3.34s, standard deviation is 0.07s.

`UPDATE`	`INSERT`	`COMMIT`	Total
1.443s	0.925s	0.225s	2.493s
1.545s	1.229s	0.159s	2.533s
1.753s	1.339s	0.306s	3.358s
1.581s	1.088s	0.168s	2.837s
1.543s	1.127s	0.154s	2.824s

Discarding the slowest and fastest result, mean is 2.73s, standard deviation is 0.14s.

The results are as expected; the queries are the same, except the `UPDATE` and `INSERT` are not using a temp table, so they're doing less work.

Finally, note that `UPDATE` in its `FROM` can use a view or sub-query, where-as `MERGE` in its `USING` cannot - it can use only a table. This on the face of it is a wholly novel restriction imposed by `MATCH`, and I can't see a need or reason for it.

Summary

So where does this leave us?

Well, `MERGE` does the following when it is updating;

1. First, you must create and populate the source table, as the source must be a table, not a sub-query or view.
2. Scan the source table, and the target table, inserting all rows which are not present into a temp table.
3. Scan the source table, and the target table, updating all rows in the target table which are present in the source table.
4. Insert into the target table all rows in the temp table.

(Why the source table in `MERGE` must be a table rather than a sub-query or view, I do not know, as a temp table is made from it anyway, and `UPDATE` is fine with both sub-queries and views.)

Now, if you did this manually, you would do the following;

1. Scan the source table, and the target table, updating all rows in the target table which are present in the source table.
2. Scan the source table, and the target table, inserting all rows which are not present in the target table.

As you can see, manual steps 1 and 2 are `MERGE` steps 2 and 3, except that with `MERGE`, the insert work goes via a temporary table, and also when performed manually, the source table can be a sub-query or view, unlike when using `MATCH`, where you must use an actual table, which means creating and populating that table.

In short, `MERGE` is more expensive, as it does everything you would do manually, plus it requires you to create the source table as an actual table, and it writes the rows to be inserted to a temp table (which given how the temp table is created, prevents merge joins).

`MERGE` is not really a new command, but is implemented using using existing SQL commands and functionality, and on the face of it, you can do a better job than `MERGE` does.

`MERGE` also brings a new and seemingly unnecessary restriction, in that it's `USING` must use a table and cannot use a sub-query or view, unlike the `FROM` on `UPDATE`.

My feeling, although I've not thought it through in any details, is that there could be opportunities here for optimization, if `MERGE` has been written as a genuinely new command - I could imagine it making a single pass over the target table, and in a single query, performing the insert and the update or delete.

However, I'm of the view the core code base of Redshift can no longer be meaningfully modified - a symptom of this being that new functionality is being bolted on the outside, rather than adding to or modifying the internals of Redshift - and the implementation of `MERGE` is in line with that theory.

Detailed answers to the full set of questions posited in the Introduction do not need to be investigated further, because `MERGE` should not be used, so the answers are irrelevant.

There's also here a larger picture which needs to be considered.

Redshift is a *sorted* clustered relational database. Sorting is a computing method which offers with no hardware costs staggering efficiency - and so, scaling - but when and only when operated correctly. If operated incorrectly, it gives you nothing; the database internally is having to doing exactly the same work as an unsorted database, you will only be able to scale by adding hardware, and you do not have indexes. You would be better off being on an unsorted clustered relational database, as you can still scale by hardware, but you get indexes (and usually a great deal more functionality than Redshift offers).

Updates - which is to say, upserts, which is to say, merge - broadly speaking usually mean sorting is being operated incorrectly. If a system is using upserts such that merge looks interesting, probably you should not be on Redshift; and so in that sense, to my eye, even if merge *was* an improvement over manually issuing insert and a delete or update, it still wouldn't make sense - it is improper functionality to optimize on a sorted relational database.

Moreover, if we look at the work the manual version of `MERGE` has to do because of the lack of support for the `ALL` argument to `SELECT EXCEPT`, what actually might have improved performance is implementing `ALL` for `EXCEPT`. That would have been a genuine and useful enhancement.

Why was time spent producing `MERGE`, which is slower and more restricted than a manual `UPDATE` and `INSERT`, than producing an improved `EXCEPT`, which would have improved a manual `UPDATE` and `INSERT`?

As it is, given the internal implementation of `MERGE`, this looks to my eye like a marketing exercise, but with negative technical value. If you use it, you think you're doing well and you've got something new and improved, when it fact you're being harmed by it.

If `MERGE` was modified not to use the temp table, and removed the restriction where the source table must be a table, then it would in performance terms be identical to a manual `UPDATE` + `INSERT`, but you would now have the nice syntax of `MERGE`, which would be the sole gain. This could also have some use in porting existing SQL from other systems to Redshift.

The only caveat in all this is that I still do not understand why a temp table is being used. I can see no reason for it at all, which worries me; have I missed something? on the other hand, I can correctly duplicate `MERGE` without it, so it really does look to be unnecessary.

Conclusion

The `MERGE` command is implemented as a wrapper around existing SQL commands, calling `CREATE TEMP TABLE`, `UPDATE` and `INSERT` or `DELETE`.

You can yourself manually issue exactly the SQL commands `MERGE` issues; it is not an implementation which brings together the work of a merge into a single, optimized command, but rather, it is a macro.

The SQL standard specification of the `MERGE` command mandates each source row either matches or does not match, but never both; an `UPDATE` and `INSERT` pair *without* a temp table *can* allow a source row to match *and* not match - this happens when the source row matches, but the update command that is then as such issued converts the row into a row which no longer matches, and then of course the `INSERT`, reading the source table a second time, will go right ahead and insert.

In merges where the `UPDATE` does *not* cause rows to match, the temp table is indeed unnecessary, so the implementation of `MERGE` in Redshift is a kind of worst-case implementation - it has to be like that, to actually meet the specification of `MERGE`, but whenever the merge you're issuing would not cause matched rows to no longer match, the temp table is not needed, and then it's pure overhead.

As such, in the benchmarks (five iterations, slowest and fastest discarded), we see;

Method	Mean	StdDev
`MERGE`	3.34s	0.07s
`UPDATE` + `INSERT`	2.73s	0.14s

Additionally, `MERGE` mandates the use of and only of a table for the merge source rows, which is not necessary when issuing an `UPDATE` yourself, as `UPDATE` works also with sub-queries and views.

All in all, the `MERGE` command is syntactically very pleasant, but under the hood, the implementation in Redshift is I think absolutely not what people expected.

Finally, `MERGE` in Redshift (but not in Postgres, for example) restricts itself to a table for source rows, and I can't see any reason for that. The SQL emitted by the `MERGE` command obfuscates its `INSERT`, so I can't know what SQL it emits for that, but testing the `CREATE TEMP TABLE` and the `UPDATE`, and my own `INSERT`, I was able to use a sub-query in all cases.

Credits

1. mauro

   In version 1 of this document, I did not understand the purpose of the temp table being used by `MERGE` (where I work primarily on Redshift, I never use `UPDATE`, never upsert, and `MERGE` was new to me). The explanation was provided by mauro, which is that the specification of the `MERGE` command mandates each source row either matches or does not match, but never both; an `UPDATE` and `INSERT` pair *without* a temp table *can* allow a source row to match *and* not match - this happens when the source row matches, but the update command that is then as such issued converts the row into a row which no longer matches, and then of course the `INSERT`, reading the source table a second time, will go right ahead and insert.

   In merges where the `UPDATE` does *not* cause rows to match, the temp table is indeed unnecessary, so the implementation of `MERGE` in Redshift is a kind of worst-case implementation - it has to be like that, to actually meet the specification of `MERGE`, but whenever the merge you're issuing would not cause matched rows to no longer match, the temp table is not needed, and then it's pure overhead.

Revision History

v1

- Initial release.

v2

- Updated with explanation from mauro about the use by `MERGE` of its temp table.

Appendix A : Full Setup and Test SQL

Off-One Setup SQL

First, the one-off configuration (all work performed in `psql`).

```
set enable_result_cache_for_session to off;
set analyze_threshold_percent to 0;
set mv_enable_aqmv_for_session to false;
\timing on
```

Target and Source Table Setup SQL

Next, the SQL which creates and populates `target_table` and `source_table`.

```
drop table if exists target_table;

create table target_table
(
  column_1  int2  not null encode raw distkey,
  column_2  int8  not null encode raw generated by default as identity( 1, 1 )
)
diststyle key
compound sortkey( column_1, column_2 );

insert into
  target_table( column_1 )
values
  ( 1 );

insert into
  target_table( column_1 )
select
  1
from
  target_table as t1,
```

```
  target_table as t2;

insert into
  target_table( column_1 )
select
  1
from
  target_table as t1,
  target_table as t2;

insert into
  target_table( column_1 )
select
  1
from
  target_table as t1,
  target_table as t2;

insert into
  target_table( column_1 )
select
  1
from
  target_table as t1,
  target_table as t2;

insert into
  target_table( column_1 )
select
  1
from
  target_table as t1,
  target_table as t2;

vacuum full target_table to 100 percent;
```

```
analyze target_table;

/* MG2 : Now, examining the table, we see this;

         dev=# select * from target_table order by column_1, column_2 limit 10;
          column_1 | column_2
         ----------+----------
                 1 |        1
                 1 |        2
                 1 |        6
                 1 |       10
                 1 |       14
                 1 |       18
                 1 |       22
                 1 |       26
                 1 |       30
                 1 |       34
         (10 rows)
*/

drop table if exists source_table;

create table source_table
(
  column_1  int8  not null encode raw distkey,
  column_2  int8  not null encode raw generated by default as identity( 1, 1 )
)
diststyle key
compound sortkey( column_1, column_2 );

insert into
  source_table( column_1 )
values
  ( 1 );
```

```
insert into
  source_table( column_1 )
select
  1
from
  source_table as t1,
  source_table as t2;

insert into
  source_table( column_1 )
select
  1
from
  source_table as t1,
  source_table as t2;

insert into
  source_table( column_1 )
select
  1
from
  source_table as t1,
  source_table as t2;

insert into
  source_table( column_1 )
select
  1
from
  source_table as t1,
  source_table as t2;

insert into
  source_table( column_1 )
select
```

```
  1
from
  source_table as t1,
  source_table as t2;

insert into
  source_table( column_1, column_2 )
values
  ( 1, 999999999 );

insert into
  source_table( column_1, column_2 )
select
  1, 999999999
from
  source_table as t1,
  source_table as t2
where
      t1.column_2 = 999999999
  and t2.column_2 = 999999999;

insert into
  source_table( column_1, column_2 )
select
  1, 999999999
from
  source_table as t1,
  source_table as t2
where
      t1.column_2 = 999999999
  and t2.column_2 = 999999999;

insert into
  source_table( column_1, column_2 )
select
```

```
  1, 999999999
from
  source_table as t1,
  source_table as t2
where
      t1.column_2 = 999999999
  and t2.column_2 = 999999999;

insert into
  source_table( column_1, column_2 )
select
  1, 999999999
from
  source_table as t1,
  source_table as t2
where
      t1.column_2 = 999999999
  and t2.column_2 = 999999999;

insert into
  source_table( column_1, column_2 )
select
  1, 999999999
from
  source_table as t1,
  source_table as t2
where
      t1.column_2 = 999999999
  and t2.column_2 = 999999999;

vacuum full source_table to 100 percent;
analyze source_table;

/* MG2 : if we now examine source_table, we find;
```

```
dev=# select count(*) from source_table;
  count
---------
 6526884
(1 row)

(which is 2*3,263,442 = 6,526,884)

dev=# select count(*) from source_table where column_2 = 999999999;
  count
---------
 3263442
(1 row)

dev=# select count(*) from source_table where column_2 != 999999999;
  count
---------
 3263442
(1 row)

dev=# select * from target_table order by column_1, column_2 limit 10;
 column_1 | column_2
----------+----------
        1 |        1
        1 |        2
        1 |        6
        1 |       10
        1 |       14
        1 |       18
        1 |       22
        1 |       26
        1 |       30
        1 |       34
(10 rows)
```

```
        dev=# select count(*) from source_table where source_table.column_2 in ( select column_2 from target_table );
          count
        ---------
         3263442
        (1 row)
*/
```

MERGE Test SQL

```
merge into target_table
using source_table on target_table.column_2 = source_table.column_2
when matched then update set column_2 = source_table.column_2
when not matched then insert values ( 1, source_table.column_2 );
```

UPDATE + INSERT Test SQL

```
begin;

update
  target_table
set
  column_2 = source_table.column_2
from
  source_table
where
  target_table.column_2 = source_table.column_2;

insert into
  target_table( column_1, column_2 )
select
  1,
  source_table.column_2
from
  source_table
where
```

```
  not exists
  (
    select
      cast( 1 as int4 )
    from
      target_table
    where
      target_table.column_2 = source_table.column_2
  );

commit;
```