Multi-Version Concurrency Control and Serialization Isolation Failure

Max Ganz II @ Redshift Research Project

5th March 2023

Abstract

This white paper describes and explains Multi-Version Concurrency Control (`MVCC` for short), which inherently entails describing and explaining transactions and table locks as these are part of `MVCC`, where aside from generally explaining what's going on inside Redshift, the particular goal is to explain serialization isolation failures, as these originate from `MVCC`; to understand their origin, the manifold ways by which they occur, how to write code such that isolation failures do not occur in the first place, and how to fix them when you are working with code where they do.

Contents

Introduction

Redshift, as you dear reader are likely aware, is able to run multiple queries concurrently; offering this functionality brings with it a range of complications and considerations all basically along the lines of - how the *hell* do we make this work?

There are in fact a certain number of computing methods available to make it all work, where Redshift (Postgres, too) of these methods uses the method known as Multi-Version Concurrency Control, or `MVCC` for short.

`MVCC` brings along with it a small number of related concepts and methods, in particular transactions and table locks, which are part and parcel of `MVCC` as a whole and have to be understood to be able to understand `MVCC` proper.

Critically, though, `MVCC` is by no means whatsoever able to handle all the possible situations that can arise with multiple queries running concurrently; and when `MVCC` is placed in a situation where it cannot cope, it *has* to abort one or more queries - because there is literally no other solution.

These aborts, which come from situations where `MVCC` cannot cope, are ***serialization isolation failures***.

There are to my mind two fundamental causes for serialization isolation failures, but there are *any* number of ways in which to combine queries to invoke either fundamental cause of failure.

It is then not safe to simply assume Redshift through the use of `MVCC` can handle *all and any* queries running concurrently; this is simply not so. To avoid inducing `MVCC` failure, you must understand what makes it fail, so you can ensure you do not design systems or issue queries make it fail in the first place - and also that when you do discover it is failing, to be able to exactly and correctly diagnose

the origin of the failure and so be able to take the exact and specific step or steps to solve the problem.

This white paper then explains the genesis of `MVCC`, how it works internally, then transactions, then table locks and finally, brings all of this together to explain and describe serialization isolation failures.

Additionally, along the way, as a part of figuring out how it all works, a number of important additional proofs were developed and/or uncovered (such as what locking behaviour occurs for `DELETE` and `UPDATE`), and these too are presented.

Finally, normally a white paper outlines in the Introduction a specific question, where the Introduction is followed by the Test Method and Results, and then the Discussion of the results.

This is predicated on the test method making sense in and of itself; however, here, with this white paper, where it is not so much investigating but rather is simply *explaining* it is of course impossible for the test method to make sense without reading the Discussion first; there are just too many novel concepts and ideas and so on.

Accordingly, for this white paper, I have moved the Test Method and the Results to Appendix A and Appendix B, respectively, with Appendix C now being the full data dump.

Discussion

Database Design Problems

The simplest possible database we can imagine supports only a single query at a time.

In this situation, life's a breeze - we never have to think about what happens when multiple queries run concurrently, because they don't; we simply execute the current query and that's it.

The queries read, they write, it's all simple and easy, except maybe for just one complication, which is what do you do when a query aborts?

A query might intend to say add a lot of new rows and get half-way through and then tries to write a `NULL` to a `NOT NULL` column and so falls over.

Question is, what do you want do about the rows which were written prior to the point the query aborted? which is to say whether or not you want the database to automatically delete those rows, or whether you're happy to leave them in the table.

A first consideration is the thought that if a query aborts, probably we'll fix it and run it again, and keep doing this until it does work. It's hard to imagine any scenario where we're happy accumulating multiple copies of the rows which are written successfully prior to each abort.

We can also imagine we had a query which runs every now and then and writes new rows into a table, let's say financial data, and that the data only makes sense from a business logic point of view when it's all in the table together,

because the nature of the computation being performed on the data is such that if you read just *some* of the rows, rather than all of them, you'll compute the wrong result and charge people the wrong amount and no one will notice for ages until finally someone does and you get tons of bad press and everyone laughs at you.

Also, in general it's much easier to reason about what's going on with queries if we know they will only see either all the new rows from queries adding rows (if they complete successfully) or none (if they abort, or are still in progress), rather than having to think about all the extra possibility arising from partially written sets of rows.

Finally, thinking about it from the other direction, as it were, we can imagine scenarios where it's bad to read a partially written set of rows but it's hard to think of scenarios where having all the rows turn up all together at once is bad.

In any event, the (far as I know) universal choice is that a query adds either all of its rows, or none at all.

With a single-query-at-a-time database, all the problems which come from interactions with other queries inherently do not exist, so when a query aborts, we can do whatever clean up work we want before we let the next query run, so life is still a breeze; we can probably think of a dozen solutions to cleaning up those rows.

However, we don't really want a database which can only run one query at a time. What we want is to be able to concurrently run multiple queries, because it's much more useful.

With this in mind, the idea that we do now have multiple queries running concurrently, let's revisit the query adding new rows, which when halfway through tries to write a `NULL` to a `NOT NULL` column and aborts.

The most obvious and immediate problem is that there is now in the table a partially written set of rows, and we need to make sure no other query reads any of them.

There are however upon a little thought plenty of further problems which emerge.

Imagine we have a query which is adding new rows, and has so far added some but not all of the rows it will write, and then a reading query starts and gets to the point where it's reads some of these rows, and *then* the writing query aborts. Now not only do we need to ensure no other queries read the partially written set of rows, we also somehow have to get the reading query to "unread" the rows it's already read!

Ouch. Complicated. Bit of a CASE NIGHTMARE GREEN. Not a programming problem I'd want to have to face.

So what do we do about this?

Well, as a starting point for developing a solution, consider that we've already stated a goal; that the rows added by a query should all turn up at once, or not at all.

Queries are in one of three states; running, aborted or completed successfully.

We could then say that queries read rows written only by queries which completed successfully; so if a query is still running - and so may have written some rows but have more to come - we ignore its rows, and if a query aborted, we ignore its rows.

This would solve both the problems described so far.

However, upon scrutiny, we can see at least one problem remains.

Imagine a query is adding new rows to a table. It's written about half the rows it will add, at which point a query begins reading rows from the table and this query happens to run much more quickly than the writing query and so reaches a point where it has read most of the rows added so far.

Now, we're currently imagining our database is arranged in such a way that queries only read the rows written by queries which have completed successfully - so here, what's actually happening is the reading query is reading the rows, as after all it can't help but see them - they're on the disk, after all - but it is *ignoring* them, because, by a mechanism we have yet to figure out, it knows they're written by a query which is still running.

However, now, at this point, the writing query finishes adding the rows it will add and completes successfully.

Now we have a problem.

The reading query has already ignored a good portion of the rows added by the writing query because at the time the reading query read them, the writing query was still running; but it will now go and read the remainder, because the writing query has completed successfully!

Our goal was to ensure that either all, or none, of the added rows show up for other queries, and this scenario breaks that goal.

However, it takes only a small modification of our proposed solution to make it handle this case : rather than saying a query can read rows from only queries which have completed successfully, what we actually want to say is a query can read rows from only queries which completed successfully *and* completed successfully *before* the query started.

So a query will ignore all rows from queries which are running, or which aborted, or which completed successfully but only *after* the query started running.

It is by this we guarantee a query will read only complete sets of rows.

This then is in fact practically telling us what we need to do to implement - we need to know which query wrote a row, and we need to know the state of queries (when they started, if they're running, aborted, or completed successfully).

To keep track of which query wrote a row, we can add a column to every table, a column managed by the database, and store a unique identifier for the writing query there, and we can also simply keep track in the database (which it will need to do anyway) of the state of each query.

This is in fact the essence of Multi-Version Concurrency Control.

As we will see, there's more detail to it, which you come to realise you need as you try to really implement the method, but this is the essence of it; keep track of query state, and have an extra column (or two, as it turns out) in each table, where you keep track of some per-row state. The goal of it all is simply to ensure that a query uses only rows which were written by queries which completed successfully *before* the query started.

Multi-Version Concurrency Control

There are three core concepts in `MVCC`.

The first is the idea of a single global counter - the "generation number" - which begins at 0, which increments by one every time a query is issued, where the database informs each query of the value of the counter when query starts, so each query knows its generation number and each query has a unique generation number and you can tell the order in which queries started from their generation number.

The second is the idea of global state on a per-query basis (using the query generation number to uniquely identify each query) which indicates if the query is running, has completed successfully, or has aborted; additionally, if a query complete successfully, this state also records the generation number at the time the query completed, so we know *when* the query completed.

The third is that each table has two extra columns, for per-row `MVCC` state.

The Generation Number and Per-Row State

The first column of these two columns indicates the generation number of the query which inserted the row (this column is necessarily always set as you can't have a row unless it was inserted), and the second indicates the generation number of the query which deleted the row (NULL when a row has not been deleted) - and I now need to explain the reason and purpose of the delete column, because on the face of it, that's crazy; if a row has been deleted, it's gone, so why would we store anything?

So, remember we talked about how a reading query should use only the rows added by queries which completed successfully before the query started?

Well, this is actually a more general concept - it's not just for adding rows - what we really mean to say is that a query should use all *changes* made by queries which completed successfully before the query began, and so that includes, for example, deletes.

If this was not so, all the problems we described before about queries reading partially written sets of rows would occur, only now with queries reading partially deleted sets of rows.

For example, we could have a query reading rows from a table, being able halfway through the table, and then another query comes along, deleting rows. It deletes some rows from the first half of the table, and then some from the second half of the table. The reading query has *already read* the first half of the table, and so *has* read those deleted rows, but will not see the deleted rows from the second half of the table.

So what we need, in exactly the same way that we needed a column to keep track of which query added a row, is a column where we store which query deleted a row, so a query can know whether or not to ignore the delete, because it was made by queries which aborted, or which are still running, or which completed successfully but only after the query started, in just the same way that it might ignore an added row.

This though has an immediate and slightly eye-opening implication : when a row is deleted, *it must remain in the table*, because queries which started earlier than the deleting query will still use it.

So how do we actually get rid of deleted rows? (which we'll call "purging", to differentiate from "deleting").

This is handled by the `VACUUM` command, in both Postgres and Redshift.

Where the database knows the status of all queries, it knows the generation number of the oldest running query. Any rows which have their delete generation set, and to a generation earlier than the oldest running query, can *never* been seen by any query ever again - and so, when `VACUUM` runs, these rows are *purged* - finally, actually deleted, removed, expunged, zapped, fried, buried in peat, made to feel unwanted, you get the picture.

Rows inserted by queries which aborted are also purged by `VACUUM`.

As an aside, everything dies horribly if the generation count wraps around to 0. This Can Never Happen. I don't know how this is handled, although I can think of one or two ways (per-table offsets from the main counter), but this is a bit of a side-show as far as we're concerned; the database handles it in the background and it makes no difference to how we reason about `MVCC`.

We should now note we've handled three - `SELECT`, `INSERT`, `DELETE` - of the four main SQL commands, which leaves `UPDATE`.

Update modifies a row, but we can already imagine we'll need the earlier version to stick around for the queries that will read it, while at the same time the new version is present for future queries.

What happens then is that when a row is updated, the delete generation number is set in the row, and then a new row is created, with the new values specified by the `SET` clause, where the new row has the insert generation number being that of the update query, and the delete generation set to `NULL`.

Databases are normally casually described as being composed of tables which are composed of rows - as if there's one row (physical entity) in the table *per record* (logical entity), so that a row and record are the same thing, and so it would be that deleting a row is the same as deleting a record and so when you delete a row/record, bam! it's gone and that's it.

With `MVCC`, what you really have are one or more rows (the physical entities), each of which is *a version of* a record (the logical entity), and every time the record (logical entity) is updated, you make a new row (physical entity), and a query uses the most recent row prior to when the query started (as any later rows cannot have come from queries which completed successfully *before* the query started).

In short then, a query will know to ignore all *changes* made by all other queries, *unless* those other queries completed successfully before the query started, and two columns, one to keep track of which query inserted a row, and another to keep track of which query deleted a row, are enough to achieve this.

Now, with regard to the delete and insert generation columns, their actual names (in Redshift, not Postgres - Postgres calls them `xmin` and `xmax`, which are empty vestigial columns you still see in all Redshift tables) are `deletexid` and `insertxid`.

The reason for these names will become clear once transactions are explained.

In Postgres these columns are invisible, but they are accessible - if you explicitly name them in the select part of a `SELECT` you get them, otherwise not - but in Redshift, in keeping with the spirit of AWS hiding information from users, these columns are both invisible and inaccessible, since you can't select them.

(In Redshift, both of these columns are `int8`, and are in the system tables specified as being `raw` encoded, but I don't think they are; I recall the system tables used to say they were `runlength` encoded, and absolutely, from how I see them behave, they *not* `raw`. There is in fact one more invisible column, managed by and used by Redshift by not part of `MVCC`, which is the row ID column. This column is also `int8` and really is `raw` encoded, so if you have a wonderfully compressed table with tons of run-length encoding, you're still going to have this one uncompressed `int8` column there, being huge at you :-)

Note that with this arrangement - extra columns per table, global query state - the cost of a query aborting is extremely small. There's no need to actually do anything to any of the records that query wrote; all that happens is that its status is changed to aborted, and that's all that queries need to know to ignore the changes made by the aborted query.

However, this also means that rows inserted by aborted queries remain in tables until `VACUUM` runs, and there is a test case to demonstrate this, but it can only be explained once *transactions* are explained, which is coming soon.

Per-Query State

So much then for the generation counter and the per-row state.

We turn now to the second core `MVCC` concept, per-query state, and how the database keeps track of per-query state, because `MVCC` is not going to work if the database needs to forever maintain track of the state of every single query that ever ran - that's too much information.

What actually happens is that the database keeps track of the state of all queries more recent than the oldest currently running query, and also the generation numbers of all aborted queries.

When `VACUUM` runs, it purges aborted rows, and when all the rows written by an aborted transaction have been purged, that query is removed from the list of aborted queries (the database has to so some additional book-keeping work to keep figure out if this goal has been achieved - there's a number of different

ways you could do this, for example, keeping track of which tables a query wrote to; the implementation will vary by database).

Now, any query generation number older than the oldest running query, which is not in the aborted list, must then have completed successfully (if it's not running, and not aborted, there *is* only one other state for a query to be in); and so when a query reads a generation number from `deletexid` or `insertxid`, either the query is more recent than the oldest running query, in which case the state is in memory, or it is aborted, in which case it is found in the aborted list, or finally it is implicitly known that it must then have completed successfully.

Recall that a query uses and only uses changes made by queries which completed successfully before the query started.

This is to handle the situation where a query starts reading rows, gets half-way through a table, and then a writing query which is making changes throughout the table, which started *before* the reading query, completes successfully.

The reading query must still ignore those changes, that it has yet to read, in the second half of the table - but how can it know to do this, if *all* we store on a per-query basis is that state of the query? the reading query will find a row changed by the writing query, look up the generation number, and find the writing query completed successfully.

We need something extra, so the reading query can know not just that the writing query completed, but *when* it completed.

I've never been about to find any information about how this is done and so I am speculating that the way this is done is by storing for each query the generation number which was current at the time the query completed.

Then, when the reading query checks the state of the writing query, it can see it complete successfully - we already know we store the query state - but it can also see if the writing query completed before or after the reading query started, because if it completed before, the generation number stored when the writing query completed will be less than the generation number of the reading query.

Transactions

Quite a bit earlier on, one of the rationales for designing databases such that the changes made by a query either all show up, or none show up, was a business use case where the changes being made *had* to be read all together, because if they were not, if only some of the changes were read, it would be a problem for business logic and the wrong results would be produced.

It's an easy jump to go from that business use case, where the results of a *single* query must all show up together, to the idea of a business use case where the results of *multiple* queries must all show up together (or it's a problem for the business logic and the wrong results would be produced).

The usual example rolled out at this point is the idea of debiting one account and crediting another; this requires two queries, but you need all the changes to occur, or none at all.

The method used to achieve this is *transactions*.

A transaction is a set of queries all of which are given the same generation number.

The user starts a transaction, by issuing the `BEGIN` command, and the transaction is given a generation number and like a query, has the same states; it can be running, aborted, or completed successfully. At this point, the transaction is running.

From this point on, now there is a running transaction, all queries issued by the user receive the generation number of the transaction. The queries run as soon as they are issued, and make all the changes they will make, but when other queries come to look up the state of the query which made those changes, they get the state of the *transaction* - so even though the queries themselves have completed successfully, the changes they have made are still seen as being from a query - the transaction in fact - which is running.

Now, with single queries, the user simply issues the query; the query then starts, and runs, and it will by itself abort or complete successfully and then the query is over. The user takes no action beyond starting the query.

With a transaction the database of course has no idea when the user has finished issuing queries using the transaction, and so it is necessary for the user when they are done issuing queries using the transaction to issue an SQL command which instructs the database that the transaction is now complete.

This causes the database to change the state of the transaction to successfully completed - which, as we can immediately perceive, given the nature of `MVCC`, makes *all* the changes made by *all* the queries in the transaction all show up.

Now, when a user comes to end the transaction, they actually have a choice - they can end the transaction by instructing the database to successfully complete the transaction (the `COMMIT` command), or they can instruct the database to end the transaction by aborting it (the `ROLLBACK` command).

In the former, the generation number used by the transaction is recorded as having completed successfully, in the later as having aborted (and so all other queries then ignore the changes the transaction made).

Finally, if a query issued using a transaction in fact itself aborts, then the transaction is automatically aborted.

This is because the transaction has a single generation number, and each generation number has a single state (running, aborted, completed successfully), and so necessarily that state is being used by all the other queries in the database to know whether or not to use or ignore the changes made by the queries in the transaction; we can hardly then have it so some of the queries in the transaction completed successfully *and* some aborted, because there's no way to communicate this to the other queries reading the changes because there is only one state.

Now - the time has come to unleash upon you the final revelation.

There *are* no single queries.

Everything is a transaction.

When you issue a `BEGIN`, you get a transaction which ends only when one of the constituent queries aborts or you issue `COMMIT` or `ROLLBACK` (which means abort, but in language `COBOL` programmers can understand).

When you do not use `BEGIN`, but directly issue a single query, you get a single-query transaction, which automatically completes when the query completes, and it completes with the same status as the query (completed successfully or aborted).

`MVCC` in fact is *always* assigning the generation number to a transaction, not to a query, and the actual formal name for the generation number is the `transaction ID`, often shortened to `xid`. You can now see why the two system columns holding per-row `MVCC` state are named `deletexid` and `insertxid`.

Queries have their own unique identifiers, of course, so the database can keep track of them, but `MVCC` isn't using those identifiers. These are naturally enough known as a "query ID".

We now finally know enough for me to be able to start discussing test cases and following we see a test case which demonstrates the behaviour of the `deletexid` and `insertxid` columns.

As a reminder, recall that due to Redshift hiding the system columns, we have to use an indirect method to read them, and what this allows us is to see the minimum and maximum value for the `deletexid` and `insertxid` columns. As such, the test is limited to inserting only two rows, but this is enough to demonstrate everything that needs to be demonstrated about these columns.

1. create test table
2. insert one row
3. query transaction_id = 9176

At this point, one row has been inserted, and the ID of the transaction which performed the insert is 9176.

We have just one row now, so both the minimum and maximum values of the two columns will be the same, as we expect then that the `insertxid` is that of the inserting transaction (which it is) and the `deletexid` is the maximum value of an `int8`, which is 9223372036854775807, and it is - this indicating that the row has no `deletexid` set.

Value	Expected	Actual
`insertxid` (min)	9176	9176
`insertxid` (max)	9176	9176
`deletexid` (min)	9223372036854775807	9223372036854775807
`deletexid` (max)	9223372036854775807	9223372036854775807

4. insert second row
5. query transaction_id = 9179

Now we have a second row. The inserting transaction ID is 9179. This then leaves us to expect the minimum `insertxid` to be 9176 (from the first row) and

the maximum 9179 (from our new row), both are which are indeed so; and that `deletexid` has not changed (after all, nothing has been deleted).

Value	Expected	Actual
`insertxid` (min)	9176	9176
`insertxid` (max)	9179	9179
`deletexid` (min)	9223372036854775807	9223372036854775807
`deletexid` (max)	9223372036854775807	9223372036854775807

6. delete first row
7. query transaction_id = 9182

Now things are a little more interesting. The first row, from transaction ID 9176, has been deleted.

Remember that all a delete does is set the `deletexid` column; the row remains in the table. It is then that the minimum `insertxid` should remain as 9176 - the row we deleted had its `deletexid` set but remains in the table, and the maximum remains at 9179 (as we inserted no more rows).

We also see now that the first row has had its `deletexid` set, but the second row still does *not* have its `deletexid` set, so the minimum `deletexid` is now 9182 (the transaction which deleted the first row) and the maximum remains max `int8`.

Value	Expected	Actual
`insertxid` (min)	9176	9176
`insertxid` (max)	9179	9179
`deletexid` (min)	9182	9182
`deletexid` (max)	9223372036854775807	9223372036854775807

8. VACUUM table

We now issue a VACUUM. This will purge the deleted first row from the table, and so we are now back to having only the second row; which was inserted by transaction 9179 and is not deleted.

Value	Expected	Actual
`insertxid` (min)	9179	9179
`insertxid` (max)	9179	9179
`deletexid` (min)	9223372036854775807	9223372036854775807
`deletexid` (max)	9223372036854775807	9223372036854775807

9. update all rows (which means one row)
10. query transaction_id = 9189

Now, an update. This sets the `deletexid` of every updated row, and inserts a new row, with the new values for that row, as specified by the `SET` clause of the

update statement.

So here, we have just the one row (the second row we inserted), and this has been updated. This leaves with one row with `deletexid` set, and one row (the newly inserted row) where it is not.

As such, `insertxid` minimum is the transaction ID of the transaction which inserted the second row, 9179, and the maximum of the update transaction, where this inserted the new row.

For `deletexid`, the original row will have this set, and to the transaction ID of the update transaction (9189), and the new row will have this set to max `int8`, both of which we find.

Value	Expected	Actual
`insertxid` (min)	9179	9179
`insertxid` (max)	9189	9189
`deletexid` (min)	9189	9189
`deletexid` (max)	9223372036854775807	9223372036854775807

So, moving on from this first test, you remember that `VACUUM` is used to purge rows from tables?

Remember also the way it works is that the database knows the transaction ID of the oldest running transaction, and what `VACUUM` does is purge all rows which have a `deletexid` *older* than this.

Well, now, consider what happens if you open a transaction and then leave it open.

Ooooooooh. Nasty. Yes, that's really what happens. `VACUUM` *cannot* purge rows, from ***any*** table, at least, not rows deleted after you opened that transaction.

We have a test case for this.

Timestamp	Event
0.0	make connections to database
0.7071206569671631	connections made
0.7071232795715332	create and populate test tables
2.9194986820220947	test tables created and populated, 100 rows each
2.9195001125335693	connection 1 : start transaction and issue select query on `table_1`
3.0636632442474365	connection 1 : select query completed, transaction left open
3.0636658668518066	connection 2 : issue delete query on `table_2`
3.1823501586914062	connection 2 : delete query complete
3.2959678173065186	connection 2 : `table_2` has 100 deleted rows
3.2959771156311035	connection 2 : vacuum full `table_2` to 100 percent

Timestamp	Event
3.685190200805664	connection 2 : vacuum complete
3.77941632270813	connection 2 : `table_2` has 100 deleted rows
3.779421806335449	connection 1 : commit transaction
3.8184521198272705	connection 1 : commit complete
3.9197051525115967	connection 2 : `table_2` has 100 deleted rows
3.9197168350219727	connection 2 : vacuum full `table_2` to 100 percent
4.239577054977417	connection 2 : vacuum complete
4.33547568321228	connection 2 : `table_2` has 0 deleted rows
4.335484981536865	disconnect connections from database
4.3360888957977295	disconnections complete

We make two test tables and give each 100 rows.

In the first connection to the database, we open a transaction and issue a select on the first table, to get a transaction ID.

We then in the second connection delete all the rows in the *second* table. So - absolutely no connection to the first table, or the first connection.

We then in the second connection issue `VACUUM` on the second table - and we see it does nothing. We still have 100 deleted rows.

We then in the first connection commit the transaction, and in the second connection repeat the `VACUUM` on the second table and *now*, finally, we've actually purged the deleted rows.

We can also turn now to the test case demonstrating that changes made by queries which abort remain in the database, the test case examining the particular example of inserting new rows.

Timestamp	Event
0.0	make connection to database
0.35033321380615234	connection made
0.35033512115478516	create empty test table
0.6810824871063232	empty test table created
0.7387111186981201	connection 1 : select count(*) from table_1 = 0
0.8353245258331299	connection 1 : STV_BLOCKLIST number rows in table_1 = 0
0.835395097732544	connection 1 : start transaction and issue insert query on `table_1`
0.9689481258392334	connection 1 : insert query complete, rows inserted
1.023545742034912	connection 1 : select count(*) from table_1 = 3

Timestamp	Event
1.1150462627410889	connection 1 : STV_BLOCKLIST number rows in table_1 = 3
1.1150541305541992	connection 1 : abort transaction
1.1560404300689697	connection 1 : abort complete
1.5720436573028564	connection 1 : select count(*) from table_1 = 0
1.700000286102295	connection 1 : STV_BLOCKLIST number rows in table_1 = 3
1.7000102996826172	connection 1 : vacuum full `table_1` to 100 percent
1.9459662437438965	connection 1 : vacuum complete
2.0201823711395264	connection 1 : select count(*) from table_1 = 0
2.1518771648406982	connection 1 : STV_BLOCKLIST number rows in table_1 = 0
2.1518847942352295	disconnect connection from database
2.152294635772705	disconnection complete

The number of rows returned by `select count(*)` is a pretty good indicator of the number of non-deleted rows in a table. It's not perfect - other transactions might be running, which have added or deleted rows, and the query performing the count of course cannot yet use those rows - but these tests have no other transactions running so the counts are in fact completely accurate.

The number of values in each block for a single column, as retrieved from `STV_BLOCKLIST`, indicate the actual total number of rows, regardless of whether they are deleted or not. (One minor note - if the distribution style is `ALL` or `AUTO(ALL)`, then the number of rows is multiplied by the number of worker nodes; but we're using `key` distribution, so we don't need to handle this).

We can then tell the difference between the total number of rows (from `STV_BLOCKLIST`) and the number of non-deleted rows (from `select count(*)`).

In the test then, first an empty table is created, and we count the total number of rows, and the number of non-deleted rows, both of which are as we would expect zero.

We then begin a transaction and insert three rows. Counting again, we see both counts are now 3; three rows in total, and of them, all 3 are not deleted.

The transaction is then aborted.

Again, we count, and now we see `select count(*)` is coming back with 0 rows - as it should do; the transaction aborted, so the count query knows to ignore those rows.

However, the rows *do* still show up in `STV_BLOCKLIST`, because they *do* exist and they *are* in the table; it's just they're dead weight, rows from an aborted query.

Next, we issue a `VACUUM`, and then count again, and now we see the count from `STV_BLOCKLIST` has returned to 0. The rows have now been purged from the table.

Moving on, one final note : in Redshift (and maybe in Postgres, I've not checked), a transaction only gets a transaction ID upon issuing its first query. Merely opening a transaction, with `BEGIN`, does *not* allocate a transaction ID. This is why a lot of the test cases open a transaction and then issue a `SELECT`.

Table Locks

`MVCC` handles a wide range of situations where multiple queries run concurrently, but the SQL specification provides for functionality which inherently and unavoidably breaks `MVCC`.

To handle these situations, `MVCC` is not enough, and some limited table locks always also have to be implemented.

I say limited because `MVCC` provides so much functionality that a fully-functional lock scheme, with enough functionality such that it can *by itself* properly control table access, isn't required.

All that's needed is a minimal and lightweight set of locks, which compliments `MVCC` and in fact *depends* upon it, because the locks provided are lightweight enough they cannot, by themselves, properly control table access.

These locks are basically a minimal extension of `MVCC` to handle certain special cases which unavoidably emerge from the SQL specification simply being what it is and offering what it offer, and which `MVCC` by itself simply cannot handle.

Now, regarding the SQL specification, the main problem I know of is `TRUNCATE TABLE`.

The purpose of this command is to completely empty a table of rows.

Now, we could do this using `DELETE`, and that would work just fine for `MVCC`, because, as we've seen, `MVCC` handles delete queries. However, this is worst case from a performance point of view. We have to write to the `deletexid` of every single row, and we already know what we really want to do is wipe the table, and this is why `TRUNCATE TABLE` exists in the first place. It's a command which says - don't mess about touching all the rows, just reset the table.

Problem is, how the hell does that work with `MVCC`? you have a dozen queries running, they're all seeing different versions of the table, and then POW - the record and row count for the table is set to zero and the files holding the data are deleted.

That's not going to fly. You're going to have a lot of very confused queries.

So what do we do?

We do a little bit - just a touch - of table locking.

As I mentioned before, this is lightweight table locking; it depends upon `MVCC` being present and only makes sense with `MVCC` in operation.

Redshift inherits its table locking from Postgres, but has simplified it a lot, because Redshift has simplified a lot some of the `MVCC` options Postgres offers. I will here then stick to describing table locking as it is in Redshift.

In Redshift, there are three types of table lock, the survivors from Postgres, which has about eight. This is also why they have strange looking names, because they're unchanged from Postgres and made sense when you had eight types of lock, where the missing locks had related names and the whole set of names made sense.

To begin with I'm not going to use the official names of the locks, because the names are totally confusing and unhelpful.

Instead I'm going to convey what the locks *do*, and then once that's clear, I can safely give you their official names.

So here's what Redshift has, my unofficial names;

Lock	Function
Exclusive	Exclusive lock
Write	Blocks Write locks and Exclusive locks
Read	Blocks Exclusive locks (but *not* Write locks)

So, to start with, the simple case is the exclusive lock - and this is what `TRUNCATE TABLE` will ask for.

When a query starts, it indicate what locks it wants on which tables. Each table has a queue of requested locks, which are serviced first-in, first-out, and there's logic, which implements the "Function" column in the table above, which decides when a lock can be granted.

If the lock quested by a query cannot be granted, the query is blocked until it can be granted, and this blocks the other queries behind the blocked query - first-in, first-out, remember.

In the case of an exclusive lock, the logic is simple : if any locks have been granted on the table, the exclusive lock is blocked until they have all been released. When all locks on a table have been released, so that no locks remain, then the exclusive lock is granted, and no other locks can be granted until the exclusive lock is released.

A truncate query requests then an exclusive lock, is blocked until all running queries have completed (and so released their locks), then the truncate query and the truncate query alone is now running on the table and as such it is now safe, for `MVCC`, for the query to reset the table, which it does, and then it releases the exclusive lock.

Turning now to the read lock, this exists simply to block exclusive locks until the query which holds the read lock has completed; every reading query takes a read lock on the tables it reads from.

The read lock *doesn't* stop write locks, because there's no need for it - we have `MVCC` to deal with the situation when multiple queries run concurrently, some

of which will be making changes (inserts, updates), some of which will only be reading.

Write locks do block each other write locks though (and the exclusive lock, of course, since writing queries depend on `MVCC`), which is the whole point of them for Redshift, because Redshift serializes writes to tables.

So here now are the hopeless official names;

Lock	Function
AccessExclusiveLock	Blocks all locks
ShareRowExclusiveLock	Blocks ShareRowExclusiveLock locks and AccessExclusiveLock locks
AccessShareLock	Blocks AccessExclusiveLock locks

And here are the locks taken by the various SQL commands;

SQL	Lock
`COPY`	ShareRowExclusiveLock (Write)
`DDL`	AccessExclusiveLock (Exclusive)
`DELETE`	See below
`INSERT`	ShareRowExclusiveLock (Write)
`SELECT`	AccessShareLock (Read)
`UNLOAD`	AccessShareLock (Read)
`UPDATE`	See below

Here `DDL` includes things like `ALTER TABLE`.

This list, which is taken from a premium support page is actually, as ever, lacks vital information and as such misleads readers.

You see it turns out what Redshift actually often does is *request* one type of lock (typically a write lock) but once that lock is granted, the lock is *converted* into another type of lock (usually exclusive).

What this means then is that if you examine a list of queued locks, what you see is *not* what you get. To actually know what locks will be granted, you will need also to examine the SQL of the queries requesting locks *and* know that they are queries which convert their locks from one form to another.

By and large I suspect it's pretty simple in practise - anything which is listed as taking an exclusive lock actually requests a write lock and converts it to exclusive once granted.

Nevertheless, the docs are - as ever - flawed and misleading, and the implementation is poor; what should have been done is that additional locks types were made, which indicate by their type that they will convert when granted, so that then looking at a list of queue locks what you see is actually what you get.

`DELETE` and `UPDATE`

The `DELETE` and `UPDATE` commands are special cases.

The official documentation attains truly Alpine heights of dizzying incomprehensibility with regard to the locking behaviour of these commands.

> UPDATE and DELETE operations behave differently because they rely on an initial table read before they do any writes. Given that concurrent transactions are invisible to each other, both UPDATEs and DELETEs have to read a snapshot of the data from the last commit. When the first UPDATE or DELETE releases its lock, the second UPDATE or DELETE needs to determine whether the data that it is going to work with is potentially stale. It will not be stale, because the second transaction does not obtain its snapshot of data until after the first transaction has released its lock.

The first sentence is reasonable. Everything else means absolutely and totally *nothing* to me, and I'm sitting here writing this white paper about `MVCC`. If you're reading it and you don't understand what it means, it's not you - do not ascribe this to how much you know or do not know about `MVCC` - it simply *has no meaning*.

The last sentence is particularly mind-boggling - "the data read under the first lock will not be stale, because the second lock, where we use this data, is taken *after* the first lock".

Huh?

Surely anything could happen in-between the two and then the data *is* stale? why would taking the second lock *later* mean the data was guaranteed not to be stale? surely it means the exact opposite?

But - as you may have read me write elsewhere - I am absolutely certain no technical staff ever review the documentation, so no one ever catches stuff like this. It just gets put out there as the official docs.

The most I ever took from the docs here was that a read lock was taken, during which the command read the table and figure out presumably which rows it would change, which never actually made sense to me as this is Big Data, so you could have an `UPDATE` changing billions of rows and so there's no way you can gather state about these rows in some sort of stage 1 and then use that information in stage 2, because there's just too much data. With Big Data, the table *is* the state, and you operate on it directly - but, whatever, okay, there's a read phase, which would naturally enough imply a read lock, and then there's a write phase, which implies a write lock (and, also, apparently, a second transaction).

So I never understood how this would work, because in-between the first transaction and its read lock, and the second transaction and its write lock, hell, the table could be have dropped, or any number of rows could have been changed - how could any state obtained in the first stage be of any possible use in the second stage?

As ever, making test cases and looking at what actually happens reveals a picture

you literally could never have imagined or predicted, and which is stranger than anything you could have anticipated, which just goes to show how useful the docs are.

Here's the output from the test case for `UPDATE` (`DELETE` behaves in the same way, so I won't give its results here as well);

1. c1 : xid 9349 : start transaction, issue alter table

xid	table	lock type	granted	start time
9349	table_1	AccessExclusiveLock	True	2021-12-03 18:38:31.164459
9349	table_1	AccessShareLock	True	2021-12-03 18:38:31.164459

So, here, connection #1, we start a transaction and issue an `alter table`. As expected, this gives us an `AccessExclusiveLock`, but unexpectedly also a `AccessShareLock`. I've no clue why - why bother when you've requested an exclusive lock? but there it is, and it doesn't affect the test case.

2. c2 : xid 9352 : start transaction, issue update

xid	table	lock type	granted	start time
9349	table_1	AccessExclusiveLock	True	2021-12-03 18:38:31.164459
9349	table_1	AccessShareLock	True	2021-12-03 18:38:31.164459
9352	table_1	ShareRowExclusiveLock	False	2021-12-03 18:38:31.480355

Now, connection 2, we start a transaction and issue an `UPDATE`. The exclusive lock in the open transaction on connection #1 is blocking all other locks, so we can see what the `UPDATE` asks for - and it's a write lock.

Now, something important here which is central to the test case, note the "start time" column. This is the time the transaction 9352 started, and it is `2021-12-03 18:38:31.480355`.

3. c3 : xid 9357 : start transaction, issue alter table

xid	table	lock type	granted	start time
9349	table_1	AccessExclusiveLock	True	2021-12-03 18:38:31.164459
9349	table_1	AccessShareLock	True	2021-12-03 18:38:31.164459
9352	table_1	ShareRowExclusiveLock	False	2021-12-03 18:38:31.480355

xid	table	lock type	granted	start time
9357	table_1	ShareRowExclusiveLock	False	2021-12-03 18:38:36.737849

Third connection, and we open another transaction, and issue another `alter table`.

As discussed just a little earlier, what's actually happening here is that a "ShareRowExclusiveLock" is requested, but it will convert to an "AccessExclusiveLock" when granted.

Anyways, although this isn't ideal, it will still be useful; the point of it was that if any further locks are requested, we'll get to see them, because they'll be queued up behind this exclusive lock. So all we'll actually get now, since it's a write lock, is that if any new write locks are issued, they'll be forced to queue, and we'll get to see them. If any new read locks are issued, they won't be blocked, because write locks do not block read locks.

4. c1 : xid 9349 : transaction committed

xid	table	lock type	granted	start time
9352	table_1	AccessShareLock	True	2021-12-03 18:38:47.008794
9352	table_1	ShareRowExclusiveLock	False	2021-12-03 18:38:47.008794
9357	table_1	ShareRowExclusiveLock	False	2021-12-03 18:38:36.737849

Now the pay-off.

The first transaction, which is the initial `alter table` query with its exclusive lock, is committed.

Now Redshift comes with the weird stuff.

First and most obvious, a `AccessShareLock` lock appeared out of nowhere on transaction 9352 (the `UPDATE` transaction). This presumably is for the initial read phase, but why wait till now to request that lock?

So, again, the list of queue locks is not truthful; there are locks which you do not seen being requested, which pop up out of nowhere.

Second, less obvious but much more profound, look at the start time on the write lock for transaction 9352 - it is now `2021-12-03 18:38:47.008794`. It used to be `2021-12-03 18:38:31.480355`!

This is completely off the map.

It means this transaction had a second write lock granted *which completely avoided the lock queue.*

So, first, on the face of it, the transaction was restarted - the transaction ID

has not changed, but the start time has changed - but that's a concept I've never come across before, that one transaction ID can represent two different transactions.

We can imagine this though is the "second transaction" mentioned in the docs (although of course "second transaction" has a more obvious meaning - two transactions - rather than meaning a completely novel concept never before seen in Redshift or mentioned anywhere in the docs, of a restarted transaction where the ID doesn't change).

Secondly, though, and this is the biggie - the write lock held by the transaction, even though the transaction has restarted, *is still granted*.

If the transaction was indeed restarted, then we can take it that the transaction, despite being restarted, retained its locks; or we must take it that the transaction, after being restarted, issued a new write lock request *which jumped the lock queue*, since it was not blocked by the pending write lock for the second `ALTER TABLE`, and was immediately granted.

Where Redshift serializes writes, there is for any single table only ever one granted write lock, and so the update command here has its write lock granted, then its transaction restarts, but in doing so retains that granted write lock and so in effect, it's exactly as if `DELETE` and `UPDATE` took a write lock and held it for the duration of their operation.

This is the mechanism by which the read and write phases of these commands ensure whatever state is obtained during the read phase remains valid for the write phase.

Transactions and Table Locks

Now, moving on, so far in what's been written, to keep things simple, so we can focus on the locks themselves - I've been writing in terms of *queries* taking locks

In fact, this is not the case.

In Postgres and in Redshift, and remember here that all queries are in a transaction, even if it's a single-query transaction which automatically completes when its query completes, it is the *transaction* which takes locks.

When a query performs tasks which require locks, the locks are held by the transaction. In the system tables, there's no record of which lock is held by which query; it's which lock is held by which transaction.

Now you may be wondering, well, okay, great, but what difference does this make?

The kicker is that transactions *accumulate* locks.

When a query is issued, it requests the locks it needed, and these, as they are taken, are held by the transaction and they are held until the transaction aborts or completes.

We can see this here, from the following test case;

```
begin;
```

```
select count(*) from table_1;
```

xid	table	lock type	granted	start time
9216	table_1	AccessShareLock	True	2021-12-03 18:38:02.949747

```
insert into table_1 ( column_1, column_2 ) values ( 5, 1 );
```

xid	table	lock type	granted	start time
9216	table_1	AccessShareLock	True	2021-12-03 18:38:02.949747
9216	table_1	ShareRowExclusiveLock	True	2021-12-03 18:38:02.949747

```
update table_1 set column_2 = 5 where column_2 = 1;
```

xid	table	lock type	granted	start time
9216	table_1	AccessShareLock	True	2021-12-03 18:38:02.949747
9216	table_1	ShareRowExclusiveLock	True	2021-12-03 18:38:02.949747

With the `UPDATE` we note nothing changed; this is because the transaction already holds the needed locks on `table_1`.

```
select count(*) from table_2;
```

xid	table	lock type	granted	start time
9216	table_1	AccessShareLock	True	2021-12-03 18:38:02.949747
9216	table_1	ShareRowExclusiveLock	True	2021-12-03 18:38:02.949747
9216	table_2	AccessShareLock	True	2021-12-03 18:38:02.949747

```
delete from table_2 where column_2 = 10;
```

xid	table	lock type	granted	start time
9216	table_1	AccessShareLock	True	2021-12-03 18:38:02.949747
9216	table_1	ShareRowExclusiveLock	True	2021-12-03 18:38:02.949747
9216	table_2	AccessShareLock	True	2021-12-03 18:38:02.949747

xid	table	lock type	granted	start time
9216	table_2	ShareRowExclusiveLock	True	2021-12-03 18:38:02.949747

```
abort;
```

Now, this is *important*, because it means the more locks a transaction takes, and the more tables it locks, and the longer the transaction lasts for, the more it tends to block other transactions - and in particular, exclusive locks held by a long-running transaction are death and ruin, because, of course, they completely block all other transactions from performing any work at all on the tables concerned.

You can do this very easily; open a transaction, and issue `ALTER TABLE` on some table. Leave the transaction open. You have now completely blocked all other transactions from doing any work at all on the table. Do Not Do This.

Serialization Isolation Failures

Now, finally, we get to the point of the entire white paper :-)

`MVCC` is a truly lovely piece of work, but there are situations where `MVCC` can't cope, and when `MVCC` cannot cope, queries have to be temporarily blocked or aborted.

When `MVCC` can't cope such that a query has to abort, that abort is called a ***serialization isolation failure***.

That's all they are - they are aborts caused by `MVCC` not being able to cope.

I think there are two basic situations with which `MVCC` cannot cope, one of which leads to a query being temporarily blocked (but not aborted) and the other which leads to a query being aborted, but there's *any* number of ways to invoke either of these two basic situations.

Blocking Failures

The first basic situation originates in that the each row can store only a single insert transaction ID and a single delete transaction ID. It is possible for two or more queries want to write their transaction ID to the same row *at the same time*, and then `MVCC`, as you can imagine, has a problem.

For example, imagine we have a table and a delete query is running, progressing through the table, setting its transaction ID into `deletexid` for some rows and it is about half-way and then another delete query begins to run, and it too begins to progress down the table - and it comes to the point where it wants to write *its* transaction ID into a row which *already* has its delete transaction ID set by the other delete query.

Uh oh. Problem.

So what happens is the second query is temporarily blocked; because it now has to find out the fate of the first query, so it can know what it itself will do.

When we talk about the fate of a query, what we really mean to say is whether it aborts or completes successfully; after all, *all* queries end up in one of these two states.

It's not very likely, but it is possible, that the first query will abort.

If it does, then all its changes are ignored by all queries - and so now, the second query can go ahead and write its transaction ID, because the transaction ID which is already there, from the aborted first query, no longer has any meaning or validity, and this solves the problem `MVCC` faced.

However, it is much more likely the first query will successfully complete.

In this particular case, where we're deleting, when the first query completes what it will mean is that the row the second query is trying to write its transaction ID has now in fact *already been deleted*.

So, what happens?

Well, to explain, I now have to draw you back to the beginning of all, where we decided that we wanted a query to see all changes by other queries, or none, and that the means to achieve this was to ensure a query would use and only use changes made by queries which completed before the query.

What matters here of course is the goal, not the means; as long as we do achieve that goal, then we're fine.

So it turns out in this situation, with the two delete queries, this goal is in fact achieved by another means; blocking.

Consider that we have the two queries, where the first query has been making changes and has progressed some way down the table and now the second query has bumped into a change made by the first query, and has been blocked.

Of all the changes made by the first query, the second query will care about those where its `WHERE` clause leads it to try to delete the same rows, and the very first time the second query discovers a change by made the first query, it blocks - and by blocking, it waits until the first query completes.

In other words, the second query, because it blocked on the first change, and blocked until the first query completed, *will* see all the changes made by the first query.

The upshot then of all this is that the first query completes successfully, and so the second query now looks at the row it has been blocked on, and because the change to it (its deletion) was performed by the first query, which *in effect* completed successfully *before* the second query, considers the row already deleted, and by an earlier transaction ID (the first query started first) and so *ignores it*.

Accordingly, the second query now continues progressing down the table and completes successfully.

We can see all of this in the following test case;

Timestamp	Event
0.0	make connections to database

Timestamp	Event
0.7049508094787598	connections made
0.7049534320831299	create and populate test table
1.7877998352050781	test table created and populated
1.7878038883209229	connection 1 : start transaction and issue delete query
1.922311782836914	connection 1 : delete query completed
1.9223129749298096	connection 2 : start transaction and issue identical delete query
1.9994795322418213	begin sleep for 10 seconds to show connection 2 is blocked
12.006201982498169	sleep completed
12.006211996078491	connection 1 : commit
12.126752853393555	connection 2 : delete query completed
12.176802158355713	disconnect connections from database
12.177324771881104	disconnections complete

The test makes a table and puts a few rows into it.

Two connections are made to the database.

The first connection starts a transaction and deletes some rows, thus obtaining a transaction ID and writing it into the delete transaction ID for those rows. The transaction stays open.

The second connection now starts a transaction and this will of course come to have a later transaction ID.

Using this second transaction, another delete query is issued, and it turns out this query is trying to delete at least one row which was already deleted by the first transaction and its delete query.

The database knows this because there's only one delete transaction ID which can be stored for a row, so when second transaction comes to write the delete transaction ID for a row which has already been deleted by the first transaction, the database can see the delete transaction ID is already set (and by a transaction which is still running).

(Here both of transactions can see the same rows because the insert transaction of the rows is earlier than both the first and second transactions).

At this point then second transaction blocks. We can see this in the test as we wait for 10 seconds, and nothing happens.

After the 10 second pause, the first transaction is committed, and then the blocked delete query is unblocked and completes successfully.

Aborting Failures

We can now consider the second basic situation, which leads to an abort occurring.

As an aside, it's much easier to use multi-query transactions than single-query transactions to invent situations where `MVCC` fails. Single-query transactions begin, run their query, and complete, where-as multi-query transactions are opened, given a transaction ID, and then stay open until we choose to complete them. As such multi-query transactions offer far more control over the ordering of the queries being issued, which makes it much easier to write test cases where events happen in the order necessary for `MVCC` to look deeply hurt, find a corner somewhere and have a bit of a cry.

Nevertheless, all failures can occur perfectly well with single-query transactions as with multi-query, but with single-query transactions for failures to occur the timing has to be just right, which makes such failures, on the whole, inherently less likely, and of course, entirely unsuitable for test cases, which is why all the test cases use transactions.

With Transactions So, let's begin with a first test case, showing an aborting failure from issuing `DELETE` queries.

Timestamp	Event
0.0	make connections to database
0.7069826126098633	connections made
0.7069878578186035	create and populate test table
1.7681422233581543	test table created and populated
1.768148422241211	connection 1 : start transaction and issue select query
2.0498664379119873	connection 1 : select query completed (transaction ID 9426)
2.0498735904693604	connection 2 : start transaction issue delete query
2.4280898571014404	connection 2 : delete query completed and transaction committed (transaction ID 9428)
2.428119659423828	connection 1 : issue identical delete query
2.47615122795105	connection 1 : transaction aborts (expected transaction IDs 9426,9428 actual 9426,9428)
2.5143280029296875	disconnect connections from database
2.514991044998169	disconnections complete

The test makes a table and puts a few rows into it.

Two connections are made to the database.

The first connection starts a transaction and performs a select so it picks up a transaction ID, and for now does nothing else; the transaction is left open.

The second connection now starts a transaction and issues a delete query. Note the second transaction (naturally) has a later transaction ID than the first transaction. The delete query completes and the transaction is then committed.

At this point we have only the first transaction open; no other transactions are now running.

The first transaction now issues a delete query, and this query is such that it tries to delete a row which has already been deleted by the second transaction (and for good measure, if the query continued past this row, it would also want to delete further rows in the table which have already been deleted by the second transaction).

Now we have a problem.

The delete query wants to delete a row which has already been deleted.

Thinking about the situation, the first transaction at this point has two options; overwrite the `deletexid` or leave the row as it is.

(Blocking, which we saw before, is not an option, because the second transaction has already completed; there's nothing to wait for - we already know the second transaction completed successfully.)

It's not possible, in fact, for the first transaction to overwrite the delete ID of the row; the rows already have the delete transaction ID of the second transaction and that second transaction has already complete successfully - and so other queries can already be using the changes made by the second transaction.

If we now change the delete transaction ID, it may be other queries have *already* used that row, who would if we made that change then *not* have used it, and we somehow have to go and get them to "unread" what they've already read.

(We might think to ourselves, well, the first transaction started earlier, and has the earlier transaction ID, so it should take precedence and it *should* overwrite `deletexid` - and that on the face of it is entirely logical - but then you would have to solve the "unreading" problem, and that's too horrible to think about. Anyways, as you will now see, there are additional problems, which also make overwriting a no-no.)

A second problem is what happens if the first transaction goes on to abort?

Then we have a bunch of rows which originally were deleted by the second transaction, which now have the delete transaction ID of the first transaction, but the first transaction has now aborted and this means queries are going to *ignored* the delete transaction ID because they can see the query that ID came from has aborted!

In other words, if the first query aborted, it would cause the rows where it had overwritten the delete transaction ID to be undeleted.

So we can't just blunder in there with our size elevens and spray-can our delete transaction ID over a delete transaction ID written by a query that has already successfully completed.

That leaves ignoring the row.

Let's think about what that means : it means of the rows the first query has been instructed to delete, some have been deleted with its transaction ID, but some - those already deleted by the later, second transaction - will have been deleted with the transaction ID of the second transaction.

That's not going to fly - it breaks the principle that either all or none of the changes made show up, and we can see why that's a good principle; it's impossible to reason about the state of a table when that kind of behaviour can occur (indeed, how does the database even indicate to the user that such an event has occurred, and provide the information they would need to be able to reason?)

So now what?

We can't overwrite, we can't ignore, what else is there?

Well, for a database, there's is *always* one other option. Ultima ratio regum. The transaction aborts.

Now one more test case, showing an aborting failure from issuing `UPDATE` queries.

Timestamp	Event
0.0	make connections to database
0.6945226192474365	connections made
0.6945266723632812	create and populate test table
1.7525432109832764	test table created and populated
1.7525529861450195	connection 1 : start transaction and issue select query
2.012314558029175	connection 1 : select query completed (transaction ID 9448)
2.012331247329712	connection 2 : start transaction issue update query
2.410795211791992	connection 2 : update query completed and transaction committed (transaction ID 9450)
2.4108104705810547	connection 1 : issue update query, same rows, different set value
2.4582180976867676	connection 1 : transaction aborts (expected transaction IDs 9448,9450 actual 9448,9450)
2.496267080307007	disconnect connections from database
2.496717929840088	disconnections complete

Example number three - now with `UPDATE`, which is offers infinitely more possibilities for havoc.

I have a table. It contains some rows, and there are no running queries or transactions.

I start a transaction. To start with, I issue a `SELECT` so I have a transaction ID.

I start a second transaction. I issue an `UPDATE`, and change the values of some columns in some rows - which means I've set their delete transaction ID and made a new record, with my transaction ID, with the new values.

This transaction is then committed, leaving us with *only* the first transaction, and this an important point; I've seen people get confused about serialization failures when they only have one transaction open because they imagine you

need two or more *running* transactions for this to occur. This is, as I hope by now my efforts in writing out this not inconsiderable document have made clear, is not the case =-)

I return to the first transaction and issue an `UPDATE`, and I affect at least some of the rows updated by the second transaction, but where my `SET` can and so often or usually will differ from the second transaction, the values I want to write are not identical - so now there's a problem, because some of the rows I want to change *have already been changed*, by the second transaction. I can see the row - the insert generation is before mine, and the delete is after (since the second transaction has a later generation) but I can see the delete transaction ID has already been set, and by a transaction which is still in-flight.

Now, you can see this is a problem - someone else has already written a new row for this record. I can't write *another* new row, with different values!

So what happens now - as ever - is that the first transaction blocks, in the hope the second will abort. When the second transaction does not abort, and so completes successfully, then the first transaction has to abort, because it can't make changes to a row it can see, because the second transaction has already changed the row; if the first transaction changed it now, it would be overwriting a *later* change with its *earlier* change ("later" and "earlier" here are in terms of transaction IDs, because that's what defines ordering in the database), which is totally not okay. This is a database, not a Tardis :-)

Without Transactions This is the same scenario as above, but without transactions. As such, there is no test case, as it depends on a race condition.

Imagine a delete query begins. This passes down a table, but as it happens not yet deleting rows. It gets about halfway, where the first query is by the chance of processor time allocation for a little while pretty much idle; and continues (the first query still idle for now) and then it finally begins to delete some rows. It processes some rows, and then it too happens to become pretty much idle.

At this point, the first query comes back to life; it continues working, but then it tries to delete one of the rows already deleted by the second query.

Now we have a problem.

At this point, the first query can of course see the row it wants to delete (for both delete queries, the row was written by a query which completed successfully before the delete queries started), and it can see the delete generation has been set, and it can also see the delete generation has been set by a query which is still running.

Now, the first query cannot simply overwrite the delete generation in the row, on the basis it has an earlier generation number and so happened first; the problem is if you do this, what happens if what happens next is that the second query completes and then the first query goes on to abort?

The second query knows nothing about what the first query is getting up to, and so then when the first query aborts, we have a *major* problem; a row the second query would have and should have deleted is now in fact *not* deleted,

because the delete generation in that row is from the first query, which aborted! (and so where it aborted the delete will be ignored by other queries).

So what actually happens now is that the first query blocks, because it might be the second query aborts; the first query can't know what it will do until it knows the fate of the second query.

If the second query does abort, then all the delete generations it has set becomes meaningless, and the first query writes its delete generation and continues with its progress down the table.

However, if the second query goes on to complete successfully, the first query is now deader than A-line flares with pockets in the knees; it aborts.

The problem is that at this point, because the second query completes successfully, queries starting from now on will use the deletes it has performed. We could argue the first query should now set the delete generation in the row it has been blocked on, because the first query started first *but* what do we then do if as the first query then continues progressing through the table it finds *more* rows from the second query which is also wants to delete?

We *cannot* have the first query change these rows, because by now, *other* queries may have started, be running, and already be using those rows. If we changed the delete generation in those rows, we would once more be in the situation where running queries could have to "unread" rows they had already read - we're back to CASE NIGHTMARE GREEN.

The basic problem is that the first query is in an impossible situation; it wants to delete a row, where the row has been deleted by a query which has completed successfully but which started *after* the first query; according to what we've so far decided about how the database should operate, the first query should only pay attention to changes made by queries which complete successfully before it started, so it should not pay attention to this delete - but the delete is already visible and has been seen and potentially used by other queries, so we *can't* make change to it.

Result? we abort. What else can we do? we certainly can't carry on.

Summary

What we see here is that there are situations, and as we can begin to sense, any number of situations with all sorts of events in all sorts of orders, where `MVCC` *can't cope*, and when that happens sometimes a query will survive - it will be able to block, and then it discovers it's okay and can continue - but sometimes a query will be forced to abort.

Note Regarding Official Docs

Having then now read this document and come to its end, I would like to point you at the documentation provided by AWS on the matter of serialization isolation failure.

I am of the view this document is much like a Wikipedia page on a technical subject; it does not explain, and you can only understand what the text is

talking about if you *already know*.

I mention this because I suspect a lot of people have read this page and been left utterly bemused. I was, when I read it before I sat down and learned how `MVCC` works, and now I do understand, I think this is page is utterly incomprehensible. It explains everything in terms which only make sense if you *already* understand `MVCC`.

So, in short, *it's not you*.

Conclusions

I picked up COVID about six days ago. I'm fine - no problems breathing - but I suspect at the moment where my body is busy making antibodies and so on, my ability to focus is reduced and so I've spent about five hours on it today and got like about 15% of the way through =-) in other words, it's not quite happening, not right now.

Part of the problem is I think I'm not actually focusing as well as I think I am; the other part of the problem is that the amount material covered is large, I've been reworking it constantly for weeks, and so I no longer have in my mind a concise grasp of the white paper and the progression of material - which you absolutely must have to write conclusions.

I have then as a practical matter elected in this case to publish now, in part because I think currently I cannot meaningfully work any more on this white paper without a break, and in part as I think COVID probably will take a week or two more to pass.

Do be clear though that the Discussion, which is the main part of the work, has received a *lot* of work, re-writing, re-organization and development; in particular, the earlier the material, the more it has been improved. I am completely happy for it to be published and I am of the view it is highly effective and viable, and it took many weekends of re-writing work to get to that point; the main substance of the white paper is absolutely ready and fit for purpose.

Revision History

v1

- Initial release.

v2

- Doc bug spotted by Albert Xu. The "Without Transactions" subsection of "Blocking Failures" was slightly gibberish - too many edits and not realizing the final result had gone off the rails a bit; now corrected.

v3

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

Appendix A : Test Method

There are three broad sets of tests. The first demonstrates the basics of `MVCC`. The second proves a number of important miscellaneous proofs, such as rows from aborted queries remaining in a table until `VACUUM` is run, that transactions accumulate locks, and so on. The third demonstrates serialization isolation failures.

The test results are given in full, without discussion and so simply for easy reference, in the Results section. The test results are given, with discussion and at the appropriate junctures during the explanation of `MVCC`, in the Discussion.

With regard to the first set of tests, there are certain difficulties testing because the two system managed columns in each table, for the insert and delete transaction IDs, in Redshift cannot be accessed using `SELECT` (unlike Postgres), which on the face of it makes it impossible to know their values and so to write test cases which use them.

However, there is one method by which you can obtain some information from these columns, which is to use `STV_BLOCKLIST`. This pseudo-table (`STV` tables are table-like representations of Redshift's actual internal state) records the minimum and maximum value in every column for every block (1mb of data).

So, we make a new table, or truncate an existing table, and insert a single row.

We can then examine `STV_BLOCKLIST` and see the minimum and maximum value for the first (and only) block for the `insertxid` and `deletexid` columns, and so we can know the value of those columns for that single row.

We can also often usefully insert one more row, since with two different values in a column, we can see both if they are different (the minimum and maximum values for the block will differ) or if they are the same (the minimum and maximum will be identical).

This method then is used for the first set of tests.

The second and third sets of tests do not need to access these columns, and need no special handling.

Basics of `MVCC`

First is a test case a sequence of steps to demonstrate the behaviour of the `insertxid` and `deletexid` system-managed columns which are present in every table.

1. A new table is created and a single row is inserted. We then examine the system tables to get the transaction ID for the insert query, and then examine `STV_BLOCKLIST` to find the minimum and maximum values for the insert and delete columns.
2. We then insert a second row into the table, and again find the transaction ID for the query and examine the minimum and maximum values for the block, for the insert and delete columns.
3. We then delete the first row.

4. We then run `VACUUM FULL`.

5. We now issue an `UPDATE`, changing the value of the single row in the table.

In-between each step, we predict and then obtain the values of the `deletexid` and `insertxid`.

Miscellaneous Proofs

1. Transactions Accumulate Locks

Two test tables are created. A transaction is started. A `SELECT`, `INSERT` and `UPDATE` are issued on the first test table, followed by a `SELECT` and `DELETE` on the second test table. The list of locks held by the transaction is shown after each query, demonstrating that the transaction accumulated locks.

2. Aborted Rows Remain Until VACUUMed

It is possible to attempt to determine the number of deleted rows in a table. This is done by using `STV_BLOCKLIST`, which records the number of rows in each block for each column - which is to say, it counts *all* rows, regardless of `MVCC` status. This allows us to obtain the total number of rows, and `SELECT COUNT(*)` allows us to obtain the number of rows usable by the current transaction. On an idle table, the difference between the two is the number of deleted rows.

A test table is created. A transaction is started and a few rows inserted. The transaction is then aborted. A `VACUUM FULL` is then issued. In-between each step in the test, we record the total number of rows and the usable number of rows, where the numbers we obtain demonstrate that aborted rows remain in a table until a `VACUUM` is issued.

3. VACUUM Removes Only Unusable Rows

Two test tables are created, both are populated with one hundred rows.

A first transaction is created, which issues a `SELECT COUNT(*)` on the first test table, to obtain a transaction ID. This transaction is then left open. Note the first table is after this not touched and this is an intentional and central part of the test design, as will become clear.

A second transaction is created, which issues a `DELETE` on the second table, deleting every row, with the transaction being left open. A `VACUUM FULL` is issued on the second table. The second transaction is then committed. Then the first transaction is finally committed, and next, finally, another `VACUUM FULL` is issued on the second table.

In-between each step in the test, we examine `STV_BLOCKLIST` to obtain the total number of rows in the table. We see from the results that the deleted rows remain in the table until the `VACUUM` is issued after the *first* transaction is committed - even though the first transaction did not touch the second table at all.

4. UPDATE Locking Behaviour

A test table is created and populated with a few rows.

A first transaction is opened and this issued an `ALTER TABLE`. This leaves the first transaction holding an exclusive lock on the table, so all other locks will be queued.

A second transaction is opened, and this issues an `UPDATE`. This obtains a write lock, which is queued because of the existing held exclusive lock.

A third transaction is opened, and this issues a further `ALTER TABLE`. This turns out to take a write lock, which will turn into an exclusive lock when it is granted.

The first transaction is now committed.

Between each step in the test, we examine the locks held on the table, and we see from these the locking behaviour of `UPDATE`.

Serialization Isolation Failures

1. Transaction Blocking With DELETE

A test table is created and populated with a few rows.

A first transaction is opened, and this issues a `DELETE`. The transaction is left open.

A second transaction is opened, and this issue an identical `DELETE`. This transaction will block, pending the fate of the first transaction, as it is attempting to delete a row which has already been deleted.

The first transaction is committed, and we now see the second transaction also completes.

2. Transaction Aborting With DELETE

A test table is created and populated with a few rows.

A first transaction is opened, which issues a `SELECT` to obtain a transaction ID. The transaction is left open.

A second transaction is opened, which begins a transaction, issues a `DELETE`, and commits.

The first transaction now issues an identical `DELETE` (to ensure it attempts to delete at least row deleted by the second transaction `DELETE`).

The first transaction now aborts.

3. Transaction Aborting With UPDATE

A test table is created and populated with a few rows.

A first transaction is opened, which issues a `SELECT` to obtain a transaction ID. The transaction is left open.

A second transaction is opened, which issues an `UPDATE`, and then commits. This leaves us with only the first transaction.

The first transaction now issues an identical `UPDATE` (to ensure it attempts to update at least row updated by the second transaction `UPDATE`).

The first transaction now aborts.

Appendix B : Results

The results are given here for ease of reference, but they are primarily presented, piece by piece along with explanation, in the Discussion.

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

The script used to generated these results in designed for readers to use, and is available here.

Test duration, excluding server bring-up and shut-down, was 82 seconds.

dc2.large, 2 nodes (1.0.33759)

insertxid and deletexid

1. create test table
2. insert one row
3. query transaction_id = 9176

Value	Expected	Actual
`insertxid` (min)	9176	9176
`insertxid` (max)	9176	9176
`deletexid` (min)	9223372036854775807	9223372036854775807
`deletexid` (max)	9223372036854775807	9223372036854775807

4. insert second row
5. query transaction_id = 9179

Value	Expected	Actual
`insertxid` (min)	9176	9176
`insertxid` (max)	9179	9179
`deletexid` (min)	9223372036854775807	9223372036854775807
`deletexid` (max)	9223372036854775807	9223372036854775807

6. delete first row
7. query transaction_id = 9182

Value	Expected	Actual
`insertxid` (min)	9176	9176
`insertxid` (max)	9179	9179
`deletexid` (min)	9182	9182
`deletexid` (max)	9223372036854775807	9223372036854775807

8. VACUUM table

Value	Expected	Actual
`insertxid` (min)	9179	9179
`insertxid` (max)	9179	9179
`deletexid` (min)	9223372036854775807	9223372036854775807
`deletexid` (max)	9223372036854775807	9223372036854775807

9. update all rows (which means one row)
10. query transaction_id = 9189

Value	Expected	Actual
`insertxid` (min)	9179	9179
`insertxid` (max)	9189	9189
`deletexid` (min)	9189	9189
`deletexid` (max)	9223372036854775807	9223372036854775807

Transactions Accumulate Locks

```
begin;
```

```
select count(*) from table_1;
```

xid	table	lock type	granted	start time
9216	table_1	AccessShareLock	True	2021-12-03 18:38:02.949747

```
insert into table_1 ( column_1, column_2 ) values ( 5, 1 );
```

xid	table	lock type	granted	start time
9216	table_1	AccessShareLock	True	2021-12-03 18:38:02.949747
9216	table_1	ShareRowExclusiveLock	True	2021-12-03 18:38:02.949747

```
update table_1 set column_2 = 5 where column_2 = 1;
```

xid	table	lock type	granted	start time
9216	table_1	AccessShareLock	True	2021-12-03 18:38:02.949747
9216	table_1	ShareRowExclusiveLock	True	2021-12-03 18:38:02.949747

```
select count(*) from table_2;
```

xid	table	lock type	granted	start time
9216	table_1	AccessShareLock	True	2021-12-03 18:38:02.949747
9216	table_1	ShareRowExclusiveLock	True	2021-12-03 18:38:02.949747
9216	table_2	AccessShareLock	True	2021-12-03 18:38:02.949747

```
delete from table_2 where column_2 = 10;
```

xid	table	lock type	granted	start time
9216	table_1	AccessShareLock	True	2021-12-03 18:38:02.949747
9216	table_1	ShareRowExclusiveLock	True	2021-12-03 18:38:02.949747
9216	table_2	AccessShareLock	True	2021-12-03 18:38:02.949747
9216	table_2	ShareRowExclusiveLock	True	2021-12-03 18:38:02.949747

```
abort;
```

DELETE Locking Behaviour

1. c1 : xid 9309 : start transaction, issue alter table

xid	table	lock type	granted	start time
9309	table_1	AccessExclusiveLock	True	2021-12-03 18:38:12.569097
9309	table_1	AccessShareLock	True	2021-12-03 18:38:12.569097

2. c2 : xid 9312 : start transaction, issue delete

xid	table	lock type	granted	start time
9309	table_1	AccessExclusiveLock	True	2021-12-03 18:38:12.569097
9309	table_1	AccessShareLock	True	2021-12-03 18:38:12.569097
9312	table_1	ShareRowExclusiveLock	False	2021-12-03 18:38:12.855109

3. c3 : xid 9314 : start transaction, issue alter table

xid	table	lock type	granted	start time
9309	table_1	AccessExclusiveLock	True	2021-12-03 18:38:12.569097
9309	table_1	AccessShareLock	True	2021-12-03 18:38:12.569097
9312	table_1	ShareRowExclusiveLock	False	2021-12-03 18:38:12.855109
9314	table_1	ShareRowExclusiveLock	False	2021-12-03 18:38:18.051012

4. c1 : xid 9309 : transaction committed

xid	table	lock type	granted	start time
9312	table_1	AccessShareLock	True	2021-12-03 18:38:28.333158
9312	table_1	ShareRowExclusiveLock	True	2021-12-03 18:38:28.333158
9314	table_1	ShareRowExclusiveLock	False	2021-12-03 18:38:18.051012

UPDATE Locking Behaviour

1. c1 : xid 9349 : start transaction, issue alter table

xid	table	lock type	granted	start time
9349	table_1	AccessExclusiveLock	True	2021-12-03 18:38:31.164459
9349	table_1	AccessShareLock	True	2021-12-03 18:38:31.164459

2. c2 : xid 9352 : start transaction, issue update

xid	table	lock type	granted	start time
9349	table_1	AccessExclusiveLock	True	2021-12-03 18:38:31.164459
9349	table_1	AccessShareLock	True	2021-12-03 18:38:31.164459
9352	table_1	ShareRowExclusiveLock	False	2021-12-03 18:38:31.480355

3. c3 : xid 9357 : start transaction, issue alter table

xid	table	lock type	granted	start time
9349	table_1	AccessExclusiveLock	True	2021-12-03 18:38:31.164459
9349	table_1	AccessShareLock	True	2021-12-03 18:38:31.164459
9352	table_1	ShareRowExclusiveLock	False	2021-12-03 18:38:31.480355
9357	table_1	ShareRowExclusiveLock	False	2021-12-03 18:38:36.737849

4. c1 : xid 9349 : transaction committed

xid	table	lock type	granted	start time
9352	table_1	AccessShareLock	True	2021-12-03 18:38:47.008794
9352	table_1	ShareRowExclusiveLock	True	2021-12-03 18:38:47.008794
9357	table_1	ShareRowExclusiveLock	False	2021-12-03 18:38:36.737849

Aborted Rows Remain Until VACUUMed

Timestamp	Event
0.0	make connection to database
0.35033321380615234	connection made
0.35033512115478516	create empty test table
0.6810824871063232	empty test table created
0.7387111186981201	connection 1 : select count(*) from table_1 = 0
0.8353245258331299	connection 1 : STV_BLOCKLIST number rows in table_1 = 0
0.835395097732544	connection 1 : start transaction and issue insert query on `table_1`
0.9689481258392334	connection 1 : insert query complete, rows inserted
1.023545742034912	connection 1 : select count(*) from table_1 = 3
1.1150462627410889	connection 1 : STV_BLOCKLIST number rows in table_1 = 3
1.1150541305541992	connection 1 : abort transaction
1.1560404300689697	connection 1 : abort complete
1.5720436573028564	connection 1 : select count(*) from table_1 = 0
1.700000286102295	connection 1 : STV_BLOCKLIST number rows in table_1 = 3
1.7000102996826172	connection 1 : vacuum full `table_1` to 100 percent

Timestamp	Event
1.9459662437438965	connection 1 : vacuum complete
2.0201823711395264	connection 1 : select count(*) from table_1 = 0
2.1518771648406982	connection 1 : STV_BLOCKLIST number rows in table_1 = 0
2.1518847942352295	disconnect connection from database
2.152294635772705	disconnection complete

VACUUM Removes Only Unusable Rows

Timestamp	Event
0.0	make connections to database
0.7071206569671631	connections made
0.7071232795715332	create and populate test tables
2.9194986820220947	test tables created and populated, 100 rows each
2.9195001125335693	connection 1 : start transaction and issue select query on `table_1`
3.0636632442474365	connection 1 : select query completed, transaction left open
3.0636658668518066	connection 2 : issue delete query on `table_2`
3.1823501586914062	connection 2 : delete query complete
3.2959678173065186	connection 2 : `table_2` has 100 deleted rows
3.2959771156311035	connection 2 : vacuum full `table_2` to 100 percent
3.685190200805664	connection 2 : vacuum complete
3.77941632270813	connection 2 : `table_2` has 100 deleted rows
3.779421806335449	connection 1 : commit transaction
3.8184521198272705	connection 1 : commit complete
3.9197051525115967	connection 2 : `table_2` has 100 deleted rows
3.9197168350219727	connection 2 : vacuum full `table_2` to 100 percent
4.239577054977417	connection 2 : vacuum complete
4.33547568321228	connection 2 : `table_2` has 0 deleted rows
4.335484981536865	disconnect connections from database
4.3360888957977295	disconnections complete

Transaction Blocking With DELETE

Timestamp	Event
0.0	make connections to database
0.7049508094787598	connections made
0.7049534320831299	create and populate test table
1.7877998352050781	test table created and populated
1.7878038883209229	connection 1 : start transaction and issue delete query
1.922311782836914	connection 1 : delete query completed
1.9223129749298096	connection 2 : start transaction and issue identical delete query
1.9994795322418213	begin sleep for 10 seconds to show connection 2 is blocked
12.006201982498169	sleep completed
12.006211996078491	connection 1 : commit
12.126752853393555	connection 2 : delete query completed
12.176802158355713	disconnect connections from database
12.177324771881104	disconnections complete

Transaction Aborting With DELETE

Timestamp	Event
0.0	make connections to database
0.7069826126098633	connections made
0.7069878578186035	create and populate test table
1.7681422233581543	test table created and populated
1.768148422241211	connection 1 : start transaction and issue select query
2.0498664379119873	connection 1 : select query completed (transaction ID 9426)
2.0498735904693604	connection 2 : start transaction issue delete query
2.4280898571014404	connection 2 : delete query completed and transaction committed (transaction ID 9428)
2.428119659423828	connection 1 : issue identical delete query
2.47615122795105	connection 1 : transaction aborts (expected transactions ID 9426,9428 actual 9426,9428)
2.5143280029296875	disconnect connections from database
2.514991044998169	disconnections complete

Transaction Aborting With UPDATE

Timestamp	Event
0.0	make connections to database
0.6945226192474365	connections made
0.6945266723632812	create and populate test table
1.7525432109832764	test table created and populated
1.7525529861450195	connection 1 : start transaction and issue select query
2.012314558029175	connection 1 : select query completed (transaction ID 9448)
2.012331247329712	connection 2 : start transaction issue update query
2.410795211791992	connection 2 : update query completed and transaction committed (transaction ID 9450)
2.4108104705810547	connection 1 : issue update query, same rows, different set value
2.4582180976867676	connection 1 : transaction aborts (expected transactions ID 9448,9450 actual 9448,9450)
2.496267080307007	disconnect connections from database
2.496717929840088	disconnections complete

Appendix C : Raw Data Dump

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


```
{'proofs': {'dc2.large': {2: {'log_type_and_datum': {'DELETE Locking Behaviour': [('message',
                                                                                   'c1 '
                                                                                   ': '
                                                                                   'xid '
                                                                                   '9309 '
                                                                                   ': '
                                                                                   'start '
                                                                                   'transaction, '
                                                                                   'issue '
                                                                                   'alter '
                                                                                   'table'),
                                                                                  ('locks',
                                                                                   [[9309,
                                                                                     'public',
                                                                                     'table_1',
                                                                                     'AccessExclusiveLock',
                                                                                     True,
                                                                                     datetime.datetime(2021, 12, 3, 18, 38, 12, 569097)],
                                                                                    [9309,
                                                                                     'public',
                                                                                     'table_1',
                                                                                     'AccessShareLock',
                                                                                     True,
                                                                                     datetime.datetime(2021, 12, 3, 18, 38, 12, 569097)]]),
                                                                                  ('message',
                                                                                   'c2 '
                                                                                   ': '
                                                                                   'xid '
                                                                                   '9312 '
                                                                                   ': '
                                                                                   'start '
                                                                                   'transaction, '
                                                                                   'issue '
                                                                                   'delete'),
                                                                                  ('locks',
                                                                                   [[9309,
                                                                                     'public',
                                                                                     'table_1',
                                                                                     'AccessExclusiveLock',
                                                                                     True,
                                                                                     datetime.datetime(2021, 12, 3, 18, 38, 12, 569097)],
                                                                                    [9309,
                                                                                     'public',
```

```
                                                  'table_1',
                                                  'AccessShareLock',
                                                  True,
                                                  datetime.datetime(2021, 12, 3, 18, 38, 12, 569097)],
                                                 [9312,
                                                  'public',
                                                  'table_1',
                                                  'ShareRowExclusiveLock',
                                                  False,
                                                  datetime.datetime(2021, 12, 3, 18, 38, 12, 855109)]]),
                                                ('message',
                                                 'c3 '
                                                 ': '
                                                 'xid '
                                                 '9314 '
                                                 ': '
                                                 'start '
                                                 'transaction, '
                                                 'issue '
                                                 'alter '
                                                 'table'),
                                                ('locks',
                                                 [[9309,
                                                   'public',
                                                   'table_1',
                                                   'AccessExclusiveLock',
                                                   True,
                                                   datetime.datetime(2021, 12, 3, 18, 38, 12, 569097)],
                                                  [9309,
                                                   'public',
                                                   'table_1',
                                                   'AccessShareLock',
                                                   True,
                                                   datetime.datetime(2021, 12, 3, 18, 38, 12, 569097)],
                                                  [9312,
                                                   'public',
                                                   'table_1',
                                                   'ShareRowExclusiveLock',
                                                   False,
                                                   datetime.datetime(2021, 12, 3, 18, 38, 12, 855109)],
                                                  [9314,
                                                   'public',
                                                   'table_1',
                                                   'ShareRowExclusiveLock',
                                                   False,
                                                   datetime.datetime(2021, 12, 3, 18, 38, 18, 51012)]]),
                                                ('message',
                                                 'c1 '
                                                 ': '
                                                 'xid '
                                                 '9309 '
                                                 ': '
                                                 'transaction '
                                                 'committed'),
                                                ('locks',
                                                 [[9312,
                                                   'public',
                                                   'table_1',
                                                   'AccessShareLock',
                                                   True,
                                                   datetime.datetime(2021, 12, 3, 18, 38, 28, 333158)],
                                                  [9312,
                                                   'public',
                                                   'table_1',
                                                   'ShareRowExclusiveLock',
                                                   True,
                                                   datetime.datetime(2021, 12, 3, 18, 38, 28, 333158)],
                                                  [9314,
                                                   'public',
                                                   'table_1',
                                                   'ShareRowExclusiveLock',
                                                   False,
                                                   datetime.datetime(2021, 12, 3, 18, 38, 18, 51012)]])],
 'Transactions Accumulate Locks': [('sql',
                                    'begin;'),
                                   ('sql',
                                    'select '
                                    'count(*) '
                                    'from '
                                    'table_1;'),
                                   ('locks',
                                    [[9216,
                                      'public',
                                      'table_1',
                                      'AccessShareLock',
                                      True,
                                      datetime.datetime(2021, 12, 3, 18, 38, 2, 949747)]]),
                                   ('sql',
                                    'insert '
                                    'into '
                                    'table_1 '
                                    '( '
                                    'column_1, '
                                    'column_2 '
                                    ') '
                                    'values '
                                    '( '
                                    '5, '
                                    '1 '
```

```
                                                 ');'),
                                                ('locks',
                                                 [[9216,
                                                   'public',
                                                   'table_1',
                                                   'AccessShareLock',
                                                   True,
                                                   datetime.datetime(2021, 12, 3, 18, 38, 2, 949747)],
                                                  [9216,
                                                   'public',
                                                   'table_1',
                                                   'ShareRowExclusiveLock',
                                                   True,
                                                   datetime.datetime(2021, 12, 3, 18, 38, 2, 949747)]]),
                                                ('sql',
                                                 'update '
                                                 'table_1 '
                                                 'set '
                                                 'column_2 '
                                                 '= '
                                                 '5 '
                                                 'where '
                                                 'column_2 '
                                                 '= '
                                                 '1;'),
                                                ('locks',
                                                 [[9216,
                                                   'public',
                                                   'table_1',
                                                   'AccessShareLock',
                                                   True,
                                                   datetime.datetime(2021, 12, 3, 18, 38, 2, 949747)],
                                                  [9216,
                                                   'public',
                                                   'table_1',
                                                   'ShareRowExclusiveLock',
                                                   True,
                                                   datetime.datetime(2021, 12, 3, 18, 38, 2, 949747)]]),
                                                ('sql',
                                                 'select '
                                                 'count(*) '
                                                 'from '
                                                 'table_2;'),
                                                ('locks',
                                                 [[9216,
                                                   'public',
                                                   'table_1',
                                                   'AccessShareLock',
                                                   True,
                                                   datetime.datetime(2021, 12, 3, 18, 38, 2, 949747)],
                                                  [9216,
                                                   'public',
                                                   'table_1',
                                                   'ShareRowExclusiveLock',
                                                   True,
                                                   datetime.datetime(2021, 12, 3, 18, 38, 2, 949747)],
                                                  [9216,
                                                   'public',
                                                   'table_2',
                                                   'AccessShareLock',
                                                   True,
                                                   datetime.datetime(2021, 12, 3, 18, 38, 2, 949747)]]),
                                                ('sql',
                                                 'delete '
                                                 'from '
                                                 'table_2 '
                                                 'where '
                                                 'column_2 '
                                                 '= '
                                                 '10;'),
                                                ('locks',
                                                 [[9216,
                                                   'public',
                                                   'table_1',
                                                   'AccessShareLock',
                                                   True,
                                                   datetime.datetime(2021, 12, 3, 18, 38, 2, 949747)],
                                                  [9216,
                                                   'public',
                                                   'table_1',
                                                   'ShareRowExclusiveLock',
                                                   True,
                                                   datetime.datetime(2021, 12, 3, 18, 38, 2, 949747)],
                                                  [9216,
                                                   'public',
                                                   'table_2',
                                                   'AccessShareLock',
                                                   True,
                                                   datetime.datetime(2021, 12, 3, 18, 38, 2, 949747)],
                                                  [9216,
                                                   'public',
                                                   'table_2',
                                                   'ShareRowExclusiveLock',
                                                   True,
                                                   datetime.datetime(2021, 12, 3, 18, 38, 2, 949747)]]),
                                                ('sql',
                                                 'abort;')],
 'UPDATE Locking Behaviour': [('message',
                               'c1 '
```

```
 ': '
 'xid '
 '9349 '
 ': '
 'start '
 'transaction, '
 'issue '
 'alter '
 'table'),
('locks',
 [[9349,
   'public',
   'table_1',
   'AccessExclusiveLock',
   True,
   datetime.datetime(2021, 12, 3, 18, 38, 31, 164459)],
  [9349,
   'public',
   'table_1',
   'AccessShareLock',
   True,
   datetime.datetime(2021, 12, 3, 18, 38, 31, 164459)]]),
('message',
 'c2 '
 ': '
 'xid '
 '9352 '
 ': '
 'start '
 'transaction, '
 'issue '
 'update'),
('locks',
 [[9349,
   'public',
   'table_1',
   'AccessExclusiveLock',
   True,
   datetime.datetime(2021, 12, 3, 18, 38, 31, 164459)],
  [9349,
   'public',
   'table_1',
   'AccessShareLock',
   True,
   datetime.datetime(2021, 12, 3, 18, 38, 31, 164459)],
  [9352,
   'public',
   'table_1',
   'ShareRowExclusiveLock',
   False,
   datetime.datetime(2021, 12, 3, 18, 38, 31, 480355)]]),
('message',
 'c3 '
 ': '
 'xid '
 '9357 '
 ': '
 'start '
 'transaction, '
 'issue '
 'alter '
 'table'),
('locks',
 [[9349,
   'public',
   'table_1',
   'AccessExclusiveLock',
   True,
   datetime.datetime(2021, 12, 3, 18, 38, 31, 164459)],
  [9349,
   'public',
   'table_1',
   'AccessShareLock',
   True,
   datetime.datetime(2021, 12, 3, 18, 38, 31, 164459)],
  [9352,
   'public',
   'table_1',
   'ShareRowExclusiveLock',
   False,
   datetime.datetime(2021, 12, 3, 18, 38, 31, 480355)],
  [9357,
   'public',
   'table_1',
   'ShareRowExclusiveLock',
   False,
   datetime.datetime(2021, 12, 3, 18, 38, 36, 737849)]]),
('message',
 'c1 '
 ': '
 'xid '
 '9349 '
 ': '
 'transaction '
 'committed'),
('locks',
 [[9352,
   'public',
   'table_1',
```

```
                                          'AccessShareLock',
                                          True,
                                          datetime.datetime(2021, 12, 3, 18, 38, 47, 8794)],
                                         [9352,
                                          'public',
                                          'table_1',
                                          'ShareRowExclusiveLock',
                                          True,
                                          datetime.datetime(2021, 12, 3, 18, 38, 47, 8794)],
                                         [9357,
                                          'public',
                                          'table_1',
                                          'ShareRowExclusiveLock',
                                          False,
                                          datetime.datetime(2021, 12, 3, 18, 38, 36, 737849)]])],
'insertxid and deletexid': [('message',
                             'create '
                             'test '
                             'table'),
                            ('message',
                             'insert '
                             'one '
                             'row'),
                            ('message',
                             'query '
                             'transaction_id '
                             '= '
                             '9176'),
                            ('xid_columns',
                             {'actual': (9176,
                                         9176,
                                         9223372036854775807,
                                         9223372036854775807),
                              'expected': (9176,
                                           9176,
                                           9223372036854775807,
                                           9223372036854775807)}),
                            ('message',
                             'insert '
                             'second '
                             'row'),
                            ('message',
                             'query '
                             'transaction_id '
                             '= '
                             '9179'),
                            ('xid_columns',
                             {'actual': (9176,
                                         9179,
                                         9223372036854775807,
                                         9223372036854775807),
                              'expected': (9176,
                                           9179,
                                           9223372036854775807,
                                           9223372036854775807)}),
                            ('message',
                             'delete '
                             'first '
                             'row'),
                            ('message',
                             'query '
                             'transaction_id '
                             '= '
                             '9182'),
                            ('xid_columns',
                             {'actual': (9176,
                                         9179,
                                         9182,
                                         9223372036854775807),
                              'expected': (9176,
                                           9179,
                                           9182,
                                           9223372036854775807)}),
                            ('message',
                             'VACUUM '
                             'table'),
                            ('xid_columns',
                             {'actual': (9179,
                                         9179,
                                         9223372036854775807,
                                         9223372036854775807),
                              'expected': (9179,
                                           9179,
                                           9223372036854775807,
                                           9223372036854775807)}),
                            ('message',
                             'update '
                             'all '
                             'rows '
                             '(which '
                             'means '
                             'one '
                             'row)'),
                            ('message',
                             'query '
                             'transaction_id '
                             '= '
                             '9189'),
                            ('xid_columns',
```

```
                                                    {'actual': (9179,
                                                                9189,
                                                                9189,
                                                                9223372036854775807),
                                                     'expected': (9179,
                                                                  9189,
                                                                  9189,
                                                                  9223372036854775807)})]},
'timestamp_and_events': {'Aborted Rows Remain Until VACUUMed': [(1638556683.8436377,
                                                                 'make '
                                                                 'connection '
                                                                 'to '
                                                                 'database'),
                                                                (1638556684.193971,
                                                                 'connection '
                                                                 'made'),
                                                                (1638556684.1939728,
                                                                 'create '
                                                                 'empty '
                                                                 'test '
                                                                 'table'),
                                                                (1638556684.5247202,
                                                                 'empty '
                                                                 'test '
                                                                 'table '
                                                                 'created'),
                                                                (1638556684.5823488,
                                                                 'connection '
                                                                 '1 '
                                                                 ': '
                                                                 'select '
                                                                 'count(*) '
                                                                 'from '
                                                                 'table_1 '
                                                                 '= '
                                                                 '0'),
                                                                (1638556684.6789622,
                                                                 'connection '
                                                                 '1 '
                                                                 ': '
                                                                 'STV_BLOCKLIST '
                                                                 'number '
                                                                 'rows '
                                                                 'in '
                                                                 'table_1 '
                                                                 '= '
                                                                 '0'),
                                                                (1638556684.6790328,
                                                                 'connection '
                                                                 '1 '
                                                                 ': '
                                                                 'start '
                                                                 'transaction '
                                                                 'and '
                                                                 'issue '
                                                                 'insert '
                                                                 'query '
                                                                 'on '
                                                                 '`table_1`'),
                                                                (1638556684.8125858,
                                                                 'connection '
                                                                 '1 '
                                                                 ': '
                                                                 'insert '
                                                                 'query '
                                                                 'complete, '
                                                                 'rows '
                                                                 'inserted'),
                                                                (1638556684.8671834,
                                                                 'connection '
                                                                 '1 '
                                                                 ': '
                                                                 'select '
                                                                 'count(*) '
                                                                 'from '
                                                                 'table_1 '
                                                                 '= '
                                                                 '3'),
                                                                (1638556684.958684,
                                                                 'connection '
                                                                 '1 '
                                                                 ': '
                                                                 'STV_BLOCKLIST '
                                                                 'number '
                                                                 'rows '
                                                                 'in '
                                                                 'table_1 '
                                                                 '= '
                                                                 '3'),
                                                                (1638556684.9586918,
                                                                 'connection '
                                                                 '1 '
                                                                 ': '
                                                                 'abort '
                                                                 'transaction'),
                                                                (1638556684.9996781,
                                                                 'connection '
                                                                 '1 '
                                                                 ': '
```

```
                                                 'abort '
                                                 'complete'),
                                                (1638556685.4156814,
                                                 'connection '
                                                 '1 '
                                                 ': '
                                                 'select '
                                                 'count(*) '
                                                 'from '
                                                 'table_1 '
                                                 '= '
                                                 '0'),
                                                (1638556685.543638,
                                                 'connection '
                                                 '1 '
                                                 ': '
                                                 'STV_BLOCKLIST '
                                                 'number '
                                                 'rows '
                                                 'in '
                                                 'table_1 '
                                                 '= '
                                                 '3'),
                                                (1638556685.543648,
                                                 'connection '
                                                 '1 '
                                                 ': '
                                                 'vacuum '
                                                 'full '
                                                 '`table_1` '
                                                 'to '
                                                 '100 '
                                                 'percent'),
                                                (1638556685.789604,
                                                 'connection '
                                                 '1 '
                                                 ': '
                                                 'vacuum '
                                                 'complete'),
                                                (1638556685.86382,
                                                 'connection '
                                                 '1 '
                                                 ': '
                                                 'select '
                                                 'count(*) '
                                                 'from '
                                                 'table_1 '
                                                 '= '
                                                 '0'),
                                                (1638556685.9955149,
                                                 'connection '
                                                 '1 '
                                                 ': '
                                                 'STV_BLOCKLIST '
                                                 'number '
                                                 'rows '
                                                 'in '
                                                 'table_1 '
                                                 '= '
                                                 '0'),
                                                (1638556685.9955225,
                                                 'disconnect '
                                                 'connection '
                                                 'from '
                                                 'database'),
                                                (1638556685.9959323,
                                                 'disconnection '
                                                 'complete')],
'Transaction Aborting With DELETE': [(1638556739.7974184,
                                      'make '
                                      'connections '
                                      'to '
                                      'database'),
                                     (1638556740.504401,
                                      'connections '
                                      'made'),
                                     (1638556740.5044062,
                                      'create '
                                      'and '
                                      'populate '
                                      'test '
                                      'table'),
                                     (1638556741.5655606,
                                      'test '
                                      'table '
                                      'created '
                                      'and '
                                      'populated'),
                                     (1638556741.5655668,
                                      'connection '
                                      '1 '
                                      ': '
                                      'start '
                                      'transaction '
                                      'and '
                                      'issue '
                                      'select '
                                      'query'),
                                     (1638556741.8472848,
```

```
                                                 'connection '
                                                 '1 '
                                                 ': '
                                                 'select '
                                                 'query '
                                                 'completed '
                                                 '(transaction '
                                                 'ID '
                                                 '9426)'),
                                                (1638556741.847292,
                                                 'connection '
                                                 '2 '
                                                 ': '
                                                 'start '
                                                 'transaction '
                                                 'issue '
                                                 'delete '
                                                 'query'),
                                                (1638556742.2255082,
                                                 'connection '
                                                 '2 '
                                                 ': '
                                                 'delete '
                                                 'query '
                                                 'completed '
                                                 'and '
                                                 'transaction '
                                                 'committed '
                                                 '(transaction '
                                                 'ID '
                                                 '9428)'),
                                                (1638556742.225538,
                                                 'connection '
                                                 '1 '
                                                 ': '
                                                 'issue '
                                                 'identical '
                                                 'delete '
                                                 'query'),
                                                (1638556742.2735696,
                                                 'connection '
                                                 '1 '
                                                 ': '
                                                 'transaction '
                                                 'aborts '
                                                 '(expected '
                                                 'transactions '
                                                 'ID '
                                                 '9426,9428 '
                                                 'actual '
                                                 '9426,9428)'),
                                                (1638556742.3117464,
                                                 'disconnect '
                                                 'connections '
                                                 'from '
                                                 'database'),
                                                (1638556742.3124094,
                                                 'disconnections '
                                                 'complete')],
'Transaction Aborting With UPDATE': [(1638556742.3124175,
                                      'make '
                                      'connections '
                                      'to '
                                      'database'),
                                     (1638556743.0069401,
                                      'connections '
                                      'made'),
                                     (1638556743.0069442,
                                      'create '
                                      'and '
                                      'populate '
                                      'test '
                                      'table'),
                                     (1638556744.0649607,
                                      'test '
                                      'table '
                                      'created '
                                      'and '
                                      'populated'),
                                     (1638556744.0649705,
                                      'connection '
                                      '1 '
                                      ': '
                                      'start '
                                      'transaction '
                                      'and '
                                      'issue '
                                      'select '
                                      'query'),
                                     (1638556744.324732,
                                      'connection '
                                      '1 '
                                      ': '
                                      'select '
                                      'query '
                                      'completed '
                                      '(transaction '
                                      'ID '
                                      '9448)'),
```

```
                                                  (1638556744.3247488,
                                                   'connection '
                                                   '2 '
                                                   ': '
                                                   'start '
                                                   'transaction '
                                                   'issue '
                                                   'update '
                                                   'query'),
                                                  (1638556744.7232127,
                                                   'connection '
                                                   '2 '
                                                   ': '
                                                   'update '
                                                   'query '
                                                   'completed '
                                                   'and '
                                                   'transaction '
                                                   'committed '
                                                   '(transaction '
                                                   'ID '
                                                   '9450)'),
                                                  (1638556744.723228,
                                                   'connection '
                                                   '1 '
                                                   ': '
                                                   'issue '
                                                   'update '
                                                   'query, '
                                                   'same '
                                                   'rows, '
                                                   'different '
                                                   'set '
                                                   'value'),
                                                  (1638556744.7706356,
                                                   'connection '
                                                   '1 '
                                                   ': '
                                                   'transaction '
                                                   'aborts '
                                                   '(expected '
                                                   'transactions '
                                                   'ID '
                                                   '9448,9450 '
                                                   'actual '
                                                   '9448,9450)'),
                                                  (1638556744.8086846,
                                                   'disconnect '
                                                   'connections '
                                                   'from '
                                                   'database'),
                                                  (1638556744.8091354,
                                                   'disconnections '
                                                   'complete')],
'Transaction Blocking With DELETE': [(1638556727.6200874,
                                                   'make '
                                                   'connections '
                                                   'to '
                                                   'database'),
                                                  (1638556728.3250382,
                                                   'connections '
                                                   'made'),
                                                  (1638556728.3250408,
                                                   'create '
                                                   'and '
                                                   'populate '
                                                   'test '
                                                   'table'),
                                                  (1638556729.4078872,
                                                   'test '
                                                   'table '
                                                   'created '
                                                   'and '
                                                   'populated'),
                                                  (1638556729.4078913,
                                                   'connection '
                                                   '1 '
                                                   ': '
                                                   'start '
                                                   'transaction '
                                                   'and '
                                                   'issue '
                                                   'delete '
                                                   'query'),
                                                  (1638556729.5423992,
                                                   'connection '
                                                   '1 '
                                                   ': '
                                                   'delete '
                                                   'query '
                                                   'completed'),
                                                  (1638556729.5424004,
                                                   'connection '
                                                   '2 '
                                                   ': '
                                                   'start '
                                                   'transaction '
                                                   'and '
                                                   'issue '
```

```
                                                 'identical '
                                                 'delete '
                                                 'query'),
                                                (1638556729.619567,
                                                 'begin '
                                                 'sleep '
                                                 'for '
                                                 '10 '
                                                 'seconds '
                                                 'to '
                                                 'show '
                                                 'connection '
                                                 '2 '
                                                 'is '
                                                 'blocked'),
                                                (1638556739.6262894,
                                                 'sleep '
                                                 'completed'),
                                                (1638556739.6262994,
                                                 'connection '
                                                 '1 '
                                                 ': '
                                                 'commit'),
                                                (1638556739.7468402,
                                                 'connection '
                                                 '2 '
                                                 ': '
                                                 'delete '
                                                 'query '
                                                 'completed'),
                                                (1638556739.7968895,
                                                 'disconnect '
                                                 'connections '
                                                 'from '
                                                 'database'),
                                                (1638556739.7974122,
                                                 'disconnections '
                                                 'complete')],
'VACUUM Removes Only Unusable Rows': [(1638556685.9959383,
                                       'make '
                                       'connections '
                                       'to '
                                       'database'),
                                      (1638556686.703059,
                                       'connections '
                                       'made'),
                                      (1638556686.7030616,
                                       'create '
                                       'and '
                                       'populate '
                                       'test '
                                       'tables'),
                                      (1638556688.915437,
                                       'test '
                                       'tables '
                                       'created '
                                       'and '
                                       'populated, '
                                       '100 '
                                       'rows '
                                       'each'),
                                      (1638556688.9154384,
                                       'connection '
                                       '1 '
                                       ': '
                                       'start '
                                       'transaction '
                                       'and '
                                       'issue '
                                       'select '
                                       'query '
                                       'on '
                                       '`table_1`'),
                                      (1638556689.0596015,
                                       'connection '
                                       '1 '
                                       ': '
                                       'select '
                                       'query '
                                       'completed, '
                                       'transaction '
                                       'left '
                                       'open'),
                                      (1638556689.0596042,
                                       'connection '
                                       '2 '
                                       ': '
                                       'issue '
                                       'delete '
                                       'query '
                                       'on '
                                       '`table_2`'),
                                      (1638556689.1782885,
                                       'connection '
                                       '2 '
                                       ': '
                                       'delete '
                                       'query '
                                       'complete'),
```

```
                                                                (1638556689.291906,
                                                                 'connection '
                                                                 '2 '
                                                                 ': '
                                                                 '`table_2` '
                                                                 'has '
                                                                 '100 '
                                                                 'deleted '
                                                                 'rows'),
                                                                (1638556689.2919154,
                                                                 'connection '
                                                                 '2 '
                                                                 ': '
                                                                 'vacuum '
                                                                 'full '
                                                                 '`table_2` '
                                                                 'to '
                                                                 '100 '
                                                                 'percent'),
                                                                (1638556689.6811285,
                                                                 'connection '
                                                                 '2 '
                                                                 ': '
                                                                 'vacuum '
                                                                 'complete'),
                                                                (1638556689.7753546,
                                                                 'connection '
                                                                 '2 '
                                                                 ': '
                                                                 '`table_2` '
                                                                 'has '
                                                                 '100 '
                                                                 'deleted '
                                                                 'rows'),
                                                                (1638556689.77536,
                                                                 'connection '
                                                                 '1 '
                                                                 ': '
                                                                 'commit '
                                                                 'transaction'),
                                                                (1638556689.8143904,
                                                                 'connection '
                                                                 '1 '
                                                                 ': '
                                                                 'commit '
                                                                 'complete'),
                                                                (1638556689.9156435,
                                                                 'connection '
                                                                 '2 '
                                                                 ': '
                                                                 '`table_2` '
                                                                 'has '
                                                                 '100 '
                                                                 'deleted '
                                                                 'rows'),
                                                                (1638556689.9156551,
                                                                 'connection '
                                                                 '2 '
                                                                 ': '
                                                                 'vacuum '
                                                                 'full '
                                                                 '`table_2` '
                                                                 'to '
                                                                 '100 '
                                                                 'percent'),
                                                                (1638556690.2355154,
                                                                 'connection '
                                                                 '2 '
                                                                 ': '
                                                                 'vacuum '
                                                                 'complete'),
                                                                (1638556690.331414,
                                                                 'connection '
                                                                 '2 '
                                                                 ': '
                                                                 '`table_2` '
                                                                 'has '
                                                                 '0 '
                                                                 'deleted '
                                                                 'rows'),
                                                                (1638556690.3314233,
                                                                 'disconnect '
                                                                 'connections '
                                                                 'from '
                                                                 'database'),
                                                                (1638556690.3320272,
                                                                 'disconnections '
                                                                 'complete')]}}}},
'tests': {'dc2.large': {2: {}}},
'versions': {'dc2.large': {2: 'PostgreSQL 8.0.2 on i686-pc-linux-gnu, '
                              'compiled by GCC gcc (GCC) 3.4.2 20041017 (Red '
                              'Hat 3.4.2-6.fc3), Redshift 1.0.33759'}}}
```