Zone Map Sorting By Data Type

Max Ganz II @ Redshift Observatory

9th August 2021

Abstract

Redshift provides functionality to sort rows in a table according to the values in the table columns. Sorting is however not according to the entire values in the columns, but according only to an eight byte signed integer, known as the *sorting value*, which is derived from each value, where the method used to derive the sorting value varies by data type. This document describes in detail the methods used to derive sorting values. About half of the data types sort as would be intuitively expected, but half do not, and system designs which assume all data types to sort as intuitively expected are *not* functioning as their designers expect.

Contents

Introduction

In conventional relational databases, which is to say unsorted relational databases, such as Postgres, rows in tables have no ordering - they are not sorted. Rows typically are stored in the order in which they were added.

Redshift is a *sorted* relational database, which is to say there is support for specifying the order in which rows are held in tables, by dint of each table in its DDL specifying a series of one or more of its columns, known as the *sort key*, where the rows in the table are sorted according to the values in those columns.

So, for example, a table with three columns, `surname`, `name` and 'age', which specifies a sort key of `surname` and then `name`, will sort all rows first by the `surname` (in ascending order), and then for the set of rows for every unique surname, those rows will be further sorted in the order of the value in the `name` column (again, in ascending order). The `age` column will be left in whatever order happened to come from the actions to sort the preceding columns.

This functionality is the entire point of Redshift and is by far the most important of the mechanisms by which Redshift is able to provide timely SQL on Big Data.

The Redshift documentation to my eye has very little to say about sorting - which is to say, there's a lot about how awesome it is, and almost nothing about how it works and how to use it correctly, and it must be used correctly, because if it is not, then Redshift effectively degenerates to being an unsorted relational database, and loses the capability to provide timely SQL on Big Data. You'd be better off with an unsorted column-store database, since they're much less knowledge-intensive to use correctly (note though this is a type of relational database AWS do not offer as a product, so the best you can do if you are sticking with AWS is Postgres).

In the documentation, the most important statement to be found is that sorting is not based on the entire value in a column, but "uses the first eight bytes of the sort key to determine sort order" (sort key here is referring to the value in the column). I remember similar statements being in a couple of places in the documentation, to the effect that the first eight bytes of the value *on disk* are used to determine the sorting order, but now all I can find is here (which is quoted above). I may be wrong, but I think this statement has been largely removed and we see now only that which was overlooked.

The only other useful statement the documentation is that sorting is ascending; so rows go from their smallest value to their largest.

So, in fact, what is happening with sorting is that Redshift *derives* a signed eight-byte value, which I will call the *sorting value*, from each value in each column, where the method used to derive the value varies by data type. The value stored on disk is *not* used, although with some simple data types (the integers, for example) by co-incidence the value on disk does indeed happens to be the same as the sorting value - really this just means the method to derive sorting values for those data types happens to be very simple.

Knowing how sorting values are derived for each data type is *necessary* to knowingly correctly design systems using Redshift. The lack of this information in the documentation is one of a number of omissions which fundamentally prevent developers from knowingly correctly designing systems using Redshift, which leads of course to failures, and this is, in my view, ultimately responsible for the success of Snowflake in capturing business from Redshift.

This document does not attempt to explain the purpose, function and implementation of sorting, as it is a large topic. This will be for a future document.

Test Method

A table is created which contains two columns.

The table has key distribution, on the first column, and the value inserted into the first column is always `0`. This allows us to ensure all rows go to the same single slice, which simplifies testing.

The second column is of the data type we are investigating; the test method as a whole iterates over every data type, dropping and recreating the test table every time.

We then insert one row into the table, with a test value going into the second column.

We then examine `STV_BLOCKLIST`, to obtain the minimum and maximum sorting values for the block which now exists for the second column, which contain the single row in the table, which we just inserted.

Since there is only one row, this allows us to see the sorting value for the value we inserted.

The table is then truncated, and further test values are inserted, one by one, retrieving the sorting value every time.

This builds up a body of data which allows us to figure out how the sorting value is being derived from the value.

Results

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

The `Line` column is used to make it easy, in the Discussion, to reference a given Value/Sorting Value pair.

dc2.large, 2 nodes (1.0.28965)

char(64)

Line	Value	Min Sorting Value (hex)	Max Sorting Value (hex)
1	‘MARTINEZ’	5a454e495452414d	5a454e495452414d
2	‘MURPHY’	59485052554d	202059485052554d
3	‘abcdefg’	67666564636261	2067666564636261
4	‘\ttuvwxyz’	7a79787776757409	7a79787776757409
5	‘tuvwxyz’	7a797877767574	207a797877767574
6	‘1234567 1234567’	37363534333231	2037363534333231
7	‘1234567a1234567’	6137363534333231	6137363534333231
8	NULL	7fffffffffffffff	8000000000000000

varchar(64)

(Note due to technical issues in the current version of the document processing software, the non-Latin scripts have a leading space.)

Line	Value	Min Sorting Value (hex)	Max Sorting Value (hex)
1	‘MARTINEZ’	5a454e495452414d	5a454e495452414d
2	‘MURPHY’	59485052554d	202059485052554d
3	‘abcdefg’	67666564636261	2067666564636261
4	‘\ttuvwxyz’	7a79787776757409	7a79787776757409
5	‘tuvwxyz’	7a797877767574	207a797877767574
6	‘1234567 1234567’	37363534333231	2037363534333231
7	‘1234567a1234567’	6137363534333231	6137363534333231
8	‘ アィイ’	82e3a382e3a282e3	82e3a382e3a282e3

Line	Value	Min Sorting Value (hex)	Max Sorting Value (hex)
9	‘アイウ’	82e3a382e3a282e3	82e3a382e3a282e3
10	‘الغيوم الداكنة تجلب المطر’	8ad9bad884d9a7d8	8ad9bad884d9a7d8
11	‘الغيوم الداكنة تجلب’	8ad9bad884d9a7d8	8ad9bad884d9a7d8
12	‘رطملا بلجت ةنكادلا مويغلا’	84d985d9b7d8b1d8	84d985d9b7d8b1d8
13	‘بلجت ةنكادلا مويغلا’	aad8acd884d9a8d8	aad8acd884d9a8d8
14	‘dark clouds bring’	6f6c63206b726164	6f6c63206b726164
15	‘dark clouds bring rain’	6f6c63206b726164	6f6c63206b726164
16	‘gnirb sduolc krad’	6473206272696e67	6473206272696e67
17	‘niar gnirb sduolc krad’	696e67207261696e	696e67207261696e
18	NULL	7fffffffffffffff	8000000000000000

date

Line	Value	Sorting Value
1	‘4713-01-01 BC’	-2451507
2	‘0001-12-31 BC’	-730120
3	‘0001-01-01 AD’	-730119
4	‘1999-12-31 AD’	-1
5	‘2000-01-01 AD’	0
6	‘2000-01-02 AD’	1
7	‘5874897-12-31 AD’	2145031948
8	NULL	9223372036854775807

float4

Line	Value	Sorting Value
1	‘NaN’	-9223372036854775808
2	-3.402823e+38	-9223372036854775808
3	-9223371761976868352.0	-9223372036854775808
4	-9223371761976868351.0	-9223371487098961920
5	-1234.56789	-1234
6	-50.75	-50
7	-50.0	-50
8	-1	-1
9	-0.5	0

Line	Value	Sorting Value
10	0	0
11	0.5	0
12	1	1
13	50.0	50
14	50.75	50
15	1234.56789	1234
16	9223371761976868351.0	9223371487098961920
17	9223371761976868352.0	9223372036854775807
18	3.402823e+38	9223372036854775807
19	NULL	9223372036854775807

float8

Line	Value	Sorting Value
1	‘NaN’	-9223372036854775808
2	-1.7e+308	-9223372036854775808
3	-9223372036854775296.0	-9223372036854775808
4	-9223372036854775295.0	-9223372036854774784
5	-1234.56789	-1234
6	-50.75	-50
7	-50.0	-50
8	-1	-1
9	-0.5	0
10	0	0
11	0.5	0
12	1	1
13	50.0	50
14	50.75	50
15	1234.56789	1234
16	9223372036854775295.0	9223372036854774784
17	9223372036854775296.0	9223372036854775807
18	1.7e+308	9223372036854775807
19	NULL	9223372036854775807

int2

Line	Value	Sorting Value
1	-32768	-32768
2	-100	-100
3	-1	-1
4	0	0
5	1	1
6	100	100
7	32767	32767
8	NULL	9223372036854775807

Line	Value	Sorting Value

int4

Line	Value	Sorting Value
1	-2147483648	-2147483648
2	-100	-100
3	-1	-1
4	0	0
5	1	1
6	100	100
7	2147483647	2147483647
8	NULL	9223372036854775807

int8

Line	Value	Sorting Value
1	-9223372036854775808	-9223372036854775808
2	-100	-100
3	-1	-1
4	0	0
5	1	1
6	100	100
7	9223372036854775807	9223372036854775807
8	NULL	9223372036854775807

numeric(18,0)

Line	Value	Sorting Value
1	-999999999999999999	-999999999999999999
2	-15	-15
3	-1	-1
4	0	0
5	‘NaN’	0
6	1	1
7	15	15
8	999999999999999999	999999999999999999
9	NULL	9223372036854775807

numeric(18,4)

Line	Value	Sorting Value
1	-99999999999999.9999	-999999999999999999
2	-15.5000	-155000
3	-15.0000	-150000
4	-15	-150000
5	-1.0000	-10000
6	0.0000	0
7	1.0000	10000
8	15	150000
9	15.0000	150000
10	15.5000	155000
11	99999999999999.9999	999999999999999999
12	NULL	9223372036854775807

numeric(38,0)

Line	Value	Sorting Value
1	-99999999999999999999999999999999999999	-5421010862427522171
2	-36893488147419103233	-3
3	-36893488147419103232	-2
4	-18446744073709551617	-2
5	-18446744073709551616	-1
6	-9223372036854775809	-1
7	-9223372036854775808	-9223372036854775808
8	-9223372036854775807	-9223372036854775807
9	-15	-15
10	-1	-1
11	0	0
12	‘NaN’	0
13	1	1
14	15	15
15	9223372036854775806	9223372036854775806
16	9223372036854775807	9223372036854775807
17	NULL	9223372036854775807
18	9223372036854775808	0
19	18446744073709551615	0
20	18446744073709551616	1
21	36893488147419103231	1
22	36893488147419103232	2
23	99999999999999999999999999999999999999	5421010862427522170

time

Line	Value	Sorting Value
1	‘00:00:00.000000’	0

Line	Value	Sorting Value
2	‘00:00:00.000001’	1
3	‘00:00:01.000000’	1000000
4	‘00:01:00.000000’	60000000
5	‘01:00:00.000000’	3600000000
6	‘23:59:59.999999’	86399999999
7	NULL	9223372036854775807

timetz

Line	Value	Sorting Value
1	‘00:00:00.000000+0’	0
2	‘00:00:00.000001+0’	1
3	‘12:00:00.000000+0’	43200000000
4	‘23:59:59.999999+0’	86399999999
5	‘22:00:00.000000-4’	7200000000
6	‘02:00:00.000000+0’	7200000000
7	NULL	9223372036854775807
8	‘00:00:00.000000-1459’	53940000000
9	‘00:00:00.000000+1459’	32460000000

timestamp

Line	Value	Sorting Value
1	‘4713-01-01 00:00:00.000000 BC’	-211810204800000000
2	‘0001-12-31 23:59:59.999999 BC’	-63082281600000001
3	‘0001-01-01 00:00:00.000000 AD’	-63082281600000000
4	‘1999-12-31 23:59:59.999999 AD’	-1
5	‘2000-01-01 00:00:00.000000 AD’	0
6	‘2000-01-01 00:00:00.000001 AD’	1
7	‘2000-01-01 00:00:01.000000 AD’	1000000
8	‘2000-01-01 00:01:00.000000 AD’	60000000
9	‘2000-01-01 01:00:00.000000 AD’	3600000000
10	‘2000-01-02 00:00:00.000000 AD’	86400000000
11	‘2000-02-01 00:00:00.000000 AD’	2678400000000
12	‘2001-01-01 00:00:00.000000 AD’	31622400000000
13	‘294276-12-31 23:59:59.999999 AD’	9223371331199999999
14	NULL	9223372036854775807

timestamptz

Line	Value	Sorting Value
1	‘4713-01-01 00:00:00.000000-1459 BC’	-211810150860000000
2	‘1999-12-31 23:00:00.000000+0 AD’	-3600000000

Line	Value	Sorting Value
3	‘2000-01-01 00:00:00.000000+0 AD’	0
4	‘2000-01-01 00:00:00.000000+1 AD’	-3600000000
5	‘294276-12-31 23:59:59.999999+1459 AD’	9223371277259999999
6	NULL	9223372036854775807

Discussion

char and varchar

There's a great deal of complexity involved in the derivation of sorting values from `char` and `varchar`.

On the very surface, it's simple : the first eight bytes of the string form the bit pattern which is used as the sorting value.

However, when we examine the sorting values for strings, the first thing we find they're *reversed*, and this needs to be explained.

Once that's out of the way, the next thing we find is there's a bug in how sorting values are formed. Redshift keeps track for each disk block the minimum and maximum sorting values of all the rows in the block. With this bug, what you find it with strings you can end up having different minimum and maximum sorting values *for the same string.* You can have a block with *one* row, which has a *different* minimum and maximum sorting value; likewise, you can have a block with two or more rows, which are different values, but which has the *same* minimum and maximum sorting values. A couple of years ago I tried for six months to get Support to understand, before eventually giving up.

Then we need to move on to how sorting values are formed for multi-byte UTF-8 strings.

Having set the scene by beginning to look into Unicode, we then need to examine what happens with right-to-left scripts.

Finally, to help us all recover from the deluge, I finish off with a simple little note about `char` and a word or two about `NULL`, at which point we can all go and have tea and biscuits.

Sorting Value Reversal

When we compare strings, what we do conceptually speaking is compare byte-by-byte from the most significant letter to the least significant letter. This leads to the question of what the most and least significant letters are.

We can demonstrate this with a little list of two surnames;

```
MARTINEZ
MURPHY
```

If we sort this little list, we compare the names from left-to-right; the "M" first (the strings are equal), then the "A" with the "U" and now we see MARTINEZ is smaller and we can in fact stop comparing at this point. (One thing we see here, and I'll come back to this since it's important, is that although MURPHY is *shorter* than MARTINEZ, it is still *larger*.) In our comparison here what we really have done is a character-by-character compare, from most significant letter to least significant letter, and we see the most-significant letter for left-to-right scripts is the left-most letter (and for right-to-left scripts, which we come to later, it is of course the right-most letter).

We now need to think about how a string is stored in memory.

To make this explanation simple we have a string which is all single-byte UTF-8/ASCII, and we see a string is an array of one byte values, and being a left-to-right script, the most significant *letter* is on the left and has the lowest memory address, the least significant letter is on the right and has highest memory address, which is also how we write a string on paper or in a document like this. So in memory, we write MARTINEZ like this;

```
MARTINEZ
```

Which is to say, from lowest ('M') to highest ('Z') memory addresses.

Now, when we store a number - say an eight byte integer like the sorting value - where we're little endian, the least significant bytes are in the lowest memory addresses, the most significant bytes are in the higher memory addresses. A problem now comes that this is *not* how we actually write numbers on paper or in a document like this.

So let's say we have the number ten million; we ourselves write it like so;

```
10,000,000.
```

In memory though, ten million is stored like this;

```
000,000,01
```

Which is to say, from lowest (the first '0') to highest (the '1') memory addresses.

Razor-witted, eagle-eyed readers who've recently imbibed extra-strength coffee may begin already to see where this is going.

When Redshift is scanning sortkey values in the Zone Map, it is performing compares of signed eight-byte integers - "is this number larger than, less than or equal to this other number?"

Numbers of course are arranged as we've just seen like so - "000,000,01" - and so when numbers are being compared, the code which does this work knows this and that their most-significant byte is in the highest memory address and it does the right thing.

The problem is though that this is *not* how strings are arranged in memory. Strings store their most-significant byte (the most-significant letter) in the lowest memory address. Strings are the wrong way around! so if we took the first eight bytes of a string, made a sorting value of it (a signed eight byte integer), and tried to compare it, we'd in effect be comparing the string from its least-significant letter toward its most-significant letter, and that's wrong.

(If we did a *string* compare, we'd be fine, because the string compare code knows the most-significant letter is in the smallest memory address and would expect it and depend upon it and do the right thing - but we're not; we're using the code which does signed integer compares).

So what happens, of course, is that when Redshift is producing sorting values for `char` or `varchar`, the first eight bytes from the string are *reversed* to produce the sorting value.

This allows Redshift to compare sorting values for strings using a normal signed eight-byte integer comparison, as by reversing the string this comparison is in effect performing a string compare.

We've now covered enough for me to be able to describe to you the explanation for an otherwise confusing observation made when examining `minvalue` and `maxvalue` in `STV_BLOCKLIST` that for blocks in `char` and `varchar` columns we can find plenty of blocks where `minvalue` is larger than the `maxvalue`.

Let's begin by looking at our two-name example data;

```
MARTINEZ
MURPHY
```

In alphanumeric sorting, MARTINEZ is smaller than MURPHY, but note that MURPHY is shorter and where it is less than eight bytes is being padded with spaces (ASCII decimal 32, hex 0x20).

Since MARTINEZ is smallest, the block will use its sorting value as the `minvalue` for the block, and the `maxvalue` will be from MURPHY.

Here are the `minvalue` and `maxvalue` as they are stored, in their reversed form, in `STV_BLOCKLIST`;

```
0b 01011010 01000101 01001110 01001001 01010100 01010010 01000001 01011010
0x       5a       45       4e       49       54       52       41       4d
          Z        E        N        I        T        R        A        M

0b 00100000 00100000 01011001 01001000 01010000 01010010 01010101 01001101
0x       20       20       59       48       50       52       55       4d
                            Y        H        P        R        U        M
```

The `minvalue` then is 0x5A454E495452414D and the `maxvalue` is the much smaller value of 0x202059485052554D :-)

What's happened is that the reversing of the byte order has put the padding, which has a low value, into the most significant bytes of the `maxvalue`, and so it becomes smaller than `minvalue`.

In fact, as we will see, padding with spaces looked to be a bug. Padding should be with binary 0. However, even if it was, we would still see a reversed `minvalue` and `maxvalue`, because MURPHY is shorter than eight bytes, and so is padded, and the reversal of the sorting value to allow integer compare to work makes the sorting value for MURPHY a small number, by putting zeros into the most significant bits.

Sorting Value Bug

For every data type (except as we will see, `char` and `varchar`), a single value has a single sorting value. This is inherent and seems obviously to be so. Redshift keeps track of the minimum and maximum sorting values for each block, and if a block has only one value, the minimum and maximum must be identical. It would hardly make sense for *one* value to have *different* sorting values for minimum and maximum - except that *is* what happens with `char` and `varchar`.

To compute the minimum sorting value for a string, Redshift takes the first eight bytes of the string and then as we've discussed reverses them but, crucially, if the string is *shorter* than eight bytes, then the sorting value being eight bytes long has to be padded to its full length.

What we find is that for the minimum sorting value, Redshift pads with binary 0, but for the maximum sorting value, padding is with *space characters* (ASCII decimal 32, hex 0x20).

So "MURPHY" becomes for the minimum sorting value (the '0' here means binary zero, not ASCII zero) "00YHPRUM", but for the maximum sorting value becomes "SSYHPRUM" (the 'S' here means ASCII space).

We see the minimum and maximum sorting values for "MURPHY" are;

```
0b 00000000 00000000 01011001 01001000 01010000 01010010 01010101 01001101
0x        0        0       59       48       50       52       55       4d
                            Y        H        P        R        U        M

0b 00100000 00100000 01011001 01001000 01010000 01010010 01010101 01001101
0x       20       20       59       48       50       52       55       4d
                            Y        H        P        R        U        M
```

As such, a block with a single row, where that row is a string which is less than eight bytes, has *different minimum and maximum sorting values*, even though there *is only one record*.

Remember now the purpose of the Zone Map, and the method by which it functions, is to keep track of the minimum and maximum sorting values in each block in each column in each table, such that when we search for a value, we compute its sorting value and compare it to the minimum and maximum sorting values for every block in the table; if the sorting value is smaller than the minimum sorting value for a block, or greater than the maximum, then Redshift knows it does not need to read that block.

What's happening here is that the maximum sorting value is *larger* than it ought to be, and as such, there are going to be times when Redshift thinks it must read a block when in fact it did not need to do so.

Now, on the face of it, the impact of this bug is vast, because the maximum sorting value becomes very large indeed, because the binary 32 used for padding is setting bits up at the most-significant end of the sorting value. The maximum sorting value for MURPHY ought to be 0x59485052554d, but is in fact 0x202059485052554d, which is a *much* larger value.

As an example, imagine we have a column which is `char(1)` and it contains the letter `f` or `m`, to indicate gender, and that the column is fully sorted, so we have blocks which are either all `f` or all `m`. (Ignore the boundary block which has some `f` and some `m`.)

We can search for either `f` (0x66) or `m` (0x6D).

Let's imagine we search for `m`.

What should happen is Redshift computes the sorting value for `m`, and then compares it against every block, and if it is equal to or greater than the minimum sorting value for a block, *and* equal to or smaller than the maximum sorting value for that block, then that block will be read; otherwise it will not be read.

The intention here then is that we end up ignoring all the blocks which contain `f` and reading all the blocks which contain `m`.

What we expect to find though because of this bug is that every block has the correct minimum sorting value, but a huge maximum sorting value; and so searching for the sorting value 0x6D we find it is always greater than or equal to the minimum sorting value for blocks (which is 0x66 for `f` blocks and 0x6D for `m` blocks) and then, critically, is always less than the maximum sorting value, both for blocks of `f` and blocks of `m`, because the maximum sorting value is so large (being hex 0x2020202020202066 and 0x202020202020206D, respectively).

In other words, we end up reading every block, instead of just those containing `m`.

However, this is *not* what happens - because the bug in how sorting values are computed also exists in *another* part of the code, and this acts to almost but not quite fully mask the problem.

It turns out that when computing the sorting value for the value being searched for, Redshift is *also* here producing two different sorting values; as before, the first padded with binary zero and the second padded with spaces.

What happens then is that Redshift uses the first version when comparing against the minimum sorting value for a block, and the second version when comparing against the maximum sorting value for a block.

So when we search for `m`, the sorting value used to compare with the minimum sorting value for a block is 0x6D, but the sorting value used to compare with the maximum sorting value for a block is 0x202020202020206D.

This second blunder almost, but not completely, masks the effect of the first blunder.

If we imagine now searching for `m`, when we examine a block containing only `f`, we will as before be greater than the minimum sorting value of the block (and so, so far, the block is eligible to be read) but then the maximum sorting value of `m` is 0x202020202020206D, but the maximum sorting value of the block is 0x2020202020202066, so we do *not* read the block.

Phew! two wrongs *do* make a right - well, almost. We are in fact not *quite* off the hook, because in fact you can still go wrong, and read blocks you do not have to read.

Imagine you have an table with a single `char(8)` column, and it's filled with random *seven* character strings, from `abcdefg` to `tuvwxyz`. (This is a contrived example so I can most easily demonstrate the problem; you can generalize after you understand it.)

Since these are seven byte strings, there is one byte of padding, and so each block will have its minimum sorting value, which is correct, and its maximum sorting value, which is much larger than it should be, because the padding byte is the most significant byte and it is set to 0x20.

We can assume every block will contain at least one `abcdefg` (the smallest value) and one `tuvwxyz` (the largest value).

As such, for each block, the minimum sorting value will be 0x0067666564636261 (remembering that the string is reversed and then padded), and the maximum sorting value will be 0x207A797877767574.

What now happens if we search for the string `\textbackslash{}ttuvwxyz`?

The minimum *and* the maximum sorting value for `\textbackslash{}ttuvwxyz` are both 0x097A797877767574, because we have an eight byte string so no padding is occurring.

We see that this minimum sorting value is larger than the minimum sorting value of the blocks, *and* that the maximum sorting value is *less* than the maximum sorting values of the blocks - and so we *do* in fact end up reading every block in the column, even though we should read none at all.

The basic problem is that by padding with 0x20, any ASCII characters below that can sneak in under the radar - there aren't many though which are commonly used, I think only space, tab and newline.

However, there is an additional aspect to this bug, which complicates matters, and makes my head hurt (which is to say, makes it hard to reason about).

You see, it turns out the strings which are affected are not only those which are less than eight bytes in length.

If a string has one or contiguous spaces which cross the eighth byte position, the bug also happens then; and in fact I think it really the proper definition of when this bug manifests.

So we find the string `1234567a1234567` has the same minimum and maximum sorting value, of 0x6137363534333231.

The string `1234567 1234567` however (note the central 'a' is now a space)has a minimum sorting value of 0x37363534333231 but a maximum sorting value of 0x2037363534333231.

(Although an explanation of the proof is beyond the scope of this document, note that for values which produce two sorting values, it is the *maximum* sorting value which is used when ordering rows in the column, and so influencing the effectiveness of sorting - just to add a bit more complexity, in case we didn't have enough already.)

I find it has now become impossible to reason out the consequences of this bug.

I would like it just be fixed, so I can stop having to think about it.

I tried for more than half a year to get Support to understand the issue, but they never did and in the end I gave up. It took months just to get the Support guy once I'd explained and he'd understood *how* sorting values were produced in his excitement at this new understanding to refrain from simply reiterating back to me how the values were being made; he seemed actually and literally to reason that because he could explain *how* they were made and so could duplicate the values found in `STV_BLOCKLIST`, *they must be correct.* Eventually I was able to get him to consider that they might be being made *in the wrong way*, so that even though we could duplicate the working, it could *still* be wrong. At one point he went off to talk to someone else, here I think trying to explain an issue he didn't properly understand to someone else who didn't understand, and they came back to me with a reply something like "this can't be a bug because if it was a bug in sorting it would have a huge impact and everyone would have noticed by now".

At this point I gave up trying.

Then after another two months or so, out of the blue, I was told again but now without any explanation this behaviour was in fact *by design* and the docs would be updated with more information. That was (at the time of writing) two years or so ago, and as far as I know, and I have looked, no update has occurred, and I've not heard from them again since then.

This experience came at a point where I'd already come to be fairly solidly disillusioned with Support, and it was about this time I stopped paying for Support. It wasn't that it was expensive - it's not, AWS ask a token fee only - but that it wasn't worth having *at all*, even if it were free.

Multi-Byte UTF-8 Strings

The `varchar` data type is UTF-8. Redshift however to my knowledge has no actual understanding of Unicode; it doesn't parse or validate, it simply treats strings as bytes (in the C/C++ manner, with 0 being the end-of-string marker). I believe the only differences between `char` and `varchar` as far as encodings go is that for `char` there is a check that the top bit in each byte being clear, and this check is not enforced for `varchar`, and that with `varchar` there are a couple of short byte sequences (presumably varying by Unicode encoding, as `COPY` supports more than UTF-8) which are scanned for and if found cause the string to be rejected. These byte sequences are the forbidden code-points.

Now, quick Unicode refresher. Each character/glyph has a unique integer number - its *code-point* - and the code-points can be represented in different ways, such as UTF-8, UCS-16 and so on. So the actual conceptual value, the code-point, is always the same, but the binary representation varies.

With UTF-8, code-points are represented in one to four bytes; the larger the code-point value, the more bytes are needed. Seven bit ASCII uses one byte, then we go to two bytes for things like Greek and accented characters, the three bytes for the Japanese scripts (all of them), and so on.

In fact, what's done is this, for code-point ranges from "Start" to "End", where the "x" bits are data-bearing bits and the other bits are structural;

Start	End	Binary Representation
0x00000000	0x0000007F	0xxxxxxx
0x00000080	0x000007FF	110xxxxx 10xxxxxx
0x00000800	0x0000FFFF	1110xxxx 10xxxxxx 10xxxxxx
0x00010000	0x001FFFFF	11110xxx 10xxxxxx 10xxxxxx 10xxxxxx

We see then for example with four byte UTF-8, we're consuming 32 bits, but there are only 21 data-bearing bits. The other 11 bits are structural.

So the first thing to note is that where Redshift doesn't know about Unicode, and it just uses the first eight bytes, a lot of the sorting value are the structural bits from Unicode. Redshift could be making better use of the eight bytes it has by only using actual data bits from the string.

Next, it's entirely possible for a multi-byte UTF-8 character to be truncated. Katakana (one of the Japanese scripts) for example are all three bytes. If any of the 125m Japanese out there are using Redshift, and storing katakana, the sorting values will contain the first two characters in full and then the first two bytes of the third character.

A problem here is that the first two bytes of any single katakana matches most of the other katakana characters. There are 96 katakana characters in the main Unicode katakana code page, which are contiguous code-points, starting at 0x30a0. We find for the first 64 characters, the first two bytes of UTF-8 are identical, and for the other characters, the first two byte are identical but the second byte differs from the first 64 characters.

This means those two bytes of the last character in the sorting value are not doing much in the way of selecting a third character; we really only have pretty much only two characters of sorting for katakana strings.

If we were using only the data-bearing bits, we'd be doing better - it's 15 bits per character. we have 8 bytes to play with, which is 64 bits, so we'd get five characters of sorting.

We can demonstrate this by looking at a few Katakana characters;

Katakana	Hex	Bin	UTF-8 binary
゠	0x30a0	11000010100000	11100011 10000010 10100000
ア	0x30a1	11000010100001	11100011 10000010 10100010
ィ	0x30a2	11000010100010	11100011 10000010 10100011
イ	0x30a3	11000010100011	11100011 10000010 10100100

Katakana	Hex	Bin	UTF-8 binary
ゥ	0x30a4	11000010100100	11100011 10000010 10100101

Katakana is left-to-right, same as Latin (and one of the reasons I chose it here for the example - easier to reason about).

So we would have a katakana word, let's make up a garbage word from the few characters above, "アィイ" ("pregnant hippo pizza"). The most significant letter is still on the left, and so this is stored in binary on the right (least significant bits). Redshift however knows nothing about UTF-8, so it's not moving data on a per-character basis (moving three bytes at once, and so reversing on a per-character basis), but simply reversing all the bytes, so we have this;

The original word : "アィイ", is "11100011 10000010 10100010, 11100011 10000010 10100011, 11100011 10000010 10100100"

This is reversed so integer compare works, and so is stored with the most significant letter in the least significant bits, like so (commas between each code-point);

10100100 10000010 11100011, 10100011 10000010 11100011, 10100010 10000010 11100011

And since the sorting value is eight bytes, one byte is thrown away, like so;

10000010 11100011, 10100011 10000010 11100011, 10100010 10000010 11100011

We can see this in the test data on line 8, with a sorting value of 0x82e3a382e3a282e3, which is the hex equivalent of the binary above.

The problem is that the remaining two bytes of the truncated last character, which are "10000010 11100011", match 32 of the 96 katakana characters. (The other 64 are all matched by "10000010 11100011", so if the final character had been one of the other 64, we would be matching 64 characters).

This means sorting is not going to happen in the way you expect.

We can see this in the test data on line 9, for the string "アィゥ". Different final character, same sorting order, even though we have two bytes of that final character in the sorting value.

As an aside, you might be thinking at this point that surely Redshift to compare strings *must* need to compare code-point with code-point, not byte with byte, because if we did compare byte with byte, when we have characters with different numbers of bytes we end up comparing structural bits to data bits and it all goes to hell. In fact, UTF-8 is cunningly designed (this is how you know no Government, standards body or large corporation had anything to do with it) and a byte-by-byte compare *is correct and valid with regard to code-points.*

This does however lead to a problem when you have invalid UTF-8.

Redshift - and this is in general right and proper for a sorted column-store relational database - performs almost no checking on data during ingestion. As such, you can perfectly happily chuck in completely invalid UTF-8 - at which point all bets are off, because its entirely possible some of the structural bits are wrong; you might have say the first two bytes of a four-byte character and then say a byte of all `1s`. Redshift will carry on with its byte-by-byte compare, but the results of that are of course going to be crazy, and in a way which is hard to predict (since you'll be having to think about structural bits as well as data-bearing bits).

Right-to-Left Strings

Where `varchar` is storing Unicode, it is possible to be storing right-to-left scripts.

When you have right-to-left scripts in Unicode, you're faced with the question of the order in which you store code-points.

Now, keep in mind that Redshift has no knowledge of Unicode; it simply stores and retrieves the bytes given to it.

Consider the phrase "dark clouds bring rain" (line 14 in the test data).

This is written in a left-to-right script, and it is stored left-to-right in its text file and in memory (the 'd' of "dark" being at the lowest memory address), where Redshift has no clue about any of this and simply uses the eight bytes starting at the lowest memory address and so in this case does The Right Thing; the sorting value ends up being "dark clo", which is what we want.

What we would *not* want is for Redshift to end up using the *wrong* end of the string, and make the sorting value from "ing rain".

Now consider the same phrase, but in Arabic, which is read right-to-left; "الغيوم الداكنة تجلب المطر".

The most-significant characters on now on the right side, "الغي" (Arabic is two bytes per code-point, so we get four characters in the sorting value, not eight).

At this point we might start to wonder if Redshift is about to use the wrong end of the string for the sorting value.

Well, it turns out all to be okay, because of the Unicode specification.

Unicode states that characters are stored *in the order in which they are read.*

So for left-to-right scripts, our example text, "dark clouds bring rain", are stored in the order in which we read, from left-to-right, something we're intuitively familiar with.

For right-to-left scripts, our example text (line 10 in the test data), "الغيوم الداكنة تجلب المطر", where we read it from the right hand side first, has the right side characters being stored *first.* So, if we look at what's in a Unicode encoded file storing this string, it is the characters, one-by-one, starting from the right hand side, and then proceeding to the left.

Redshift of course is oblivious to this : it simply uses the first eight bytes it reads from the file - but because Unicode specifies right-to-left scripts are also stored right-to-left, what Redshift is doing *still* picks up the Arabic equivalent of “dark clo” (well, just “dark” - two bytes per character for Arabic).

We can see this on lines 10 and 11. On line 10, we have “الغيوم الداكنة تحجب المطر”, on line 11 we have only the first three (which is to say, the right-most) words, “الغيوم الداكنة تحجب”. If Redshift is doing the right thing, the sorting value for these two strings will be identical; if Redshift is doing the wrong thing, the sorting values will differ because the wrong end of the string is being used - and we can see the sorting values are indeed identical.

Just to show what happens when the wrong thing does happen, on lines 12 and 13, I’ve taken the example Arabic string (both the full four-word string on line 12, and the first three words only on line 13) and stored it left-to-right. Here we do see the sorting values differ, and this is as described occurring because the wrong end of the string is now being used to produce the sorting value.

The same is in done in Latin on lines 14 to 17, to make it easier to perceive.)

So all is well, except for one tiny thought… given that `varchar` simply stores whatever you give it, regardless of Unicode[1], I wonder if there are people out there storing *non*-Unicode encoding in `varchar`s. If you are, or if you have a client, who is doing this, and it’s a right-to-left encoding, remember that for such encodings you also must store the characters right-to-left, or you’ll end up storing by the wrong end of the string.

And, Finally…

First, note that `char` is 7-bit ASCII, and Redshift checks when `char` values are inserted that the top bit is not set, and will fail the insert if it is. As such, the sorting value for `char` wastes one byte of data; as with the structural bits in UTF-8, we could be doing better.

Second, note (test data line 18) `NULL` has strange values for `char` and `varchar`. Usually, as you will see with the other data types, `NULL` is given the maximum possible sorting value (9223372036854775807). With strings, though, there are again *two* sorting values - different for minimum and maximum - and the values are 7fffffffffffffff and 8000000000000000, respectively.

I have no idea why.

date

`date` is a four byte data type, and so should be fully represented in the sorting value.

The minimum value Redshift accepts is `4713-01-01 BC`. The maximum value Redshift accepts is `5874897-12-31 AD` (the docs claim the maximum is `294276 AD`, with no mention of dates, but in any event this is incorrect).

[1]There’s a slight caveat to this. Redshift I think scans the byte-stream, and rejects certain sequences of bytes, which represent forbidden code-points. I do not think Unicode decoding is occurring; it’s just a scan of the byte stream. However, this could interfere with non-Unicode encodings.

We see from line 1 and line 8 that these values result in a sorting value which ranges from -2451507 to 2145031948, with each day changing the sorting value by 1.

We also see `NULL` has a unique value.

Also of interest is that the sorting value of 0 is generated for `2000-01-01 AD`, which is a date found in a number of places in the system tables, where it means in fact there is no actual date information - it's basically a special value, where we have to hope there's never a *real* value on the same date :-)

In short, `date` works as intuitively expected, and has no unexpected behaviour.

float4 and float8

A `float4` is four bytes and a `float8` is eight bytes, and so both should be fully represented in the sorting value.

This does not in fact occur.

What actually happens in both cases is that the sorting value is the integer part only of the value. The fractional part is discarded.

(Note the test data for `float4` and `float8` are the same, so any test data line numbers are valid for both data types.)

Looking to the test data at lines 6 and 7, we see that -50.75 and -50 both have the sorting value of -50.

Similarly, looking lines 10 to 12, we see if either float type is being used to store percentages as values in the range 0 to 1, all the sorting values (except of course for 1.0) will be 0.

This method for deriving the sorting value inherently causes a further problem.

The number range of a `float4` is -3.4e-38 to 3.4e38.

The number range of a `float8` is -1.7e-308 to 1.7e308.

The number range of a signed eight byte integer (the sorting value) is -9.2e18 to 9.2e18.

In other words, floats can store much larger, and much smaller, values than sorting values.

So the question then is what happens when the integer part of the float is smaller or larger than can be represented by a sorting value?

What happens is that the sorting value is clamped to the minimum and maximum value the sorting value can represent.

Note there's a slight complication in this : the more a floating point value differs from zero (positively or negatively) the larger the inherent floating point inaccuracy becomes. For those of you not familiar with this, think of floating point numbers as being like a wire fence with posts in. The wire is the infinitely precisely continuum of numbers, but you are only allowed to store the numbers at the posts. If you store a number between two posts, it is *moved* - changed -

to be the number at the nearest post. So if yous store, say, 550,000,000,000.5, the number which is *actually* stored, and which you will get back when you read it, is different, and might be, say, 550,000,000,341.62. As numbers get larger or smaller, the gaps between the posts become larger. The distance between the posts is much larger for `float4` than it is for `float8`.

When we're at the minimum and maximum values for the sorting value, the inaccuracy for a `float4` is 274,877,907,455, for a `float8` is 511.

This means that once we get to within that inaccuracy of the minimum or maximum sorting value, then the sorting value will already be at it's maximum or minimum (and be being clamped).

To put this into numbers, it means that all `float4` values equal to or below -9223371761976868352.0, and all `float8` values equal to or below -9223372036854775296.0, have the same sorting value (the minimum sorting value), of -9223372036854775808.

We can see this on lines 3 and 4, and 16 and 17 of the test data. Changing the value by 1 results in a change of about 550,000,000,000 in the sorting value; but on lines8, 10 and 12, we see changing the value by 1 only results in a change of 1 in the sorting value.

Similarly, for positive values, all `float8` values equal to or above 9223371761976868352.0, and all `float8` values equal to or above 9223372036854775296.0, have the same sorting value (the minimum sorting value), of 9223372036854775807.

Data Type	Minimum	Maximum
`float4`	-9223371761976868352.0	9223371761976868352.0
`fliest8`	-9223372036854775296.0	9223372036854775296.0

Bear in mind now the minimum and maximum `float4` value is 3.402823e+38 (nineteen more zeros than the min/max sorting value) and for `float8` is 1.7e+308 (two hundred and eighty-nine more zeros), you can see most of the number range of the floats have the same sorting value.

We can see this behaviour for both `float4` and `float8` on lines 3 and 4, and 16 and 17.

Now, I may be wrong, but I think there is a much better - indeed, the normal - method for sorting floats, and if this is correct, then the current method, of using the integer part of the sorting value, is a *serious* design blunder.

So; floating point values are stored according to the IEEE-754 specification. With this specification numbers are stored in two parts (within the four or eight bytes in use), known as the mantissa and the exponent.

The mantissa is - to put is loosely but hopefully descriptively - the first few non-zero digits in the number, and the exponent is the number of zeros to add, either before or after those digits.

So a mantissa might be say 58512 and the exponent might be 6 - giving the number 58,512,000,000.

Now, when you take the bit pattern formed by numbers in IEEE-754 representation and sort them as if they were normal unsigned two's complement signed integers, what you find is that the positive numbers sort correctly, but the negative numbers have their sorting order reversed.

However, if you `XOR` with 0, then all the numbers sort correctly.

Why is Redshift not doing this?

If it did, then floating point numbers would be fully represented in the sorting value, and so would have no clamping, and would also express their fractional part in the sorting order, and so and would behave as intuitively expected.

Finally, changing subject, we see that the special number `NaN` ("Not A Number") has the sorting value -9223372036854775808 (the smallest possible sorting value), and `NULL`, as usual, has the largest possible sorting value, 9223372036854775807.

In both cases these values are also being used by other numbers (in fact of course with floats, by *many* other numbers, since the min and max sorting values are the sorting values most of the number range is clamped to), so they are not unique.

In short, `float4` and `float8` work in a completely unexpected manner, which has serious issues, and indeed in a manner which may represent a serious design blunder.

int2, int4 and int8

The `int` types are simple. Their value is used as the sorting value and as such they behave almost entirely as intuitively expected.

The single exception is that `NULL` for `int8` does not have a unique value, as it is as usual given the maximum possible sorting value, which is 9223372036854775807. The other integer types never use the maximum sorting value, so for them `NULL` has a unique value.

numeric

The first (and as ever, undocumented) fact to know is that the `numeric` type comes in two forms; an eight byte form and a sixteen byte form. Both are stored as normal two's compliment signed integer values. It is and only is the DDL which determines the form is used; the actual values being stored is *not* involved. It is simply that if in the DDL the `numeric` precision is 19 or less, eight bytes are used. If precision is 20 or more, sixteen.

(Note if you talk to Support about how `numeric` is encoded, they explain it behaves like `varchar`. By this they mean that you in the DDL specify the maximum precision but that values when stored actually only use as much store as they need - in exactly the same way as you specify a maximum length for a `varchar`, but then if you store a string shorter than this, only the actual store needed for the string is consumed. They then point you at a page in the Postgres docs, here, which states;

> "The actual storage requirement is two bytes for each group of four decimal digits, plus eight bytes overhead."

This information is correct for Postgres but incorrect for Redshift. Redshift uses eight bytes or sixteen bytes, depending purely on the DDL. Support are distributing incorrect and misleading information.)

We need to consider the behaviour of the eight byte and sixteen byte `numeric` separately, as they differ.

Before we do so, a brief aide memoir may be useful, with regard to how `numeric` works : *all* values are stored as integers, but Redshift by knowing from the DDL the scale can figure out where the decimal point should go when it comes to actually displaying or doing work with the value.

Eight Byte Form

We have two sets of test data for eight-byte `numeric`, one for `numeric(18,0)` and one for `numeric(18,4)`.

Where the data type is using eight bytes, it should be fully represented in the sorting value, and this is what we find.

The actual value being stored for the `numeric` is an eight byte integer, and this is directly used - in the same way as an `int8` - as the sorting value.

Unexpectedly, however, the special value `NaN` (Not A Number) has the sorting value 0. With the `float` types, `NaN` is stored as -9223372036854775808 (the smallest possible sorting value).

(One note about `numeric` with a precision of 19. According to the usual way we think of a numeric the maximum value should be 19 digits of 9s, e.g. 9,999,999,999,999,999,999. However, where this data type is actually under the hood a signed eight byte integer, the maximum value for `numeric(19,n)` is 9,223,372,036,854,775,807 (the maximum for a signed eight byte integer). The documentation writes about this, but it does not explain why.)

Moving on to `numeric(18,4)`, we see the seemingly fractional values, such as -15.5000 on line 2, are really stored as -155000, which is an integer, and that -155000 is indeed being used directly as the sorting value.

This is the right thing to do, and as such, values are being sorted as would be intuitively expected.

Unexpectedly, trying to store `NaN` in a `numeric` with a non-zero scale results in an assert. This is likely a bug.

```
-----------------------------------------------
error:  Assert
code:      1000
context:   numeric_scale(preset_size) == (numeric_scale(size)) -
query:     2668
location:  step_project.cpp:251
process:   padbmaster [pid=1791]
-----------------------------------------------
```

(This is why there is no `NaN` test data for `numeric(18,4)`.)

`NULL` as usual has the sorting value 9223372036854775807, the largest possible sorting value.

In short, eight byte `numeric` works as intuitively expected, and has no unexpected behaviour, except for the minor matter of `NaN` being 0, and throwing an assert if scale is non-zero.

Sixteen Byte Form

Now things get a little tricky for Redshift. It's dealing with a sixteen byte integer, but the sorting value is an eight byte integer. What happens?

The answer is twofold;

1. For values in the range of a signed eight byte integer, the sorting value is the lower eight bytes of the value, and so behaviour is what we would intuitively expect.
2. For values outside of this range, the sorting value is the upper eight bytes of the value, and so every 2^64 values has the same sorting value, *and* all these sorting values overlap with the sorting values produced for the values in the range of a signed eight byte integer.

Looking to the results to give example, we see from lines 7 and 16 inclusive examples of case #1; the lower eight bytes are in use.

Lines 1 to 6, and 18 to 23, show the upper eight bytes in use, and we see then sorting values for positive numbers begin at 0 and increment every 2^64 values, and for negative values, sorting values begin at -1 and decrement every 2^64 values.

In other words, large blocks of numbers have the same sorting value.

This is unexpected, and *of course* not documented, but it is understandable : there are only eight bytes in the sorting value, so it's hard to see what else could be done. However, it should absolutely, like so much that is not, have been documented.

Finally, we can turn to the minimum and maximum possible values, which are -99,999,999,999,999,999,999,999,999,999,999,999,999 and 99,999,999,999,999,999,999,999,999,999,999,999,999, respectively (38 digits of 9). The sorting values are -5421010862427522171 and 5421010862427522170, respectively, which shows you just how large these numbers are - there are 5421010862427522170 lots of 2^64 numbers in both the negative and positive directions…

So, in short, sixteen byte `numeric` does not work as intuitively expected. Large number ranges have the same sorting value, and this is not something which users could be expected to figure out on their own.

time

`time` is an eight byte value, and so should be fully represented in the sorting value.

We can see that this is the case. The value range for the data type is `00:00:00.000000` to `23:59:59.999999`, and these values give sorting values of 0 and 86399999999, respectively.

We see from lines 1 and 2 that adding one microsecond increases the sorting value by 1, and from line 7 that `NULL` has a unique value.

In short, `time` works as intuitively expected, and has no unexpected behaviour.

timetz

Redshift does not really have timezone support and as such this data type is a superficial wrapper to the `time` data type.

What's unexpected is that Redshift does *not* store timezone information. What actually happens when you populate a `timetz` is that you provide a time and a timezone, and the timezone is applied to the time to convert it to `UTC`, and this is stored.

The docs never actually come out and say this - actually, it's worse than that; the docs misled you to thinking the timezone *is* stored.

> "Use the TIMETZ data type to store the time of day with a time zone."

I mean if you read that, and you're coming to this thinking - as you will be - that this is a data type which supports timezones, what are you going to think? You're going to think the timezone is being stored. I see a constant low rate of questions on-line where people are flummoxed by it, as they are expecting Redshift to be storing the timezone.

All time and date values in Redshift are treated as and assumed to be UTC, because of the absence of timezone information. Accordingly, there is no way for you, unless you yourself store the original timezone, to go back to the timezone the original time or date was in.

This approach unfortunately leads to a fundamental problem with the derivation of the sorting value for `timetz`.

If we look back to the `time` type, we see that it represents one day, and each microsecond represents a sorting value of 1.

Now, the `timetz` type is only a wrapper for the `time` type; `timetz` simply and only causes a modification to be performed on the time before it is stored in a `time`, converting it to UTC. So, for example, the time `12:00` with the timezone UTC-2, will be converted to `14:00` and that is the time which will be stored.

The killer problem is that if the timezone moves the time to be before `00:00:00.000000` or after `23:59:59.999999`, the time does not move into another day - there *is* no other day, because a `time` only can represent a single day - so it gets *wrapped*.

So, for example, if you have the time `22:00:00` and the timezone UTC-4, this becomes `02:00:00`, and so has the same sorting value as all times at `02:00:00` in UTC (or at `03:00:00` in UTC-1, or `04:00:00` in UTC-2, etc). This is shown on lines 5 and 6 of the test data.

You can now see the problem - many different times unexpectedly end up with the same sorting value.

What's actually happening is that sorting is occurring on the basis of the UTC time, regardless of day, rather than occurring on the basis of the timezone and then the time.

This is why it's impossible to evoke a negative sorting value; the `time` type doesn't have negative sorting values (`00:00:00.000000` is 0, `23.59.59.999999` is 86399999999, with each microsecond adding 1 to the sorting value), and the `timetz` is mapping every time into the period of a single day, and so every time ends up being between `00:00:00.000000` and `23.59.59.999999`.

The question then is the impact this has the efficiency of the zone map.

Where `timetz` throws away the timezone, what we're looking at is of course identical to going from a compound sorted table with two columns, the first for timezone and the second for time, to a compound sorted table with one column, which is the time converted into UTC and wrapped into a single day.

Let's imagine we want to perform a scan, based on a given time and timezone.

In the latter case, the first column means we're now only looking at blocks with times from our timezone; we then narrow down the actual blocks we're going to scan based on the blocks which contain times which happened at about when our time occurred.

In the former case, we have only the one column, so we're scanning times from *all* timezones where those times happened to occur in UTC at about the same time as the event we're looking for.

In other words, as we might expect, loss of information (the timezone) reduces our ability to narrow a search.

I regard the wrapping of times into one day, and its impact on sorting, as unexpected behaviour. I do not think users will expect `14:00:00` Eastern Time to have the same sorting value as `21:00` Central European Time, and it's hard to reason about. It is easiest to think about timezones sorting first, and then times afterwards; rather than trying in your head to remember the offset of each timezone from UTC and apply that to each time and so come up with the UTC time and so the sorting value. I am not a robot.

Note that in real life (setting aside daylight saving, because it varies by country and legislation, which varies over time, so it's complex) the largest negative timezone is -14 and the largest positive timezone is +12.

However, the `timetz` accepts a timezone ranging from -1459 to +1459 hours (yes - one thousand, four hundred and fifty nine hours, which is a bit more than two months). This is shown on lines 8 and 9 of the test data. The sorting values just happen to end up being whatever they are given whatever number of days of "wrapping round" which occurs.

These large values should never be used - to do so will always be an error - and so they should not be supported.

Finally, `NULL` has a unique sorting value (test data line 7).

So, with regard to `timetz`, it may be sorting by UTC works fine for a given use case, but when it does not, you will need to use two columns; a normal `time` column and a `smallint` where you store the timezone. Put both in the compound sortkey, with the timezone column first. This will get you the expected behaviour.

In short, `timetz` does not work as intuitively expected, and I advise this data type is not used, but replaced with a manual two-column solution.

I am rather of the view this data type should not have been implemented, with instead there simply being a page advising and explaining the use of one column for timezone and one for time, and I am certainly of the view that having been implemented as it has been, its behaviour should have been documented.

timestamp

`timestamp` is an eight byte value, and so should be fully represented in the sorting value, and this is what we find.

As with `time`, each microsecond difference in value adjusts the sorting value by 1 (test data line 5 and 6).

The timestamp `2000-01-01 00:00:00.000000`, as with `date`, gives the sorting value of 0 (line 5).

The minimum timestamp is `4713-01-01 00:00:00.000000 BC`, and this gives the smallest attainable sorting value, of -211810204800000000 (line 1). There's a lot of negative sorting value range still available, so in principle the minimum timestamp could have been much further back in time.

The maximum timestamp is `294276-12-31 23:59:59.999999 AD` (line 13), which gives the sorting value 9223371331199999999. We can see from line 12 that one year of values consumes 31622400000000 from the sorting value range, and we can see that 9223371331199999999 + 31622400000000 = 9223402953599999999, which is larger than the maximum sorting value of 9223372036854775807 by 30916745224192. In other words, the maximum timestamp value is the largest complete year which can be represented uniquely in the sorting value. It's not clear to me why this wasn't also done for years BC, but there you go; perhaps the developers are not ancient history buffs.

Finally, we see `NULL` has a unique sorting value (line 14).

In short, `timestamp` works as intuitively expected, and has no unexpected behaviour.

timestamptz

As with `timetz`, `timestamptz` is a thin veneer over a normal `timestamp`. As before what happens is that the timezone is specified with the timezone, the timezone is applied to the timestamp to convert it to UTC, and the timestamp is stored. The timezone information is lost; there is no way to find out what timezone the timestamp was originally in.

Again, as with `timetz`, we see on lines 2 and 4, that the sorting value of `1999-12-31 23:00:00.000000+0 AD` and `2000-01-01 00:00:00.000000+1 AD` are the same; which is to say, sorting is ends up actually based on the UTC timestamp.

The first matter of note of course is that now we're working with a timestamp, not just a time, this operation to adjust the timestamp according to its timezone can change the date of a timestamp, by dint of the timezone when applied moving the timestamp into the next or previous day.

That can be imagined to lead to problems; someone might know the date of an event, but not know the timezone and entirely reasonably not know they would *need* to know the timezone, and so not be able to find the event.

The main consideration though is sorting, since this determines how effectively blocks will be culled by the zone map, which is absolutely central to performance with Big Data.

Where `timestamptz` throws away the timezone, what we're looking at is of course identical to going from a compound sorted table with two columns, the first for timezone and the second for timestamp, to a compound sorted table with one column, which is the timestamp converted into UTC.

Let's imagine we want to perform a scan, based on a given timestamp and timezone.

In the latter case, the first column means we're now only looking at blocks with timestamps from our timezone; we then narrow down the actual blocks we're going to scan based on the blocks which contains timestamps which happened at about when our timestamp occurred.

In the former case, we have only the one column, so we're scanning timestamps from *all* time-zones where those timestamps happened to occur in UTC at about the same time as the event we're looking for.

In other words, as we might expect, loss of information (the timezone) reduces our ability to narrow a search.

We can also see this approach is problematic to reason about. I must, in my mind, convert all my times and time zones to UTC, to figure out their sorting order; so if I have say the timestamps `2021-06-05 14:35:20 Alma-Ata Time`, `2021-06-06 05:55:18 Australian Eastern Time`(note this the day after), and `2021-06-05 01:30:00 Eastern Time`, what order will they sort in?

Speaking as a human, "does not compute".

Of course, given that a `timestamp` is an eight byte type, it's hard to see what else could be done; but then I would actually argue this data type should not have been introduced in the first place. It does not properly perform its job, but by its existence will lead developers into using it, and as a result they are writing inferior systems.

What should have happened is that the documentation should have *explained* that you will need two columns, one to store the timezone, one to store timestamps.

To facilitate this support could have been added for a *timezone* data type.

Moving on, we see all the other attributes of this data type (which timestamp produces a sorting value of 0, minimum and maximum values, that the minimum and maximum timezone difference is over two months worth of hours, and so on) are, of course, identical to that of `timestamp`, since `timestamptz` is under the hood actually a `timestamp`.

In short, `timestamptz` does not work as intuitively expected, and I advise this data type is not used, but replaced with a manual two-column solution.

(The existing Redshift support for timezone *names* can be used to figure out the numeric timezone of a timestamp, by creating a `timestamptz` of a timestamp with timezone, and then subtracting from that the timestamp *without* the timezone.)

Conclusions

Of the data types examined in this document, we observe then some behave as we would intuitively expected, and some do not.

Data Type	Intuitively Correct
char	✗
varchar	✗
date	✓
float4	✗
float8	✗
int2	✓
int4	✓
int8	✓
numeric (eight byte form)	✓
numeric (sixteen byte form)	✗
time	✓
timetz	✗
timestamp	✓
timestamptz	✗

Knowing how sorting values are derived is necessary to know what a Redshift system is doing when scans and joins occur, which in turn is necessary to knowingly correctly design a system using Redshift.

The way in which you would naturally think `date`, the `int`s, `numeric` with precision 19 or less, `time` and `timestamp` to sort is indeed what they do and is technically right and correct.

The other data types behave in entirely unexpected and unexpectable ways, such that system using these data types with the expectation they sort as you would intuitively expect are incorrectly designed.

The `char` and `varchar` data types are by far the most complex to understand, in part as strings (the values) and integers (the sorting values) sort differently and this must be handled by Redshift, as Unicode leads to certain matters (such as right-to-left scripts) which developers need to be aware of, and there being one probably minor bug in how sorting values are generated (which leads to more blocks being read than would otherwise need to be).

The `float4` and `float8` types derive their sorting values by using the integer part only of the value, which leads to fractional percentages (values in the range 0 to 1) all having the same sorting value (except of course for 1.0, but that hardly saves you :-), and which also leads to clamping, where values which are below or above the minimum and maximum sorting value are clamped to the minimum and maximum sorting values, respectively. It seems to me possible that this approach might be a serious blunder, and that there is actually a different and in fact the usual and proper way to sort floats, which simply hasn't been used.

The `numeric` type is either eight bytes or sixteen bytes, depending purely on the DDL. When precision is 19 or less, the `numeric` will be eight bytes, when 20 or more, sixteen. The sixteen byte form of `numeric` over the value range of a signed eight byte integer, uses the lower eight bytes of the value to form the sorting value; outside of this range, it uses the upper eight bytes. In the former, behaviour is intuitively correct, but in the latter, each block of 2^64 values has the same sorting value, and these overlap with the sorting values produced in the former case.

The `timetz` data type is highly non-intuitive. Where Redshift does not truly support timezones, what this type actually does is apply the time-zone to the time, to convert it to UTC, and then store the UTC time. The time-zone is thrown away after that, and so there is no way to know the original time-zone of the time. The problem with this is that if the time-zone moves the time into the previous or next day, the time simply *wraps around*. `12:00:00 UTC` is stored as the same value as `14:00:00 UTC+2`; and where these times and time-zones end up with the same value being stored, they naturally end up with the same sorting value.

Much the same problem is found with `timestamptz`, where again the time-zone is not stored, but rather applied to convert the timestamp to UTC (which also means the date of the timestamp can unexpectedly change), and so different times in different time-zones end up with the same sorting value. This is problematic to reason about, and reduces the effectiveness of the Zone Map.

With both the time-zone type data types, I advise they are not used, it's better to roll your own solution, using two columns, one for the time-zone and one for the time/timestamp.

Unexpected Findings

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

1. `numeric` data types with a non-zero scale throw an assert when set to `NaN`.

Further Questions

1. How are sorting values derived for `hllsketch` and `super` types?

Revision History

v1

- Initial release.

v2

- Metadata changes. No content changes.

v3

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

v4

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

v5

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

Appendix A : Raw Data Dump

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

PLEASE NOTE due to technical limitations of the XeLaTeX parser, the raw results are being processed as raw text. As a result, the Arabic and Katana text is not being displayed. If you do use the data here, you will need once you have copied it to insert, for the `varchar` data type, the missing Arabic text, which *is* correctly displayed in the Results section.

```
{'proofs': {'dc2.large': {2: {'char(64)': [("'MARTINEZ'",
                                              [6504691313660805453,
                                               '5a454e495452414d',
                                               6504691313660805453,
                                               '5a454e495452414d',
                                               1]),
                                             ("'MURPHY'",
                                              [98167120090445,
                                               '59485052554d',
                                               2314948375588525389,
                                               '202059485052554d',
                                               1]),
                                             ("'abcdefg'",
                                              [29104508263162465,
                                               '67666564636261',
                                               2334947517476856417,
                                               '2067666564636261',
                                               1]),
                                             ("'\ttuvwxyz'",
                                              [8825217399293047817,
                                               '7a79787776757409',
                                               8825217399293047817,
                                               '7a79787776757409',
                                               1]),
                                             ("'tuvwxyz'",
                                              [34473505465988468,
                                               '7a797877767574',
                         2340316514679682420,
                         '207a797877767574',
                         1]),
                       ("'1234567 1234567'",
                        [15540725856023089,
                         '37363534333231',
                         2321383735069717041,
                         '2037363534333231',
                         1]),
                       ("'1234567a1234567'",
                        [7005127347535032881,
                         '6137363534333231',
                         7005127347535032881,
                         '6137363534333231',
                         1]),
                       ('NULL',
                        [9223372036854775807,
                         '7fffffffffffffff',
                         -9223372036854775808,
                         '8000000000000000',
                         1])],
         'date': [("'4713-01-01 BC'",
                   [-2451507,
                    'ffffffffffda97cd',
                    -2451507,
                    'ffffffffffda97cd',
                    1]),
                  ("'0001-12-31 BC'",
                   [-730120,
                    'fffffffffff4dbf8',
                    -730120,
                    'fffffffffff4dbf8',
                    1]),
                  ("'0001-01-01 AD'",
                   [-730119,
                    'fffffffffff4dbf9',
                    -730119,
                    'fffffffffff4dbf9',
                    1]),
                  ("'1999-12-31 AD'",
                   [-1,
                    'ffffffffffffffff',
                    -1,
                    'ffffffffffffffff',
                    1]),
                  ("'2000-01-01 AD'", [0, '0', 0, '0', 1]),
                  ("'2000-01-02 AD'", [1, '1', 1, '1', 1]),
                  ("'5874897-12-31 AD'",
                   [2145031948,
                    '7fda970c',
                     2145031948,
                     '7fda970c',
                     1]),
                    ('NULL',
                     [9223372036854775807,
                     '7fffffffffffffff',
                     -9223372036854775808,
                     '8000000000000000',
                     1])],
 'float4': [("'NaN'",
             [-9223372036854775808,
              '8000000000000000',
              -9223372036854775808,
              '8000000000000000',
              1]),
            ('-3.402823e+38',
             [-9223372036854775808,
              '8000000000000000',
              -9223372036854775808,
              '8000000000000000',
              1]),
            ('-9223371761976868352.0',
             [-9223372036854775808,
              '8000000000000000',
              -9223372036854775808,
              '8000000000000000',
              1]),
            ('-9223371761976868351.0',
             [-9223371487098961920,
              '8000008000000000',
              -9223371487098961920,
              '8000008000000000',
              1]),
            ('-1234.56789',
             [-1234,
              'fffffffffffffb2e',
              -1234,
              'fffffffffffffb2e',
              1]),
            ('-50.75',
             [-50,
              'ffffffffffffffce',
              -50,
              'ffffffffffffffce',
              1]),
            ('-50.0',
             [-50,
              'ffffffffffffffce',
              -50,
              'ffffffffffffffce',
              1]),
            ('-1',
             [-1,
              'ffffffffffffffff',
              -1,
              'ffffffffffffffff',
              1]),
            ('-0.5', [0, '0', 0, '0', 1]),
            ('0', [0, '0', 0, '0', 1]),
            ('0.5', [0, '0', 0, '0', 1]),
            ('1', [1, '1', 1, '1', 1]),
            ('50.0', [50, '32', 50, '32', 1]),
            ('50.75', [50, '32', 50, '32', 1]),
            ('1234.56789',
             [1234, '4d2', 1234, '4d2', 1]),
            ('9223371761976868351.0',
             [9223371487098961920,
              '7fffff8000000000',
              9223371487098961920,
              '7fffff8000000000',
              1]),
            ('9223371761976868352.0',
             [9223372036854775807,
              '7fffffffffffffff',
              9223372036854775807,
              '7fffffffffffffff',
              1]),
            ('3.402823e+38',
             [9223372036854775807,
              '7fffffffffffffff',
              9223372036854775807,
              '7fffffffffffffff',
              1]),
            ('NULL',
             [9223372036854775807,
              '7fffffffffffffff',
              -9223372036854775808,
              '8000000000000000',
              1])],
 'float8': [("'NaN'",
             [-9223372036854775808,
              '8000000000000000',
              -9223372036854775808,
              '8000000000000000',
              1]),
            ('-1.7e+308',
             [-9223372036854775808,
              '8000000000000000',
              -9223372036854775808,
              '8000000000000000',
              1]),
            ('-9223372036854775296.0',
             [-9223372036854775808,
              '8000000000000000',
              -9223372036854775808,
              '8000000000000000',
              1]),
            ('-9223372036854775295.0',
             [-9223372036854774784,
              '8000000000000400',
              -9223372036854774784,
              '8000000000000400',
              1]),
            ('-1234.56789',
             [-1234,
              'fffffffffffffb2e',
              -1234,
              'fffffffffffffb2e',
              1]),
            ('-50.75',
             [-50,
              'ffffffffffffffce',
              -50,
              'ffffffffffffffce',
              1]),
            ('-50.0',
             [-50,
              'ffffffffffffffce',
              -50,
              'ffffffffffffffce',
              1]),
            ('-1',
             [-1,
              'ffffffffffffffff',
              -1,
              'ffffffffffffffff',
              1]),
            ('-0.5', [0, '0', 0, '0', 1]),
            ('0', [0, '0', 0, '0', 1]),
            ('0.5', [0, '0', 0, '0', 1]),
            ('1', [1, '1', 1, '1', 1]),
            ('50.0', [50, '32', 50, '32', 1]),
            ('50.75', [50, '32', 50, '32', 1]),
            ('1234.56789',
             [1234, '4d2', 1234, '4d2', 1]),
            ('9223372036854775295.0',
             [9223372036854774784,
              '7ffffffffffffc00',
              9223372036854774784,
              '7ffffffffffffc00',
              1]),
            ('9223372036854775296.0',
             [9223372036854775807,
              '7fffffffffffffff',
              9223372036854775807,
              '7fffffffffffffff',
              1]),
            ('1.7e+308',
             [9223372036854775807,
              '7fffffffffffffff',
              9223372036854775807,
              '7fffffffffffffff',
              1]),
            ('NULL',
             [9223372036854775807,
              '7fffffffffffffff',
              -9223372036854775808,
              '8000000000000000',
              1])],
 'int2': [('-32768',
           [-32768,
            'ffffffffffff8000',
            -32768,
            'ffffffffffff8000',
            1]),
          ('-100',
           [-100,
            'ffffffffffffff9c',
            -100,
            'ffffffffffffff9c',
            1]),
          ('-1',
           [-1,
            'ffffffffffffffff',
            -1,
            'ffffffffffffffff',
            1]),
          ('0', [0, '0', 0, '0', 1]),
          ('1', [1, '1', 1, '1', 1]),
          ('100', [100, '64', 100, '64', 1]),
          ('32767',
           [32767, '7fff', 32767, '7fff', 1]),
          ('NULL',
           [9223372036854775807,
            '7fffffffffffffff',
            -9223372036854775808,
            '8000000000000000',
            1])],
 'int4': [('-2147483648',
           [-2147483648,
            'ffffffff80000000',
            -2147483648,
            'ffffffff80000000',
            1]),
          ('-100',
           [-100,
            'ffffffffffffff9c',
            -100,
            'ffffffffffffff9c',
            1]),
          ('-1',
           [-1,
            'ffffffffffffffff',
            -1,
            'ffffffffffffffff',
            1]),
          ('0', [0, '0', 0, '0', 1]),
          ('1', [1, '1', 1, '1', 1]),
          ('100', [100, '64', 100, '64', 1]),
          ('2147483647',
           [2147483647,
            '7fffffff',
            2147483647,
            '7fffffff',
            1]),
          ('NULL',
           [9223372036854775807,
            '7fffffffffffffff',
            -9223372036854775808,
            '8000000000000000',
            1])],
 'int8': [('-9223372036854775808',
           [-9223372036854775808,
            '8000000000000000',
            -9223372036854775808,
            '8000000000000000',
            1]),
          ('-100',
           [-100,
            'ffffffffffffff9c',
            -100,
            'ffffffffffffff9c',
            1]),
          ('-1',
           [-1,
            'ffffffffffffffff',
            -1,
            'ffffffffffffffff',
            1]),
          ('0', [0, '0', 0, '0', 1]),
          ('1', [1, '1', 1, '1', 1]),
          ('100', [100, '64', 100, '64', 1]),
          ('9223372036854775807',
           [9223372036854775807,
            '7fffffffffffffff',
            9223372036854775807,
            '7fffffffffffffff',
            1]),
          ('NULL',
           [9223372036854775807,
            '7fffffffffffffff',
            -9223372036854775808,
            '8000000000000000',
            1])],
'numeric(18,0)': [('-999999999999999999',
                   [-999999999999999999,
                    'f21f494c589c0001',
                    -999999999999999999,
                    'f21f494c589c0001',
                    1]),
                  ('-15',
                   [-15,
                    'fffffffffffffff1',
                    -15,
                    'fffffffffffffff1',
                    1]),
                  ('-1',
                   [-1,
                    'ffffffffffffffff',
                    -1,
                    'ffffffffffffffff',
                    1]),
                  ('0', [0, '0', 0, '0', 1]),
                  ("'NaN'", [0, '0', 0, '0', 1]),
                  ('1', [1, '1', 1, '1', 1]),
                  ('15', [15, 'f', 15, 'f', 1]),
                  ('999999999999999999',
                   [999999999999999999,
                    'de0b6b3a763ffff',
                    999999999999999999,
                    'de0b6b3a763ffff',
                    1]),
                  ('NULL',
                   [9223372036854775807,
                    '7fffffffffffffff',
                    -9223372036854775808,
                    '8000000000000000',
                    1])],
'numeric(18,4)': [('-99999999999999.9999',
                   [-999999999999999999,
                    'f21f494c589c0001',
                    -999999999999999999,
                    'f21f494c589c0001',
                    1]),
                  ('-15.5000',
                   [-155000,
                    'fffffffffffda288',
                    -155000,
                    'fffffffffffda288',
                    1]),
                  ('-15.0000',
                   [-150000,
                    'fffffffffffdb610',
                    -150000,
                    'fffffffffffdb610',
                    1]),
                  ('-15',
                   [-150000,
                    'fffffffffffdb610',
                    -150000,
                    'fffffffffffdb610',
                    1]),
                  ('-1.0000',
                   [-10000,
                    'ffffffffffffd8f0',
                    -10000,
                    'ffffffffffffd8f0',
                    1]),
                  ('0.0000', [0, '0', 0, '0', 1]),
                  ('1.0000',
                   [10000,
                    '2710',
                    10000,
                    '2710',
                    1]),
                  ('15',
                   [150000,
                    '249f0',
                    150000,
                    '249f0',
                    1]),
                  ('15.0000',
                   [150000,
                    '249f0',
                    150000,
                    '249f0',
                    1]),
                  ('15.5000',
                   [155000,
                    '25d78',
                                      155000,
                                      '25d78',
                                      1]),
                                    ('99999999999999.9999',
                                     [999999999999999999,
                                      'de0b6b3a763ffff',
                                      999999999999999999,
                                      'de0b6b3a763ffff',
                                      1]),
                                    ('NULL',
                                     [9223372036854775807,
                                      '7fffffffffffffff',
                                      -9223372036854775808,
                                      '8000000000000000',
                                      1])],
                  'numeric(38,0)': [('-99999999999999999999999999999999999999'
                                     [-5421010862427522171,
                                      'b4c4b357a5793b85',
                                      -5421010862427522171,
                                      'b4c4b357a5793b85',
                                      1]),
                                    ('-36893488147419103233',
                                     [-3,
                                      'fffffffffffffffd',
                                      -3,
                                      'fffffffffffffffd',
                                      1]),
                                    ('-36893488147419103232',
                                     [-2,
                                      'fffffffffffffffe',
                                      -2,
                                      'fffffffffffffffe',
                                      1]),
                                    ('-18446744073709551617',
                                     [-2,
                                      'fffffffffffffffe',
                                      -2,
                                      'fffffffffffffffe',
                                      1]),
                                    ('-18446744073709551616',
                                     [-1,
                                      'ffffffffffffffff',
                                      -1,
                                      'ffffffffffffffff',
                                      1]),
                                    ('-9223372036854775809',
                                     [-1,
                                      'ffffffffffffffff',
                                      -1,
                                      'ffffffffffffffff',
                    1]),
                  ('-9223372036854775808',
                   [-9223372036854775808,
                    '8000000000000000',
                    -9223372036854775808,
                    '8000000000000000',
                    1]),
                  ('-9223372036854775807',
                   [-9223372036854775807,
                    '8000000000000001',
                    -9223372036854775807,
                    '8000000000000001',
                    1]),
                  ('-15',
                   [-15,
                    'fffffffffffffff1',
                    -15,
                    'fffffffffffffff1',
                    1]),
                  ('-1',
                   [-1,
                    'ffffffffffffffff',
                    -1,
                    'ffffffffffffffff',
                    1]),
                  ('0', [0, '0', 0, '0', 1]),
                  ("'NaN'", [0, '0', 0, '0', 1]),
                  ('1', [1, '1', 1, '1', 1]),
                  ('15', [15, 'f', 15, 'f', 1]),
                  ('9223372036854775806',
                   [9223372036854775806,
                    '7ffffffffffffffe',
                    9223372036854775806,
                    '7ffffffffffffffe',
                    1]),
                  ('9223372036854775807',
                   [9223372036854775807,
                    '7fffffffffffffff',
                    9223372036854775807,
                    '7fffffffffffffff',
                    1]),
                  ('NULL',
                   [9223372036854775807,
                    '7fffffffffffffff',
                    -9223372036854775808,
                    '8000000000000000',
                    1]),
                  ('9223372036854775808',
                   [0, '0', 0, '0', 1]),
                  ('18446744073709551615',
                                      [0, '0', 0, '0', 1]),
                                     ('18446744073709551616',
                                      [1, '1', 1, '1', 1]),
                                     ('36893488147419103231',
                                      [1, '1', 1, '1', 1]),
                                     ('36893488147419103232',
                                      [2, '2', 2, '2', 1]),
                                     ('99999999999999999999999999999999999999',
                                      [5421010862427522170,
                                       '4b3b4ca85a86c47a',
                                       5421010862427522170,
                                       '4b3b4ca85a86c47a',
                                       1])],
                   'time': [("'00:00:00.000000'",
                             [0, '0', 0, '0', 1]),
                            ("'00:00:00.000001'",
                             [1, '1', 1, '1', 1]),
                            ("'00:00:01.000000'",
                             [1000000,
                              'f4240',
                              1000000,
                              'f4240',
                              1]),
                            ("'00:01:00.000000'",
                             [60000000,
                              '3938700',
                              60000000,
                              '3938700',
                              1]),
                            ("'01:00:00.000000'",
                             [3600000000,
                              'd693a400',
                              3600000000,
                              'd693a400',
                              1]),
                            ("'23:59:59.999999'",
                             [86399999999,
                              '141dd75fff',
                              86399999999,
                              '141dd75fff',
                              1]),
                            ('NULL',
                             [9223372036854775807,
                              '7fffffffffffffff',
                              -9223372036854775808,
                              '8000000000000000',
                              1])],
                   'timestamp': [("'4713-01-01 00:00:00.000000 BC'",
                                  [-211810204800000000,
                                   'fd0f7fbdaf17e000',
                    -211810204800000000,
                    'fd0f7fbdaf17e000',
                    1]),
                  ("'0001-12-31 23:59:59.999999 BC'",
                   [-63082281600000001,
                    'ff1fe2ffc59c5fff',
                    -63082281600000001,
                    'ff1fe2ffc59c5fff',
                    1]),
                  ("'0001-01-01 00:00:00.000000 AD'",
                   [-63082281600000000,
                    'ff1fe2ffc59c6000',
                    -63082281600000000,
                    'ff1fe2ffc59c6000',
                    1]),
                  ("'1999-12-31 23:59:59.999999 AD'",
                   [-1,
                    'ffffffffffffffff',
                    -1,
                    'ffffffffffffffff',
                    1]),
                  ("'2000-01-01 00:00:00.000000 AD'",
                   [0, '0', 0, '0', 1]),
                  ("'2000-01-01 00:00:00.000001 AD'",
                   [1, '1', 1, '1', 1]),
                  ("'2000-01-01 00:00:01.000000 AD'",
                   [1000000,
                    'f4240',
                    1000000,
                    'f4240',
                    1]),
                  ("'2000-01-01 00:01:00.000000 AD'",
                   [60000000,
                    '3938700',
                    60000000,
                    '3938700',
                    1]),
                  ("'2000-01-01 01:00:00.000000 AD'",
                   [3600000000,
                    'd693a400',
                    3600000000,
                    'd693a400',
                    1]),
                  ("'2000-01-02 00:00:00.000000 AD'",
                   [86400000000,
                    '141dd76000',
                    86400000000,
                    '141dd76000',
                    1]),
                  ("'2000-02-01 00:00:00.000000 AD'",
                   [2678400000000,
                    '26f9d14a000',
                    2678400000000,
                    '26f9d14a000',
                    1]),
                  ("'2001-01-01 00:00:00.000000 AD'",
                   [31622400000000,
                    '1cc2a9eb4000',
                    31622400000000,
                    '1cc2a9eb4000',
                    1]),
                  ("'294276-12-31 23:59:59.999999 "
                   "AD'",
                   [9223371331199999999,
                    '7fffff5bb3b29fff',
                    9223371331199999999,
                    '7fffff5bb3b29fff',
                    1]),
                  ('NULL',
                   [9223372036854775807,
                    '7fffffffffffffff',
                    -9223372036854775808,
                    '8000000000000000',
                    1])],
    'timestamptz': [("'4713-01-01 "
                     "00:00:00.000000-1459 BC'",
                     [-211810150860000000,
                      'fd0f7fca3e2af500',
                      -211810150860000000,
                      'fd0f7fca3e2af500',
                      1]),
                    ("'1999-12-31 23:00:00.000000+0 "
                     "AD'",
                     [-3600000000,
                      'ffffffff296c5c00',
                      -3600000000,
                      'ffffffff296c5c00',
                      1]),
                    ("'2000-01-01 00:00:00.000000+0 "
                     "AD'",
                     [0, '0', 0, '0', 1]),
                    ("'2000-01-01 00:00:00.000000+1 "
                     "AD'",
                     [-3600000000,
                      'ffffffff296c5c00',
                      -3600000000,
                      'ffffffff296c5c00',
                      1]),
                    ("'294276-12-31 "
                     "23:59:59.999999+1459 AD'",
                     [9223371277259999999,
                      '7fffff4f249f8aff',
                      9223371277259999999,
                      '7fffff4f249f8aff',
                      1]),
                    ('NULL',
                     [9223372036854775807,
                      '7fffffffffffffff',
                      -9223372036854775808,
                      '8000000000000000',
                      1])],
    'timetz': [("'00:00:00.000000+0'",
                [0, '0', 0, '0', 1]),
               ("'00:00:00.000001+0'",
                [1, '1', 1, '1', 1]),
               ("'12:00:00.000000+0'",
                [43200000000,
                 'a0eebb000',
                 43200000000,
                 'a0eebb000',
                 1]),
               ("'23:59:59.999999+0'",
                [86399999999,
                 '141dd75fff',
                 86399999999,
                 '141dd75fff',
                 1]),
               ("'22:00:00.000000-4'",
                [7200000000,
                 '1ad274800',
                 7200000000,
                 '1ad274800',
                 1]),
               ("'02:00:00.000000+0'",
                [7200000000,
                 '1ad274800',
                 7200000000,
                 '1ad274800',
                 1]),
               ('NULL',
                [9223372036854775807,
                 '7fffffffffffffff',
                 -9223372036854775808,
                 '8000000000000000',
                 1]),
               ("'00:00:00.000000-1459'",
                [53940000000,
                 'c8f131500',
                 53940000000,
                 'c8f131500',
                 1]),
               ("'00:00:00.000000+1459'",
                [32460000000,
                 '78ec44b00',
                 32460000000,
                 '78ec44b00',
                 1])],
    'varchar(64)': [("'MARTINEZ'",
                     [6504691313660805453,
                      '5a454e495452414d',
                      6504691313660805453,
                      '5a454e495452414d',
                      1]),
                    ("'MURPHY'",
                     [98167120090445,
                      '59485052554d',
                      2314948375588525389,
                      '202059485052554d',
                      1]),
                    ("'abcdefg'",
                     [29104508263162465,
                      '67666564636261',
                      2334947517476856417,
                      '2067666564636261',
                      1]),
                    ("'\ttuvwxyz'",
                     [8825217399293047817,
                      '7a79787776757409',
                      8825217399293047817,
                      '7a79787776757409',
                      1]),
                    ("'tuvwxyz'",
                     [34473505465988468,
                      '7a797877767574',
                      2340316514679682420,
                      '207a797877767574',
                      1]),
                    ("'1234567 1234567'",
                     [15540725856023089,
                      '37363534333231',
                      2321383735069717041,
                      '2037363534333231',
                      1]),
                    ("'1234567a1234567'",
                     [7005127347535032881,
                      '6137363534333231',
                      7005127347535032881,
                      '6137363534333231',
                      1]),
                    ("'  '",
     [-9015182246505446685,
      '82e3a382e3a282e3',
      -9015182246505446685,
      '82e3a382e3a282e3',
      1]),
    ("'  '",
     [-9015182246505446685,
      '82e3a382e3a282e3',
      -9015182246505446685,
      '82e3a382e3a282e3',
      1]),
    ("'                '",
     [-8441510587424725032,
      '8ad9bad884d9a7d8',
      -8441510587424725032,
      '8ad9bad884d9a7d8',
      1]),
    ("'            '",
     [-8441510587424725032,
      '8ad9bad884d9a7d8',
      -8441510587424725032,
      '8ad9bad884d9a7d8',
      1]),
    ("'                '",
     [-8873914420618022440,
      '84d985d9b7d8b1d8',
      -8873914420618022440,
      '84d985d9b7d8b1d8',
      1]),
    ("'            '",
     [-6135964446350530344,
      'aad8acd884d9a8d8',
      -6135964446350530344,
      'aad8acd884d9a8d8',
      1]),
    ("'dark clouds bring'",
     [8028901226587513188,
      '6f6c63206b726164',
      8028901226587513188,
      '6f6c63206b726164',
      1]),
    ("'dark clouds bring rain'",
     [8028901226587513188,
      '6f6c63206b726164',
      8028901226587513188,
      '6f6c63206b726164',
      1]),
    ("'gnirb sduolc krad'",
     [7238164633312915047,
      '6473206272696e67',
                                                               7238164633312915047,
                                                               '6473206272696e67',
                                                               1]),
                                                              ("'niar gnirb sduolc krad'",
                                                               [7597123010476206446,
                                                               '696e67207261696e',
                                                               7597123010476206446,
                                                               '696e67207261696e',
                                                               1]),
                                                              ('NULL',
                                                               [9223372036854775807,
                                                               '7fffffffffffffff',
                                                               -9223372036854775808,
                                                               '8000000000000000',
                                                               1])]}}},
     '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 1.0.28965'}}}
```

About the Author

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

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

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

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

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

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

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

Redshift Cluster Cost Reduction Service

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

Broadly speaking, to give guidance, savings are expected fall into one of two categories; either something like 20%, or something like 80%. The former is for systems where the business use case is such that Redshift cannot be operated correctly, and this outcome requires no fundamental re-engineering work, the latter is for systems where Redshift can be operated correctly, and usually requires fundamental re-engineering work (which you may or may not wish to engage in, despite the cost savings, in which case we're back to the 20%).

Details and contact information are on the web-site.