Query Memory Allocation Behaviour with Strings

Max Ganz II @ Redshift Research Project

26th September 2021 (updated 1st May 2024)

Abstract

An open question has been whether with `varchar` and `char` Redshift when moving rows to memory, either during data load (`COPY`) or query processing, allocates the full length of the string as specified in the DDL, or only the allocates the actual length of the string. Four query steps were investigated (`aggr`, `hash`, `sort` and `unique`) and for all of them `varchar` does *not* allocate its full length in the DDL, but rather only the length of the actual string, but `char` *does* allocate the full length in the DDL, regardless of the length of the actual string. It seems not entirely unreasonable, although certainly it is not proven, to imagine these findings may be generally true for query processing. No findings were made for behaviour during data load.

Contents

Introduction

An important open question is whether or not Redshift, when bringing `varchar` or `char` into memory from disk needs to allocate memory for the full length of the string as defined in the DDL of the table, or if it needs only allocates the memory needed for the actual string held in a value.

If memory must be allocated for the DDL length of the string, `varchar(max)` becomes almost problematic, because even a few of these in a row massively increase memory use per row, and memory use per row is the critical factor when it comes to determining if hashes run from memory or from disk and if hashes run from disk, cluster performance is hammered, to an extent that it must more or less be avoided at all costs, and, critically, hash joins (of course) need to perform a hash.

To put it in more concrete terms, every query has a fixed amount of memory available to it, depending on your WLM queue and slot configuration, and although merge joins are completely insensitive to memory, hash joins *must* have the hash fit into memory or cluster performance is hammered.

If rows take up a megabyte of memory each, and a query has (as it typical) a few hundred megabytes of memory available (that is memory per slice, which is the value which matters), you can have *very* few rows in the table before the hash joins run from disk.

(Of course, Redshift will let you do all of this. There are no warnings. If you want to hash join say two ten terabyte tables, Redshift will go right and try to do it. It's just the cluster will have ground to a halt and you won't have a clue why.)

The general view in this matter, of whether memory allocation is to the length of DDL, has been that it is, given comments like this;

> Strongly recommend that you not use VARCHAR(MAX) as the default column size. This requires more memory to be allocated during all phases of the query and will reduce them amount of work that can be done in memory without spilling to disk.

Which comes from Joe Harris, senior bigwig developer since the dawn of Redshift (well, metaphorically speaking - he was a user, and then joined AWS, but he's been on the dev team now for many years and is as the dev team members go the most publicly visible and active).

However, to my knowledge, there has never been any actual evidence.

About a week ago (after an earlier abortive attempt, months ago) I finally figured out a viable test method, and this white paper documents the test method and the results.

Test Method

In the system tables there are a set of tables each of which carries information about each type of step, e.g. aggregation, hash, unique, etc.

Of these, six contain a column `is_diskbased`, which indicates if the step, lacking sufficient memory, was unable to run from memory and had to run from disk.

The basis of the test method is to find the number of rows for a given data type where a test query is just able to run in memory, and just unable to run in memory, and and then change the data type; if this affects whether the query runs in memory or not, we can conclude the data type is affecting how much memory is being allocated.

So, for example, we would begin by creating a table with `varchar(1)` and a single character length string, and finding out the number of rows to just run in memory and the number to just run on disk, and then change the data type to `varchar(65535)`, while keeping the string length at one character, and see if this affects whether the query runs in memory or on disk.

If there is no change in behaviour, we can reasonably conclude the DDL length makes no difference to memory allocation.

Of the six steps which indicate whether they ran from disk, results were found for `aggr`, `hash`, `sort` and `unique`. Of the remaining two, I do not offhand know how to make a query which forces a `save` step, and the `window` step seemed always to run from memory (within the limits of the number of rows I could reasonably test with).

(Of course, the behaviour of four particular steps is not by any stretch of the imagination *all* query behaviour, but these four steps can be effectively investigated, and what we learn of them will likely be of considerable value in our best effort judgement of query behaviour as a whole.)

Note that for the `sort` test, the smallest data types are `varchar(1)` and `char(1)`, but for the other steps, the smallest data types had to be `varchar(5)` and `char(5)`, because the way those other steps work mean if a single character string was used, memory allocation would always be tiny, and so always run in memory.

For example, consider `unique`. A single character string ranges from ASCII 32 to ASCII 126 - about 95 characters. No matter how many rows are in the table, the `unique` step will only need to allocate 95 entries in memory, to keep track of the number of rows for each possible character.

A five character string however has about 95^5 possible strings, and now the memory allocated is large enough that it is possible to end up running on disk - it depends on the number of rows, since the strings are randomly generated; the number of rows is roughly the number of entries the step has to allocate to keep track of the count of each string.

Results

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

Test duration was 1,933 seconds (including server bring up and shut down).

dc2.large, 2 nodes (1.0.30840)

aggr

data type	rows	mem/disk
varchar(5)	5.875m	memory
varchar(5)	6.000m	disk
varchar(65535)	5.875m	memory
varchar(65535)	6.000m	disk
char(5)	7.500m	memory
char(5)	7.750m	disk

hash

data type	rows	mem/disk
varchar(5)	1.750m	memory
varchar(5)	2.000m	disk
varchar(65535)	1.750m	memory
varchar(65535)	2.000m	disk
char(5)	1.750m	memory
char(5)	2.000m	memory

sort

data type	rows	mem/disk
varchar(1)	8.000m	memory
varchar(1)	8.125m	disk
varchar(65535)	8.000m	memory
varchar(65535)	8.125m	disk

data type	rows	mem/disk
char(1)	13.500m	memory
char(1)	13.750m	disk
char(2)	13.500m	memory
char(2)	13.750m	disk
char(3)	13.500m	memory
char(3)	13.750m	disk
char(4)	13.500m	memory
char(4)	13.750m	disk
char(5)	13.500m	disk
char(5)	13.750m	disk
char(10)	8.250m	memory
char(10)	8.500m	disk

unique

data type	rows	mem/disk
varchar(5)	6.500m	memory
varchar(5)	7.000m	disk
varchar(65535)	6.500m	memory
varchar(65535)	7.000m	disk
char(5)	10.000m	memory
char(5)	10.125m	disk
char(8)	10.000m	memory
char(8)	10.125m	disk
char(9)	7.500m	memory
char(9)	8.000m	disk

Discussion

Looking at `aggr`, the boundary between memory and disk for `varchar(5)` is between 5.875m and 6m rows (remembering we're using `varchar(5)` as we have to use five character strings with `aggr`, or the step always runs in memory). We see that if we move to `varchar(65535)`, this does not change.

We see exactly the same behaviour (although with different numbers of rows marking the boundary between memory and disk) for `hash`, `sort` and 'unique.

This indicates that for `varchar` the DDL length is *not* being allocated, but, rather, only the length of the actual string.

Turning to `char`, and looking at `hash`, we see that with `char(5)`, the boundary between memory and disk is between 1.75m and 2m rows, but with `char(17)` the boundary is between 1.5m and 1.75m rows - even though the actual strings are always five characters in length.

This indicates that for `char` the DDL length *is* being allocated, rather than only the length of the actual string.

Additionally, some interesting behaviour with `char` was observed.

Looking at `sort`, we see that `varchar(1)` has the boundary between memory and disk between 8m and 8.125m rows, where `char(1)` has that boundary between 13.5m and 13.75m rows. The obvious conclusion is that, at least in this specific case where the DDL lengths are the same and the string is the full length of the DDL, `char` is using less memory than `varchar`.

What is almost certainly the cause of this is that `varchar` has a four byte overhead, which indicates the length of the actual string in the `varchar`. A `char` value does not have this overhead, but, of course, instead always has its full length in the DDL allocated and processed, which is usually a much larger overhead.

We can see this when we look at `char(10)`, and compare the row counts with `char(1)`. Although the actual string is in all cases always one character, `char(10)` has the boundary between memory and disk between 8.25m and 8.5m rows, where as `char(1)` has it between 13.5m and 13.75m rows.

Finally, something curious can be seen in the `char` results for `sort`. The boundary between memory and disk is between 13.5m and 13.75m rows for `char(1)` through to `char(4)`, inclusive both ends. It is only with `char(5)` we see the boundary move.

The same behaviour can be seen with `unique`, where the boundary is unchanged for `char(5)` through to `char(8)`, inclusive both ends (and then with `unique` we see a large change, which is interesting).

I'm not sure what to make of this. A deeper investigation of how the boundary moves is needed - perhaps the boundary always moves every four characters, rather than (as we might expect) every character.

Note there are two broad areas where memory allocation for strings has been considered an open question. The first is in queries, which has been investigated in this paper, but the second is in data loading, via `COPY`. The question here is whether during data load each row must have the string columns allocated in memory to their full DDL length. As yet I have not thought of a method to investigate this situation, as there is no obvious memory/disk type indicator for `COPY`.

The findings contradict the statement (reproduced in the Introduction) from Joe Harris with regard to `varchar`. That statement though was made over a year ago. Presumably given Joe's comment what was said must have been true; if it is not true now, then a really serious improvement has been made in query processing, but without this improvement being announced to developers.

In my view, AWS obfuscate everything which is not a strength and in line with this, full allocation of `varchar` in query processing would never have been - and never was - documented. This of course rather tends also to mean that once this behaviour has been improved upon, it's a bit difficult to announce, because then you also have to explain to developers why you never mentioned it in the past.

In both cases, before and after, *developers needed to know*, so they could knowingly correctly design Redshift systems.

Conclusions

For the steps `aggr`, `hash`, `sort` and `unique`, `varchar` does *not* allocate memory equal to the DDL length, but only the length of the actual string.

For the steps `aggr`, `hash`, `sort` and `unique`, `char` *does* allocate memory equal to the DDL length, rather than only the length of the actual string.

It does not seem unreasonable, although it is certainly unproven, to consider it possible that query processing in general behaves in the this way, as it would be odd (although of course not impossible) for these four steps to do so while other aspects of query processing do not.

These findings on the face of it appear to directly contradict the statement (reproduced in the Introduction) from Joe Harris regarding `varchar` behaviour, but that statement was made over a year ago; maybe things have changed.

Note no findings have been made with regard to the allocation of memory during data load (`COPY`), where behaviour may well differ from the behaviour found with queries.

Revision History

v1

- Initial release.

v2

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

v3

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

Appendix A : Raw Data Dump

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

```
{'proofs': {'dc2.large': {2: {'aggr': {'char(5)': {7.5: 'memory', 7.75: 'disk'},
                                       'varchar(5)': {5.875: 'memory',
                                                      6: 'disk'},
                                       'varchar(65535)': {5.875: 'memory',
                                                          6: 'disk'}},
                              'hash': {'char(5)': {1.75: 'memory', 2: 'memory'},
                                       'varchar(5)': {1.75: 'memory',
                                                      2: 'disk'},
                                       'varchar(65535)': {1.75: 'memory',
                                                          2: 'disk'}},
                              'sort': {'char(1)': {13.5: 'memory',
                                                   13.75: 'disk'},
                                       'char(10)': {8.25: 'memory',
                                                    8.5: 'disk'},
                                       'char(2)': {13.5: 'memory',
                                                   13.75: 'disk'},
                                       'char(3)': {13.5: 'memory',
                                                   13.75: 'disk'},
                                       'char(4)': {13.5: 'memory',
                                                   13.75: 'disk'},
                                       'char(5)': {13.5: 'disk', 13.75: 'disk'},
                                       'varchar(1)': {8: 'memory',
                                                      8.125: 'disk'},
                                       'varchar(65535)': {8: 'memory',
                                                          8.125: 'disk'}},
                              'unique': {'char(5)': {10: 'memory',
                                                     10.125: 'disk'},
                                         'char(8)': {10: 'memory',
                                                     10.125: 'disk'},
                                         'char(9)': {7.5: 'memory', 8: 'disk'},
                                         'varchar(5)': {6.5: 'memory',
                                                        7: 'disk'},
                                         'varchar(65535)': {6.5: 'memory',
```

```
                                                                  7: 'disk'}}}}},
 'tests': {'dc2.large': {2: {}}},
 'versions': {'dc2.large': {2: 'PostgreSQL 8.0.2 on i686-pc-linux-gnu, '
                               'compiled by GCC gcc (GCC) 3.4.2 20041017 (Red '
                               'Hat 3.4.2-6.fc3), Redshift 1.0.30840'}}}
```

About the Author

I am a C programmer - kernel development, high performance computing, networking, data structures and so on.

I read the C. J. Date book, the classic text on relational database theory, and having learned the principles, wrote a relational database from scratch in C, which purely by chance set me up quite nicely for what came next, moving into data engineering in late 2011, when I joined as the back-end engineer two friends in their startup.

In that startup, I began using Redshift the day it came out, in 2012 (we had been trying to get into the beta programme).

We were early, heavy users for a year and a half, and I ending up having monthly one-to-one meetings with one of the Redshift team managers, where one or two features which are in Redshift today originate from suggestions made in those meetings, such as the distribution style `ALL`.

Once that was done, after a couple of years of non-Redshift data engineering work, I returned to Redshift work, and then in about mid-2018 contracted with a publisher to write a book about Redshift.

The book was largely written but it became apparent I wanted to do a lot of things which couldn't be done with a book - republish on every new Redshift release, for example - and so in the end I stepped back from the contract and developed the web-site, where I publish investigation into, and ongoing monitoring of, Redshift.

So for many years now I've been investigating Redshift sub-systems full-time, one by one, and this site and these investigations are as far as I know the and the only source of this kind of information about Redshift.

Redshift Cluster Cost Reduction Service

I provide consultancy services for Redshift - advice, design, training, getting failing systems back on their feet pronto, the usual gamut - but in particular offer a Redshift cluster cost reduction service, where the fee is and only is one month of the savings made.

Broadly speaking, to give guidance, savings are expected fall into one of two categories; either something like 20%, or something like 80%. The former is

for systems where the business use case is such that Redshift cannot be operated correctly, and this outcome requires no fundamental re-engineering work, the latter is for systems where Redshift can be operated correctly, and usually requires fundamental re-engineering work (which you may or may not wish to engage in, despite the cost savings, in which case we're back to the 20%).

Details and contact information are on the web-site.