Bytedict Encoding

Max Ganz II @ Redshift Observatory

24th October 2021

Abstract

The `bytedict` dictionary contains up to 256 entries, but the final entry is always used as an end-of-dictionary marker and so does not hold a value from the block. The dictionary holds a copy of the values which are placed in the dictionary and so its size varies according to the size of the column data type; `varchar` are stored with their full DDL length, regardless of the string length. The dictionary is stored in the block, so a large dictionary means less storage space for values. The dictionary is populated by up to the first 255 distinct values in the block (and so is affected by sorting), but is only as large as it needs to be, rather than always being the size of 256 entries. A value which is present in the dictionary is replaced by a one byte index into the dictionary. A value which is not present in the dictionary is written out in full, plus a one byte overhead. A `NULL` (as opposed to `NOT NULL`) `bytedict` column consumes an additional 1 bit per value, even for `varchar`, which normally consumes an additional 1 byte per value.

Contents

Introduction

Redshift is a column-store relational database, which means that each column in a table is stored contiguously.

It is often the case that the data in a single column has similar characteristics - for example, might be all names, or ages, or say integer values within a given range; in other words, data which is far from randomly distributed across the value range for the data type of the column.

This provides an opportunity for unusually effective data compression, as well as an opportunity to blunder terribly, as Redshift offers a range of data compression methods (known in the Redshift documentation as "encodings"), most of which work spectacularly well with and only with data which expresses the characteristics necessary for that data compression method, and which often works spectacularly badly with data lacking those characteristics.

It is then necessary to understand the type of data characteristics suitable for each of the data compression methods offered by Redshift, as well of course as the behaviours and limitations of the data compression methods, so the good choices can be made when selecting data compression for columns.

This document is one in a series, each of which examines one data compression method offered by Redshift, which here investigates the `bytedict` encoding.

Test Method

A unsorted table is created which stores all rows on a single slice, and which possesses an `int8 not null` column which uses `bytedict` encoding.

The `bytedict` dictionary is documented as holding 256 values, so this column is populated with the values 0 to 255, inclusive both ends, to fill the dictionary.

Then rows are inserted (using a self-join, with a full `vacuum` and `analyze` after every `insert`), using the value 0, which is expected to be in the dictionary, until the first block is full (which is known by a second block being formed).

We then examine how many rows are present in the first block.

This process is repeated, dropping and remaking the table every time, but with the values 254, 255 and 256, checking the size of the dictionary.

The next tests focus on `varchar`.

The first creates an unsorted table, which as before stores all rows on a single slice, which possesses a `varchar(1) not null` column, using `bytedict` encoding.

A single 1 byte string is created and inserted as many times as possible into the table, using self-join, with a full `vacuum` and `analyze` after each `insert`, until the first block is full (know as a second block is now in existence).

We then count the number of rows in the first block.

This test is then repeated, but with the DDL indicating a 65535 length `varchar`. The string used remains 1 byte in length.

The tests seem fairly minimal, but in fact knowing what was inserted and the number of rows allows us to deduce the answers to all the open `bytedict` questions.

Results

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

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

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

Test duration, excluding server bring-up and shut-down, was 65 seconds.

dc2.large, 2 nodes (1.0.32574)

Insert Value	Rows	Block Size	Bytes/Value
0	1046405	1048576	1.002075
254	1046405	1048576	1.002075
255	116495	1048576	9.001039
256	116495	1048576	9.001039

Insert Value	Rows found in Dict	Rows not found in Dict	Total Bytes for Rows	Unused Space
254	1046405	0	1046405	2171
255	255	116240	1046415	2161

Insert Value	Bytes Remaining	Dict Size	Unused Space
254	2171	2040	131
255	2161	2040	121

DDL	NOT NULL	Rows
1	NOT NULL	1048455
65535	NOT NULL	917387
1	NULL	931960

Discussion

We begin with the official docs, which can be found here.

The docs state that each block has a table, with 256 entries, each entry storing a one-byte value which is the index number of the entry and also a value from the column, where when that value is seen in the block, it is replaced by the one byte value from the table.

This doesn't make sense, and I suspect this is garbled version of what you would think actually happens, which is that the table stores 256 values, and when those values are seen in the data, they are replaced by a one byte index (implicitly known) which is their position in the table.

There's no mention of how the values are chosen, or if they can be replaced. It's not clear if the table is stored in the block or not (the docs say "a separate dictionary of unique values is created"). It is stated that if a value is found which is not in the table, it is stored in its full, uncompressed length, but there is no mention of any overheads in this case.

So the questions are;

1. How are the table values chosen?
2. How much overhead is there for each table entry?
3. How much overhead is there for a value which is outside of the table?
4. Where is the table stored?

My guess is that the values are chosen on the basis of being the first distinct 256 values in the column.

The first test is to create an unsorted table which stores all its blocks on one slice, where the table has a not null `int8` column using `bytedict`, where I first store the values 0 to 255, inclusive both ends, which should fill the dictionary, and then fill the rest of the first block with the value 0, which should be in the dictionary.

I then repeat this test, but with a few more values - 254, 255 and 256 (from knowledge obtained from prior experimentation).

The goal here is to store a value which is outside of the dictionary, which should be stored with the normal 8 bytes of an `int8`, but we will also then see what overhead there is for being outside of the dictionary.

These are the numbers we find, where the "Insert Value" is the single value being used to populate the block once the dictionary has been written.

Insert Value	Rows	Block Size	Bytes/Value
0	1046405	1048576	1.002075
254	1046405	1048576	1.002075
255	116495	1048576	9.001039
256	116495	1048576	9.001039

We observe;

1. The table is actually 255 entries, not 256.
2. When a value is in the dictionary, it is indeed replaced by a single byte.
3. When a value is not in the dictionary, the value is indeed written in full, and the overhead is an additional 1 byte.
4. The dictionary has been populated by the first 255 distinct values in the block, and these are not replaced by later values, no matter how often they appear in the block.

Knowing all this now, we can examine this data a bit more deeply.

Insert Value	Rows found in Dict	Rows not found in Dict	Total Bytes for Rows	Unused Space
254	1046405	0	1046405	2171
255	255	116240	1046415	2161

We know the size of a block (one megabyte), and we can directly compute the space taken by rows (one byte for every row in a dictionary, nine bytes for every row outside). Working out then the number of rows found in the dictionary, the number not found in the dictionary we know the total number of bytes for rows, and we find there's some unused space.

A dictionary for `int8` of 255 elements which are 8 bytes each is 2040 bytes.

It rather looks like the dictionary is in the block.

That would make sense, in fact, in general : blocks are independently compressed and from a software engineering point of view you'd want them to be wholly independent - that whenever you have a block, you have in your hand everything you need to access and use the data in that block.

A "separate dictionary", in the sense of being separate from the block, wouldn't make sense, both from that design perspective, but also from a fundamental Big Data point of view. Whenever you picked up a `bytedict` block, you would then have to do some extra work to find the dictionary - and were that to involved a *disk seek*, then death and ruin - you've just lost your capability to process Big Data in a timely manner.

If we take it then the dictionary is in the block, we then note we have a little unused space.

Insert Value	Bytes Remaining	Dict Size	Unused Space
254	2171	2040	131
255	2161	2040	121

That's quite curious. There's room for some more values to be stored, whether they are inside or outside of the dictionary. What gives?

So, all disk I/O in Redshift is in one megabyte blocks, which is to say, 1,048,576 bytes. I note in the official docs about geometry one of the limitation is that the maximum size of a `geometry` type is 1,048,447 bytes, which leaves 129 bytes unaccounted for.

Interesting, eh? almost what we see here and that has the feel of a per-block header to me.

So now we have a pretty good idea of what's going on with `bytedict`;

1. The dictionary is in the block.
2. The dictionary has 255 elements.
3. The values stored in the dictionary are the first 255 distinct values in the block.
4. When a value in the block is in the dictionary, it is replaced by a single byte.
5. When a value in the block is not in the dictionary, it is written in full, plus a one byte overhead.
6. Blocks have a little bit of unused space.

Wouldn't it have been nice if all this was just in the documentation in the first place?

By the way, as an side, given that `NULL` is implemented for almost all data types as a one bit flag for each value, couldn't not-in-dictionary have been implemented as a one bit flag also?

A few more questions come to mind;

1. Does the dictionary always consume the space required for 255 entries, or does it consume only the space required for the actual number of entries?
2. What happens with `varchar`? the dictionary seems to be index based, but `varchar` vary in length. It's not obvious how those two work together.
3. What happens with NULL? it a NULL value is taken into the dictionary, so we have a dictionary entry for NULL, or does each value in the block still carry a bit (everything but `varchar`) or a byte (`varchar`) which is used as a flag to indicate the value is NULL?

These questions can be answered with three more tests; create a table with a `varchar(1) not null`, storing a 1 byte string as many times as possible, and then see how many rows are in the first block. Repeat this, but change the DDL to 'null'. Then repeat again, still with a 1 byte string, but now with the DDL specifying a `varchar(65535) not null`.

DDL	NULL	Rows
1	NOT NULL	1048455
1	NULL	931960
65535	NOT NULL	917387

This is fascinating!

Okay, so, let's begin with the first the third rows, where all that changes is the length in the DDL - here we see, even though in both cases the actual string is 1 byte, we see when the DDL is 65535 we have 131,068 less values stored in the block.

That demonstrates - although seemingly a bit strangely, given the difference in rows in much larger than the difference in DDL length - that the dictionary entries store the full length of the DDL.

The difference in the number of rows is 131,068.

The size of a dictionary entry when the DDL is 65535 is, naturally enough, 65535; and we immediately note that the difference in the number of rows is almost exactly the size of *two* dictionary entries, which is 65535*2 = 131,070.

It looks, the two byte difference notwithstanding, awfully like *two* dictionary entries have been allocated, even though there was only one unique string.

I guess the extra entry is being used to indicate the end of the dictionary, but it seems like an improper method to do so, since it is costly with long data types.

This would explain why the documentation states the dictionary is 256 entries, when really it is 255. It's because the final entry is being used to mark end-of-dictionary.

This also means the math in the tables for the `int8` work above is slightly wrong, since it assumes there are 255 entries in the dictionary. In fact the dictionary is 2048 bytes, not 2040.

Now we can turn to the first and second rows, which examine the behaviour of `NULL`.

Normally, each value in a `varchar` column which allows `NULL` carry each a one byte overhead, used to hold a flag which is set when the value is `NULL`.

We see that with a 1 byte `NOT NULL` string, and `bytedict`, we have 1048455 values in a block.

When we change to `NULL`, we drop to 931,960 values.

It is immediately apparent that there is *not* a one byte flag per value; if there was, the size of each value would have doubled, and so the number of rows would be about 500,000 or so.

Similarly, it is also apparent that there still *is* a NULL marker on a per-value basis, because the number of values is a block has decreased; each value is now taking up a bit space.

In fact, if we look at the numbers, there are 1048455 - 931960 = 116,495 less rows in the block, where each block is one byte in length.

The number of bits in 116495 bytes is 116495 * 8 = 931,960.

We also see the number of rows in the `NULL` block is 931,960.

We conclude that the `NULL` marker for `bytedict` encoded values is 1 bit in length, regardless of the data type.

So, we can infer the following;

1. The dictionary is only as large as it need to be; it does *not* reserve space for all entries and so consume that space even when the entries are not used.
2. With `varchar`, the dictionary is allocating the DDL length of the string not the actual length. I suspect this may also mean the four byte header is being removed.
3. The dictionary holds 255 entries, and allocates one extra entry (of the same size) to indicate end-of-dictionary.
4. Columns with `bytedict` encoding when `NULL` (as opposed to `NOT NULL`) consume an additional 1 bit per value, even with `varchar`, which normally consumes 1 byte per value.

We can now also make sense of the warning given in the documentation;

> Byte-dictionary encoding is not always effective when used with VARCHAR columns. Using BYTEDICT with large VARCHAR columns might cause excessive disk usage. We strongly recommend using a different encoding, such as LZO, for VARCHAR columns.

This warning - like all in the docs - reminds me of an observation made by Richard Feynman in one of his memoirs. He said, basically, that giving people sets of rules, without explaining the principles behind those rules, doesn't work; people can follow the rules, but because they do not know why those rules exist, they are still going to do lots of things which are wrong.

(This was something which happened during the Manhattan Project. There was a plant where all the research was done, and a plant where the bombs were made. The Army for security reasons wanted the plant were the bombs were made to know *nothing* about what they were doing - which was a problem, because when you're handling radioactive materials, there are safety issues you need to know about, or maybe people die or maybe you get a big explosion. The bomb building plant had been given a whole bunch of safety rules, like "don't put too much Uranium in one pile". They were following all the rules - but they were doing things like having one pile of Uranium in one room, against the wall, and another pile in the neighbouring room, up against the other side of the same wall. They were following the rules, but getting it wrong, because they didn't understand the principles.)

However, given what we've now learned about the dictionary, we can interpret the warning : the problem is that with `varchar`, developers often specify long lengths in the DDL but then use short strings. If you do this with `bytedict`, the dictionary will be *huge*, taking up most of the space in the block, almost all of it with empty space.

Conclusions

The dictionary contains 256 entries, but only 255 are used; the final entry is an end-of-dictionary marker.

The dictionary is populated by the first 255 distinct values in the block.

The dictionary holds a copy of the values which are placed in the dictionary (and so the longer the data type, the more space the dictionary consumes).

With the `varchar` type, the dictionary stores the DDL length of the data type, not the length of the actual string. In the extreme case, with `varchar(65535)`, the dictionary holds 15 entries, each 65535 bytes in length, (the final entry, despite being full-size, being used for the end-of-dictionary marker), and is 983,025 bytes in length.

The dictionary is stored in the block.

When a value is in the dictionary, it is replaced by a one-byte value (which is the index of its position in the dictionary).

When a value is not in the dictionary, it is written out in full, with an additional one-byte marker to indicate a non-compressed value.

When a `bytedict` encoded column is `NULL` (as opposed to `NOT NULL`), an additional 1 bit is stored per value, used for a flag to indicate the value is `NULL` or not. This is true for all data types, including `varchar`, where `varchar` normally uses 1 byte for this flag.

Blocks when full (in the sense that the `insert` of one more record would form a new block) have a little unused space; for example, `int8 not null` where all values are in the dictionary has 123 bytes of unused space.

The documentation contains a warning, thus;

> Byte-dictionary encoding is not always effective when used with VARCHAR columns. Using BYTEDICT with large VARCHAR columns might cause excessive disk usage. We strongly recommend using a different encoding, such as LZO, for VARCHAR columns.

What this is actually referring to is that the full DDL length of a `varchar` is stored in the dictionary. If you make a `varchar` with a large DDL length, and then store short strings, the dictionary will be huge, but mainly composed of empty space.

The same is true for long `char` types, since they also always store the full length in the DDL, but `char` only goes up to 4096.

Further Questions

1. This leaves me now with one final question : what comes first, the dictionary or the block?

Does Redshift scan the block for the values which will form the dictionary, build the dictionary, and then populate the block? or does Redshift build the dictionary as it goes along? this can make a big difference; imagine we have a column which is a long `varchar`, where the block begins with many repeated copies of the same value, and then later has many distinct values.

If the dictionary is built first, it will be full before any encoding is done, and there will not be much space for values, and so not many of the repeated values will be stored; but if the dictionary is built as Redshift goes along, it will begin with just one entry and have lots of room for many of the repeated values, and so maybe occupy most of the block and not have room for the later distinct values.

Remembering that all I/O is in one megabyte blocks, Redshift always reads an entire block then writes the entire block, and that whenever a block is written, it's being stored permanently and so must be in a sane, readable state, I think it's going to depend in part on the way in which rows are being added to a block.

If rows are being added one by one, by say single-row `insert`, then necessarily each row which could go into the dictionary will do so, and that will occur at the time that row is presented, and all rows after that which match that row do so.

However, we can imagine a more reasonable and common scenario where a `COPY` is being executed, or a `VACUUM`, or a multi-row `INSERT`, and so we have a stream of incoming rows. Is the dictionary built up as rows arrive, or is the incoming flow of rows scanned (and buffered) until the dictionary is full? is there even an actual flow of rows? that's how I would expect the implementation to be designed, but I could be completely wrong.

So I'm not sure, and I need to test, but I need to think of a decent, simple, robust way to test this, and how it may vary in different situations.

Revision History

v1

- Initial release.

v2

- Changed speculation as to purpose of unused space in blocks from being for `VACUUM` to being a per-block header.

v3

- Fixed a typo in the Conclusion, "market" should have been "marker".

v4

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

v5

- Added "About the Author". made site name in title a link, and made each chapter start a new page.
- Updated links to amazonredshiftresearcproject.org to redshiftresearcproject.org.

v6

- 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.


```
{'proofs': {'dc2.large': {2: {'bytedict_results': {'int8': {0: 1046405,
                                                          254: 1046405,
                                                          255: 116495,
                                                          256: 116495},
                                                 'varchar': {'ddl_1_string_1': 1048455,
                                                             'ddl_1_string_1_null': 931960,
                                                             'ddl_65535_string_1': 917387}}}}},
 '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 1.0.32574'}}}
```

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.