AZ64 Encoding

Max Ganz II @ Amazon Redshift Research Project

5th March 2023

Abstract

The `AZ64` encoding is the Gorilla compression method from Facebook (2015), but with the 1-bit encoding for repeating values removed and replaced by no less than three distinct runlength-like compression methods, all of which seem completely crazy. With repeating values, when the number of repetitions is 2 to 63, no runlength encoding occurs and the full Gorilla encoding is used for each row, a lost compression opportunity; when the number of repeating values is a positive integer power of 2 between (inclusive both ends) 64 and 16384, each group of repeating values is encoded into a "storage unit" which is between 5 and 33 (inclusive both ends) bytes depending on data type which stores a copy of the value and a 15 bit counter, allowing for billions of rows per block, where I think (but have not yet investigated) materialization is block based and so needing to materialize *any* row in that block means *all* rows must be materialized; and, finally, when the number of repetitions is 65 or more, but not a positive integer power of 2, the number of rows stored per block varies depending on the value being encoded, the number of repetitions and the bit-pattern of the number of repetitions (!), as Gorilla-like encoding is being used on that bit-pattern, such that increasing the number of repetitions often *decreases* the number of rows stored in a block (as the bit-pattern of the number of repetitions has become less favourable to Gorilla encoding).

Contents

Introduction **2**

Test Method **2**

Results **3**
- dc2.large, 2 nodes (1.0.35649) 3
  - Method #1 3
  - Method #2 7
  - Method #3 9
  - Method #4 10

Discussion **13**
- Overview 15
- Encoding Method #1 : Gorilla Encoding 16
  - Gorilla Encoding in Redshift 21

Encoding Method #2 : Gorilla/Runlength Encoding with 63 or Fewer Repetitions 23
Encoding Method #3 : Runlength Encoding with Positive Integer Power of 2 Repetitions 24
Encoding Method #4 : Gorilla/Runlength Encoding with 65 or More Repetitions 26

Conclusions **28**

Revision History **30**
v1 30
v2 30

Appendix A : Raw Data Dump **31**

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 work 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 `AZ64` encoding.

Test Method

The basic test method used is to create a test table, with a single raw encoded column, such that all rows are on a single slice, and then populate the test table with rows, issuing `VACUUM` between `INSERT`s, until more than one block has been filled. This allows us to see how many rows were needed to fill one block, the first block.

I can then vary exactly what data is being inserted, to try and figure out how the encoding method is working, by seeing how changing the data affects the number of rows which fit into the first block.

So, for example, the test might be all rows with the same value, or alternating rows with two different values, or ascending values, and so on.

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 9736 seconds.

dc2.large, 2 nodes (1.0.35649)

Method #1

int2

Datum	Value
First value	0
First value (pattern)	0
Second value	1
Second value (pattern)	1
XOR	1
Stored bit pattern	1
Stored length in bits	1
Overhead in bits	0.5
Bits per Block	8388608
Estimated rows per block	5592405.33
Actual rows per block	5591809

Datum	Value
First value	51
First value (pattern)	110011
Second value	60
Second value (pattern)	111100
XOR	001111
Stored bit pattern	1111
Stored length in bits	4
Overhead in bits	0.5
Bits per Block	8388608
Estimated rows per block	1864135.11
Actual rows per block	1863937

Datum	Value
First value	-32768
First value (pattern)	1000000000000000
Second value	-1
Second value (pattern)	0000000000000001
XOR	0111111111111111
Stored bit pattern	111111111111111
Stored length in bits	15
Overhead in bits	0.5
Bits per Block	8388608
Estimated rows per block	541200.52
Actual rows per block	541121

Datum	Value
First value	-32768
First value (pattern)	1000000000000000
Second value	0
Second value (pattern)	0000000000000000
XOR	1000000000000000
Stored bit pattern	1
Stored length in bits	1
Overhead in bits	0.5
Bits per Block	8388608
Estimated rows per block	5592405.33
Actual rows per block	5591809

int4

Datum	Value
First value	0
First value (pattern)	0
Second value	1
Second value (pattern)	1
XOR	1
Stored bit pattern	1
Stored length in bits	1
Overhead in bits	1.0
Bits per Block	8388608
Estimated rows per block	4194304.00
Actual rows per block	4193856

Datum	Value
First value	51
First value (pattern)	110011

Datum	Value
Second value	60
Second value (pattern)	111100
XOR	001111
Stored bit pattern	1111
Stored length in bits	4
Overhead in bits	1.0
Bits per Block	8388608
Estimated rows per block	1677721.60
Actual rows per block	1677505

Datum	Value
First value	-2147483648
First value (pattern)	10000000000000000000000000000000
Second value	-1
Second value (pattern)	00000000000000000000000000000001
XOR	01111111111111111111111111111111
Stored bit pattern	1111111111111111111111111111111
Stored length in bits	31
Overhead in bits	1.0
Bits per Block	8388608
Estimate rows per block	262144.00
Actual rows per block	262081

Datum	Value
First value	-2147483648
First value (pattern)	10000000000000000000000000000000
Second value	0
Second value (pattern)	00000000000000000000000000000000
XOR	10000000000000000000000000000000
Stored bit pattern	1
Stored length in bits	1
Overhead in bits	1.0
Bits per Block	8388608
Estimated rows per block	4194304.00
Actual rows per block	4193856

int8

Datum	Value
First value	0
First value (pattern)	0
Second value	1
Second value (pattern)	1

Datum	Value
XOR	1
Stored bit pattern	1
Stored length in bits	1
Overhead in bits	2.0
Bits per Block	8388608
Estimated rows per block	2796202.67
Actual rows per block	2795904

Datum	Value
First value	51
First value (pattern)	110011
Second value	60
Second value (pattern)	111100
XOR	001111
Stored bit pattern	1111
Stored length in bits	4
Overhead in bits	2.0
Bits per Block	8388608
Estimated rows per block	1398101.33
Actual rows per block	1397952

Datum	Value
First value	-9223372036854775808
First value (pattern)	1000000000000000000000000000000000000000000000000000000000000000
Second value	-1
Second value (pattern)	0000000000000000000000000000000000000000000000000000000000000001
XOR	0111111111111111111111111111111111111111111111111111111111111111
Stored bit pattern	111111111111111111111111111111111111111111111111111111111111111
Stored length in bits	63
Overhead in bits	2.0
Bits per Block	8388608
Estimated rows per block	129055.51
Actual rows per block	129025

Datum	Value
First value	-9223372036854775808
First value (pattern)	1000000000000000000000000000000000000000000000000000000000000000
Second value	0
Second value (pattern)	0000000000000000000000000000000000000000000000000000000000000000
XOR	1000000000000000000000000000000000000000000000000000000000000000
Stored bit pattern	1
Stored length in bits	1
Overhead in bits	2.0
Bits per Block	8388608
Estimated rows per block	2796202.67
Actual rows per block	2795904

numeric(38,0)

Datum	Value
First value	0
First value (pattern)	0
Second value	1
Second value (pattern)	1
XOR	1
Stored bit pattern	1
Stored length in bits	1
Overhead in bits	4.0
Bits per Block	8388608
Estimated rows per block	1677721.60
Actual rows per block	1677504

Datum	Value
First value	51
First value (pattern)	110011
Second value	60
Second value (pattern)	111100
XOR	001111
Stored bit pattern	1111
Stored length in bits	4
Overhead in bits	4.0
Bits per Block	8388608
Estimated rows per block	1048576.00
Actual rows per block	1048448

Method #2

Data Type	Values	# Repetitions	Actual Rows per Block
int2	0/1	2	5591810
int2	0/1	3	5591811
int2	0/1	4	5591812
int2	0/1	62	5591842
int2	0/1	63	5591817
int2	0/1	64	13420416
int2	0/1	65	5694016
int2	65/119	2	1524994
int2	65/119	3	1524993
int2	65/119	4	1524996
int2	65/119	62	1525014
int2	65/119	63	1525041
int2	65/119	64	13420416
int2	65/119	65	1567800

Data Type	Values	# Repetitions	Actual Rows per Block
int4	0/1	2	4193856
int4	0/1	3	4193856
int4	0/1	4	4193856
int4	0/1	62	4193856
int4	0/1	63	4193856
int4	0/1	64	7455744
int4	0/1	65	4251072
int4	65/119	2	1397952
int4	65/119	3	1397952
int4	65/119	4	1397952
int4	65/119	62	1397952
int4	65/119	63	1397952
int4	65/119	64	7455744
int4	65/119	65	1433770

Data Type	Values	# Repetitions	Actual Rows per Block
int8	0/1	2	2795904
int8	0/1	3	2795904
int8	0/1	4	2795904
int8	0/1	62	2795904
int8	0/1	63	2795904
int8	0/1	64	3947136
int8	0/1	65	2821248
int8	65/119	2	1198210
int8	65/119	3	1198209
int8	65/119	4	1198212
int8	65/119	62	1198212
int8	65/119	63	1198260
int8	65/119	64	3947136
int8	65/119	65	1224470

Data Type	Values	# Repetitions	Actual Rows per Block
numeric(38,0)	0/1	2	1677504
numeric(38,0)	0/1	3	1677504
numeric(38,0)	0/1	4	1677504
numeric(38,0)	0/1	62	1677504
numeric(38,0)	0/1	63	1677504
numeric(38,0)	0/1	64	2033344
numeric(38,0)	0/1	65	1686592
numeric(38,0)	65/119	2	931968
numeric(38,0)	65/119	3	931968
numeric(38,0)	65/119	4	931968
numeric(38,0)	65/119	62	931968
numeric(38,0)	65/119	63	931968
numeric(38,0)	65/119	64	2033344

Data Type	Values	# Repetitions	Actual Rows per Block
numeric(38,0)	65/119	65	947765

Method #3

Data Type	Data Type Size	Storage Unit Size	Number Encoding Units per Block
int2	2	5	209715
int4	4	9	116508
int8	8	17	61680
numeric(38,0)	16	33	31775

(Note the negative value of -859340800 is due to the Redshift `int4` system table column recording the number of rows per block overflowing.)

Data Type	Values	# Repetitions	Storage Units per Block	Estimated Rows per Block	Actual Rows per Block
int2	0/1	64	209715	13421772	13420416
int2	0/1	128	209715	26843545	26840832
int2	0/1	256	209715	53687091	53681664
int2	0/1	512	209715	107374182	107363328
int2	0/1	16384	209715	3435973836	-859340800
int2	65/119	64	209715	13421772	13420416
int2	65/119	128	209715	26843545	26840832
int2	65/119	256	209715	53687091	53681664
int2	65/119	512	209715	107374182	107363328
int2	65/119	16384	209715	3435973836	-859340800
int4	0/1	64	116508	7456540	7455744
int4	0/1	128	116508	14913080	14911488
int4	0/1	256	116508	29826161	29822976
int4	0/1	512	116508	59652323	59645952
int4	0/1	16384	116508	1908874353	1908670464
int4	65/119	64	116508	7456540	7455744
int4	65/119	128	116508	14913080	14911488
int4	65/119	256	116508	29826161	29822976
int4	65/119	512	116508	59652323	59645952
int4	65/119	16384	116508	1908874353	1908670464
int8	0/1	64	61680	3947580	3947136
int8	0/1	128	61680	7895160	7894272
int8	0/1	256	61680	15790320	15788544
int8	0/1	512	61680	31580641	31577088
int8	0/1	16384	61680	1010580540	1010466816

Data Type	Values	# Repetitions	Storage Units per Block	Estimated Rows per Block	Actual Rows per Block
int8	65/119	64	61680	3947580	3947136
int8	65/119	128	61680	7895160	7894272
int8	65/119	256	61680	15790320	15788544
int8	65/119	512	61680	31580641	31577088
int8	65/119	16384	61680	1010580540	1010466816
numeric(38,0)	0/1	64	31775	2033601	2033344
numeric(38,0)	0/1	128	31775	4067203	4066688
numeric(38,0)	0/1	256	31775	8134407	8133376
numeric(38,0)	0/1	512	31775	16268815	16266752
numeric(38,0)	0/1	16384	31775	520602096	520536064
numeric(38,0)	65/119	64	31775	2033601	2033344
numeric(38,0)	65/119	128	31775	4067203	4066688
numeric(38,0)	65/119	256	31775	8134407	8133376
numeric(38,0)	65/119	512	31775	16268815	16266752
numeric(38,0)	65/119	16384	31775	520602096	520536064

Method #4

Data Type	Values	# Repetitions	Actual Rows per Block
int2	0/1	64 (0b1000000)	13420416
int2	0/1	65 (0b1000001)	5694016
int2	0/1	66 (0b1000010)	5796780
int2	0/1	67 (0b1000011)	5793624
int2	0/1	68 (0b1000100)	6003856
int2	0/1	69 (0b1000101)	5890624
int2	0/1	70 (0b1000110)	5991232
int2	0/1	71 (0b1000111)	5985229
int2	0/1	72 (0b1001000)	6424640
int4	0/1	64 (0b1000000)	7455744
int4	0/1	65 (0b1000001)	4251072
int4	0/1	66 (0b1000010)	4308096
int4	0/1	67 (0b1000011)	4306358
int4	0/1	68 (0b1000100)	4421440
int4	0/1	69 (0b1000101)	4359744
int4	0/1	70 (0b1000110)	4414592
int4	0/1	71 (0b1000111)	4411328
int4	0/1	72 (0b1001000)	4645512
int4	0/1	90 (0b1011010)	4854656
int4	0/1	91 (0b1011011)	4846296
int4	0/1	92 (0b1011100)	4946624
int4	0/1	93 (0b1011101)	4883008
int4	0/1	94 (0b1011110)	4927808
int4	0/1	95 (0b1011111)	4918720
int4	0/1	96 (0b1100000)	5920800
int4	0/1	97 (0b1100001)	4953499

Data Type	Values	# Repetitions	Actual Rows per Block
int4	0/1	98 (0b1100010)	4996928
int4	0/1	99 (0b1100011)	4987323
int4	0/1	100 (0b1100100)	5083500
int4	0/1	101 (0b1100101)	5020224
int4	0/1	102 (0b1100110)	5062400
int4	0/1	103 (0b1100111)	5052253
int4	0/1	104 (0b1101000)	5255016
int4	0/1	105 (0b1101001)	5083470
int4	0/1	106 (0b1101010)	5124480
int4	0/1	107 (0b1101011)	5113856
int4	0/1	108 (0b1101100)	5206144
int4	0/1	109 (0b1101101)	5143552
int4	0/1	110 (0b1101110)	5183424
int4	0/1	111 (0b1101111)	5172489
int4	0/1	112 (0b1110000)	5591824
int4	0/1	113 (0b1110001)	5200640
int4	0/1	114 (0b1110010)	5239488
int4	0/1	115 (0b1110011)	5228130
int4	0/1	116 (0b1110100)	5316800
int4	0/1	117 (0b1110101)	5254976
int4	0/1	118 (0b1110110)	5292800
int4	0/1	119 (0b1110111)	5281152
int4	0/1	120 (0b1111000)	5470320
int4	0/1	121 (0b1111001)	5306752
int4	0/1	122 (0b1111010)	5343616
int4	0/1	123 (0b1111011)	5331804
int4	0/1	124 (0b1111100)	5417088
int4	0/1	125 (0b1111101)	5356160
int4	0/1	126 (0b1111110)	5392170
int4	0/1	127 (0b1111111)	5379974
int4	0/1	128 (0b10000000)	14911488
int4	0/1	129 (0b10000001)	5464698
int4	0/1	130 (0b10000010)	5563350
int4	0/1	131 (0b10000011)	5549422
int4	0/1	132 (0b10000100)	5766592
int4	0/1	133 (0b10000101)	5634146
int4	0/1	134 (0b10000110)	5734530
int4	0/1	135 (0b10000111)	5718870
int4	0/1	136 (0b10001000)	6199696
int4	0/1	137 (0b10001001)	5803594
int4	0/1	138 (0b10001010)	5905710
int4	0/1	139 (0b10001011)	5888318
int4	0/1	140 (0b10001100)	6116096
int4	0/1	190 (0b10111110)	8131050
int4	0/1	191 (0b10111111)	8091142
int4	0/1	192 (0b11000000)	22367232
int4	0/1	193 (0b11000001)	8175866
int4	0/1	194 (0b11000010)	8302230

Data Type	Values	# Repetitions	Actual Rows per Block
int4	65/119	64 (0b1000000)	7455744
int4	65/119	65 (0b1000001)	1433770
int4	65/119	66 (0b1000010)	1470348
int4	65/119	67 (0b1000011)	1469243
int4	65/119	68 (0b1000100)	1545708
int4	65/119	69 (0b1000101)	1504200
int4	65/119	70 (0b1000110)	1541050
int4	65/119	71 (0b1000111)	1538854
int4	65/119	72 (0b1001000)	1705968
int4	65/119	90 (0b1011010)	1870848
int4	65/119	91 (0b1011011)	1863953
int4	65/119	92 (0b1011100)	1948652
int4	65/119	93 (0b1011101)	1894503
int4	65/119	94 (0b1011110)	1932452
int4	65/119	95 (0b1011111)	1924700
int4	65/119	96 (0b1100000)	3050048
int4	65/119	97 (0b1100001)	1954560
int4	65/119	98 (0b1100010)	1992732
int4	65/119	99 (0b1100011)	1984257
int4	65/119	100 (0b1100100)	2071040
int4	65/119	101 (0b1100101)	2013435
int4	65/119	102 (0b1100110)	2051648
int4	65/119	103 (0b1100111)	2042387
int4	65/119	104 (0b1101000)	2236728
int4	65/119	105 (0b1101001)	2071040
int4	65/119	106 (0b1101010)	2109312
int4	65/119	107 (0b1101011)	2099340
int4	65/119	108 (0b1101100)	2188096
int4	65/119	109 (0b1101101)	2127360
int4	65/119	110 (0b1101110)	2165900
int4	65/119	111 (0b1101111)	2155176
int4	65/119	112 (0b1110000)	2609488
int4	65/119	113 (0b1110001)	2182595
int4	65/119	114 (0b1110010)	2221120
int4	65/119	115 (0b1110011)	2209840
int4	65/119	116 (0b1110100)	2300164
int4	65/119	117 (0b1110101)	2236689
int4	65/119	118 (0b1110110)	2275276
int4	65/119	119 (0b1110111)	2263380
int4	65/119	120 (0b1111000)	2466960
int4	65/119	121 (0b1111001)	2289683
int4	65/119	122 (0b1111010)	2328370
int4	65/119	123 (0b1111011)	2315844
int4	65/119	124 (0b1111100)	2407584
int4	65/119	125 (0b1111101)	2341625
int4	65/119	126 (0b1111110)	2380288
int4	65/119	127 (0b1111111)	2367280
int4	65/119	128 (0b10000000)	14911488

Data Type	Values	# Repetitions	Actual Rows per Block
int4	65/119	129 (0b10000001)	2404560
int4	65/119	130 (0b10000010)	2455872
int4	65/119	131 (0b10000011)	2441840
int4	65/119	132 (0b10000100)	2562912
int4	65/119	133 (0b10000101)	2479120
int4	65/119	134 (0b10000110)	2531456
int4	65/119	135 (0b10000111)	2516400
int4	65/119	136 (0b10001000)	2795888
int4	65/119	137 (0b10001001)	2553680
int4	65/119	138 (0b10001010)	2606976
int4	65/119	139 (0b10001011)	2590960
int4	65/119	140 (0b10001100)	2718240
int4	65/119	190 (0b10111110)	3589312
int4	65/119	191 (0b10111111)	3560240
int4	65/119	192 (0b11000000)	22367232
int4	65/119	193 (0b11000001)	3597520
int4	65/119	194 (0b11000010)	3664896

Discussion

There are two major points to begin with.

The first major point is that Amazon have published very nearly no information about the `AZ64` encoder.

This is problematic.

Usually, a given encoder works staggeringly well when and only when used with data which is just right for that encoder - data which expresses characteristics which are appropriate for the way in which the encoder works.

Similarly, and critically, when a given encoder is used with data which is not appropriate for the encoder, often you find the results are as staggeringly *bad* as they would be good if the data had been just right.

So, for example, a runlength encoder, which stores a value and then a count of how many times that value then occurs, works staggeringly well when the values being encoded are repetitions of the same value; and staggeringly badly when this is not so.

In all cases, to select the appropriate encoder for the data in hand, it is absolutely necessary to know how the encoders work - and so what data is just right for them - and to know what data you have in hand. There's no getting around this.

An encoder which has no documentation describing how it works, or at least the characteristics of data it works well with, *is absolutely and totally useless in every single possible way*, and cannot be used under any circumstances whatsoever, and this is the case with `AZ64`.

As far as I can tell, the *only* information about `AZ64` is found in an AWS a blog post.

(There's also the official doc page, but it's completely devoid of content and as such not worth even linking to, because there's absolutely no information in it.)

That blog post compares `AZ64` to `LZO` and `ZSTD`, and makes claims for `AZ64` compressing much more, and much more quickly.

As we will see, however, `AZ64` is a fundamentally different type of compression method to `LZO` and `ZSTD`, and as such the data it works well, and works badly with, is fundamentally different to the data for `LZO` and `ZSTD` (which although different methods, are fundamentally similar).

This difference means comparing `AZ64` to `LZO` and `ZSTD` is absolutely incorrect.

It's akin to comparing runlength encoding to delta encoding; you simply *cannot* compare them direct, without explanation, because they're utterly different to each other and are used in totally different situations and with totally different data.

Additionally, I have unpublished benchmarks which utterly contradict the performance claims made; I find `AZ64` about the same, or very slightly faster, than all the other encoders; the claims of 40% or 70% faster are, as far as I've found, completely and utterly incorrect.

I regard that blog post as one of the very worst ever published by Amazon.

I suspect the unidentified chap who wrote the blog post to be the usual chap who does the docs; and although I could be wrong, I think from all I've seen over the years that the way it works is that someone gives him some information, which he doesn't really understand, he writes it up, and no one technical ever reviews what is written.

The second major point is that I have spent a *lot* of time trying to figure out how `AZ64` works over the last four months, but I have been only partially successful.

I believe I have discerned the core encoding methods in use, and there are four of them, but the problem is, although I may be wrong, as far as I can tell `AZ64` looks to be half nonsense.

It appears to use one encoding method from Facebook, the Gorilla method, which is excellent and makes sense, and three encoding methods, all of which are runlength-like, which are baroquely complex, even bizarre, while at the same time always being far less efficient, more complex, and much harder to reason about than simplest and obvious runlength method of one copy of the datum and a count of repetitions.

`AZ64` to me *make no sense*, and this makes it problematic to figure out how it operates; it's easy to understand a method which makes optimal design choices because it makes sense but hard to understand a method which does not, because it could be doing anything for any reason at all.

Having said this then, let us turn to `AZ64`.

Overview

Essentially, I think `AZ64` consists of the Gorilla encoder, published by Facebook in 2015, with the runlength encoding behaviour removed, plus three different runlength encoding methods, which I think are hand-rolled by Amazon.

`AZ64` selects one of its four encoding methods depending on the situation, but there is in this a lot of strange behaviour, which I can only really describe, as I cannot explain in it terms which make sense to me.

The four encoding methods are explained in detail, but to begin with, an overview of the methods and when they are used, to give a general feel for what's going on;

1. Gorilla Encoding

   This can be considered the default method; it is used when there are no repetition of values, which is to say, when we have rows where the value differs from the value in the previous row.

2. Gorilla/Runlength Encoding with 63 or Fewer Repetitions

   Gorilla encoding specifies behaviour to handle repetition of values, but that behaviour has not been implemented.

   Instead, when there is is repetition of values, but when the number of repetitions is 63 or less, each value is encoded as if it were a single value with no repetition (and so a great deal of compression is lost, as Gorilla encodes repeated values using a single bit).

   The normal, simple and obvious method for encoding repeated values is to have a single copy of the value, and then a count of the number of repetitions. I have no idea why it was not used.

3. Runlength Encoding with Positive Integer Power of 2 Repetitions

   This method is used when there is repetition of values, and the number of repetitions is positive integer power of 2, e.g. 64, 128, 256, etc.

   (This encoding method handles a maximum of 16384 repetitions. When there are more repetitions than this, they are encoded as blocks of 16384 repetitions, with the final block probably not a power of 2, and so handled by one of the other encoding methods.)

   With this method, the value being encoded makes no difference to the number of rows in a block (unlike the other three methods).

   This method fits the most rows into a block, and in fact can fit *billions* of rows into a single block, which is extremely dangerous, as Redshift normally can materialize rows only on a per-block basis, and so any query which causes any rows in such a block to need to be materialized causes *all* the rows in the block to be materialized.

   It has been possible to develop math to exactly predict the behaviour of this encoding method, and so to be able to exactly predict (as is then proved with the test suite) the maximum possible number of rows per block for the different data types.

As before, the normal, simple and obvious method for encoding repeated values is to have a single copy of the value, and then a count of the number of repetitions. I have no idea why it was not used.

4. Gorilla/Runlength Encoding with 65 or More Repetitions

   This method is used when there is repetition of values, but when the number of repetitions is 65 or more *and* the number of repetitions is not a positive integer power of 2 (as when this is the case, method #3 is used).

   With this method, both the value being encoded makes a difference to the number of rows in a block (as is normal with Gorilla encoding), but also the *bit pattern* of the number of repetitions makes a difference, *and* the actual number of repetitions makes a difference (the bit pattern makes a difference in the number of leading and trailing zeros, and so affects encoding in a way which is different to its actual, numeric value), and as such increasing the number of repetitions often *decreases* the number of rows encoded per block, because the bit pattern becomes less favourable.

   This is crazy. It is difficult to reason about, and it's worse in every way than the normal, simple and obvious method for encoding repeated values, by having a single copy of the value, and then a count of the number of repetitions. I have no idea why it was not used.

   It looks like this encoding method behaves for the first 64 repetitions in the same way as method #2 (so there is no extra compression beyond Gorilla encoding with its repetition handling removed), but after that, as the number of extra repetitions increases (and depending on the bit pattern of the number of repetitions), there is an increasing gain in the number of rows per block.

Encoding Method #1 : Gorilla Encoding

So, I will begin with the simplest and the sensible part of `AZ64`.

As far as I can tell, when the data values have *no* repetition (so each value differs from the previous value), `AZ64` encodes using the Gorilla encoding method from Facebook, published in 2015, PDF available here.

(Note Gorilla encoding specifically supports highly efficient compression where there is repetition, but as I will explain it looks to me this functionality has been removed from this implementation.)

At this point I need to explain how Gorilla encoding works.

The Gorilla encoding method is based around the use of the bit-wise operator `XOR`.

Here's the `XOR` logic chart.

A	B	XOR
0	0	0
0	1	1
1	0	1

A	B	XOR
1	1	0

When we come to load data into Redshift, we typically have an incoming set of rows, either from `COPY` or `INSERT`, and a table we're going to load the rows into, and we then load the incoming rows into the table.

To make life simple, let us here imagine the table has only a single column.

So we have a set of rows, each with one column, where each row then holds a value in that column, and these rows and their values are loaded into the table.

With Gorilla encoding, however, we store is the first value in full (in the first row), but after that, we store `XOR` results, rather than the actual original values presented by `COPY` or `INSERT`.

So we take the first value and the second value, compute the `XOR` result, and store the result in the second row; then we take the second value and the third value, compute the `XOR` result and store the result in the third row, and so on and so on.

Apart from the first value, which is a special case, we are *not* storing the actual, original values - they are thrown away once they've been used to compute their `XOR` result.

So we have our incoming rows and their values, from the `COPY` or `INSERT` command, we iterate over them, using them to compute `XOR` results, and we store the `XOR` results, throwing away the actual original values from `COPY` or `INSERT`.

Now, a key property of `XOR` is that if you have one of the two values used to produce a `XOR` result, and you have the result itself, you can from that value and the result compute the *other* of the two values which generated the `XOR` result.

To render this algebraically;

```
a XOR b      = result
a XOR result = b
b XOR result = a
```

Now, if we think back to what we've stored in the column in our table, we have the first value in full, and then a series of `XOR` results.

We can in fact fully recompute all the original values from the `COPY` or `INSERT` command.

We begin with the first row in the table; this stored in full the first original value, so we directly have the first original value.

We then load the value from the second row, which as we recall is the `XOR` result of the value from the first row with the value in the second row.

So now we have one of the original two values (the value from the first row), and the `XOR` result, and so we can directly compute (by performing an `XOR` of the original value and the result) the *other* of the two values used to compute the stored `XOR` result (the original value from the second row).

So we can imagine our original incoming values were `5, 8, 10, 19, 6`, then we stored the `5` in full in the first row, and stored the `XOR` result of `5 and 8` in the second row; and so now we take the `5` from the first row, the `XOR` result from the second row, and that lets us compute that the original value in the second row was `8`.

Since we know now the full original value in the second row, we can in exactly the same way take that value, and the `XOR` result in the *third* row, to compute the original value in the *third* row (the `10`).

We continue doing this, row after row, to fully compute all the original values.

Now, this is all good and well but so far pointless, because the length in bits of an `XOR` result is the same as the length in bits as the values being `XOR`ed, so we're not saving any space and the whole point of this was data compression, after all.

However, we now can describe some special data processing which can be performed, which leads to data compression.

Let's begin by looking at one or two example `XOR` calculations.

First, here's an `XOR` for two values which have lots of different bits in their bit-patterns;

datum	value
A	1011 1011 1010 0000 1001
B	0000 1110 1110 1000 0000
XOR	1011 1010 0101 0111 0110

Second, here's an `XOR` for two values which have lots of identical bits in their bit-patterns;

datum	value
A	0011 1101 1100 0000 1000
B	0011 1101 0101 0000 1000
XOR	0000 0000 1001 0000 0000

Notice the difference in the results? When two bits in `A` and `B` are the same, *you get a 0 bit in the result*, so the more similar the two bit-patterns are, the more 0 bits you get in the result.

It's at this point we have a lovely opportunity for some cunning data compression.

So, first, remember, we're storing entire `XOR` results. If we could find a way somehow to need to store only *partial* results, we'd save a lot of disk space.

Next, remember how we're decompressing the rows; we start with the first row, which stores `A` in full, and then we use that along with the `XOR` result stored in the second row, to compute the original value for the second row; and we then use the original value from the second row along with the `XOR` result in the third row, to compute the original value for third row, and so on…

So we normally have the first value of the two values used to make the `XOR` result, and the result itself (and this allows us to compute the second value, which is the original value of the row we're looking at).

Now, remember - when in the result a bit is 0, then the bits in `A` and `B` were the same.

We always have `A` (the original value of the current row), so when in the result we have a 0 bit, we already *know* what the bit must be in `B` - it's the same as the matching bit in `A`.

So in fact, in theory, if we could find a way *not* to need to store the 0 bits in the result - but still know where they were in the result - we would save a ton of disk space; we could store a partial result, but still have all the information we need to recompute `B`.

Now go back and have another look at the second `XOR` result, with lots of 0 bits.

Notice the large number of contiguous leading 0 bits and contiguous trailing 0 bits?

Rather than storing them all, why not store a *count* of the the number of leading 0 bits and the count of the number of trailing 0 bits? there are eight leading and eight trailing bits, so we're using eight bits, but we could for example use just four bits to store the actual number eight. We also store in full the bits in-between the leading and trailing 0 bits, because there's no obvious reasonable way given a bunch of mixed up 1 bits and 0 bits to avoid storing the 0 bits but still be able to figure out where they are.

So, what we actually store is something like the number of leading 0 bits, the actual bits in-between the leading and trailing 0 bits, and the length of the bits in-between. We do not need to store the number of trailing 0 bits, because we know the length of the data type of the column - of the three lengths, leading 0 bits, bits in-between, trailing 0 bits - we only need to store any two, because we know the length of the data type.

In fact, Gorilla does some fancy footwork (which I'll come to) to store the necessary length information, much more so than just storing a count of the number of contiguous 0 bits.

Now, what we do make of this, as a data compression method?

If we look at the first `XOR` example, we see that we save almost nothing at all - there is only a single contiguous trailing 0 bit; so we save only a single bit. The worst case then obviously is when both the very first and very last bits both differ, and then Gorilla saves nothing - indeed, even if all the bits in-between were 0 bits, it wouldn't help us, because Gorilla works by compression contiguous leading and trailing 0 bits; and the longer the data type with this happening, the worse it is, because the the result is the same length as the length of the data type.

The best case is when two numbers are identical. Then every bit in the result is a 0 bit, and in fact in this case Gorilla encoding just stores a single 0 bit and that’s it.

However, we have to remember it’s the *bit-pattern* which matters, so our base-10 intuition is not going to serve us well - for example, numbers which are integer powers of 2 are going to be very different to their neighbouring numbers;

```
4294967296 = 100000000000000000000000000000000
4294967295 = 011111111111111111111111111111111
```

Only 1 difference, but the bit-patterns are almost entirely unlike each other.

However, we then see this;

```
4294967296 = 100000000000000000000000000000000
2147483648 = 010000000000000000000000000000000
```

And now we find our two numbers, which are about two billion apart, are almost identical for Gorilla encoding; we have no leading 0 bits, two bits of “in-between” bits, and then 31 bits of trailing 0 bits, which we save.

You can reason about Gorilla compression, but you have to *think* about it. It operates in the domain of base 2, not base 10, and the more similar numbers are, the better they compress - but similarity means contiguous leading and trailing 0 bits in the `XOR` result.

When we have small numbers in large data types, Gorilla works very well, because we end up with really long contiguous leading 0 bits. Imagine you have a 64 bit data type, with two numbers being `XOR`ed, and both only use say the least significant four or five bits each; you’re going to save about 60 bits in the `XOR` result, from the huge number of leading contiguous 0 bits.

What this really means normally of course is that the data type is wrong; it’s much too long for the values. However, there are going to be situations where although normally the values are small, there is occasionally going to be a very large value, and so the large data type is needed; and here Gorilla does very well - although so do the `mostly` encoders.

I mentioned earlier that Gorilla encoding does some fancy footwork to encode length information. This is what we find in the PDF;

1. When the value is identical to the previous value, store a 0 bit and that’s it; no lengths are stored, no body, we’re done.
2. When re-using previously stored length information (explained below), store a 1 bit (so we can tell the difference between this and what’s stored for identical values), and then a 0 bit (to indicate re-use), and then the XOR body (see notes below about this not making sense).
3. When writing length information, store a 1 bit (non-identical), then a 1 bit (to indicate storing lengths) the leading zero length (in 5 bits), the body length (in 6 bits) and then the XOR body.

Re-use of previously stored length information is something which Gorilla chooses to do, as an optimization. When storing a value, if the length of both the leading zeros and the body are *less* than the most recently stored lengths (so the lengths stored for the most recent value which actually did have lengths written out), just write a 1 bit, which indicates the re-use of the most recently stored lengths, and then write the body.

Note that this is what to me the white paper seems to say, but it doesn't quite make sense to me - I think you'd need to pad the XOR body with leading and trailing 0 bits to match the lengths being used, and so you'd only do perform re-use if the padding was less than the 11 bits needed to store the lengths.

Gorilla Encoding in Redshift

Now, it's not exactly clear to me how Redshift is encoding Gorilla length information. I know for sure (as will be described in the next section) that the standard Gorilla encoding of identical values is not in use; Redshift *has* deviated from the standard Gorilla spec.

Empirically, by issuing test cases, I can demonstrate that the number of bits needed to store the current row is the number of bits of contiguous difference between the bit-patterns of the current row and the previous row, plus an overhead, where the overhead is;

Data Type Length	Overhead in Bits per Row
2	0.5
4	1
8	2
16	4

Experimentation shows this formula is accurate. It is impossible to be exactly accurate, because in Redshift each block has a small header, something like 120 bytes, but that header length seems to vary a bit, so it's not possible to exactly know how much of a block is available to store user data, and you need to know that exactly, to exactly figure out how many rows will be stored in a block.

Let's look at some examples of alternating values being stored and the number of rows we estimate will be found in one block, and the number of rows we actually find. The "Values" column has two values, and these alternate through the entire block, the "# Bits Stored" is per row, as is the "Overhead", and "Estimated Rows per Block" is computed by dividing the number of bits in one block by the number of bits needed for one row. Finally, we then have the "Actual Rows per Block", as found by the test script.

(The negative numbers in "Values" unfortunately do not render very well.)

Data Type	Values	# Bits Stored	Overhead	Expected Rows per Block	Actual Rows per Block	
int2	0/1	1	0.5	1.5	5,592,405	5,591,809
int4	0/1	1	1	2	4,194,304	4,193,856
int8	0/1	1	2	3	2,796,202	2,795,904
int2	51/60	4	0.5	4.5	1,864,135	1,863,937
int4	-2147483648/-1	32	1	33	262,144	262,881

Data Type	Values	# Bits Stored	Overhead	Expected Rows per Block	Actual Rows per Block	
int4	-2147483648/0	1	1	2	4,194,304	4,193,856

Particularly given the large number of difference in rows for the last two examples, which differ only by storing a -1 or a 0, it's hard to imagine Gorilla encoding is *not* in use, because we would need to dream up some other compression method which has exactly the same extremely unusual and highly distinctive characteristics and behaviour. It is only reasonably possible to explain these findings by imaging `XOR` based compression, which compresses contiguous leading and trailing 0 bits in the `XOR` result, and that is Gorilla encoding.

We can look more closely at the final two findings.

First, `int4`, storing values -2147483648 and -1, with one repetition of each (e.g. -2147483648/-1/-2147483648/-1/-2147483648/-1/etc).

```
-2147483648 = 10000000000000000000000000000000
-1          = 11111111111111111111111111111111
```

There are 31 bits of contiguous difference between each pair of values, and a 1 bit overhead per value for `int4`, so we expect each value to use 32 bits. There are 1024*1024*8 = 8,388,608 bits in one block. 8,388,608 / 32 = 262,144 rows per block. The actual number of rows found, by experimentation, is 262,881.

Second, `int4`, storing values -2147483648 and 0, with one repetition of each (e.g. -2147483648/0/-2147483648/0/-2147483648/0/etc).

```
-2147483648 = 10000000000000000000000000000000
0           = 00000000000000000000000000000000
```

There is 1 bit of contiguous difference between each pair of values, and a 1 bit overhead per value, so we expect each value to use 2 bits. There are 1024*1024*8 = 8,388,608 bits in one block. 8,388,608 / 2 = 4,194,304 rows per block. The actual number of rows found, by experimentation, is 4,193,856.

So, in summary, we can then make some hard and fast rules about the use of `AZ64` when it is using Gorilla encoding;

1. *Never* use `AZ64` (with Gorilla encoding) on any columns in interleaved tables. The values in all columns in interleaved tables are effectively randomly ordered.
2. *Never* use `AZ64` (with Gorilla encoding) on any columns in unsorted tables.
3. With compound sorted tables, *only* use `AZ64` (with Gorilla encoding) on columns which are part of the sortkey. *Never* use `AZ64` (with Gorilla encoding) on columns which are outside of the sortkey.
4. *Never* use `AZ64` with data which has consecutive rows with the same value, as the 1-bit runlength encoding behaviour in Gorilla has been removed

and replaced by what appear to be three different, all crazy, runlength methods.

In short, use `AZ64` when and only when the value is a row is always different to the previous row, and when the `XOR` result for each row has as many contiguous leading and trailing 0 bits as is possible.

Turning to the results, taking one of them to explain it, this is the result for `int2` with values -32768/-1.

Datum	Value
First value	-32768
First value (pattern)	1000000000000000
Second value	-1
Second value (pattern)	0000000000000001
XOR	0111111111111111
Stored bit pattern	111111111111111
Stored length in bits	15
Overhead in bits	0.5
Bits per Block	8388608
Estimated rows per block	541200.52
Actual rows per block	541121

We see the first and second values and their bit-patterns, then the `XOR` result of those bit-patterns which is almost all 1 bits. Gorilla strips away the contiguous leading and trailing 0 bit, which here saves us only the single leading 0 bit, and so we then see the bits we'll actually store, and that we store 15 bits.

Next is the unexplained but observed overhead per row, which for `int2` is 0.5 bits. Then we have a simple constant, the number of bits in one block, and as one block is one megabyte, that is 8388608 bits.

We can now estimate the number of rows per block - each row will consume 15.5 bits, and 8388608 divided by 15.5 gives 541200.52; finally we have the actual number of rows found by the test script.

The slight difference is I think due to the per-block header.

Encoding Method #2 : Gorilla/Runlength Encoding with 63 or Fewer Repetitions

Now, Gorilla encoding as defined in its PDF normally handles repeated values extremely well. The first value is stored, and after that every repetition of that value is stored using a single bit. It is expected then that repetitions will lead to excellent compression.

This is however not what is found.

Bizarrely, when storing alternating blocks of values (such as 0/1, or 65/99), when each of the values is repeated, up to 63 times, the number of values stored in a block very nearly does not change - a difference of a few rows, out of millions, as we change the number of repetitions by 1; in other words, runlength compression is not occurring.

This is to me incomprehensible. It's simply a lost compression opportunity; there is no gain by it, and no reason for not compressing.

So here we have results from the test script, for the values 0/1 and 65/119.

Each row of the table is the result of a single test, where of the two values in a value pair, each is repeated a give number of times, as specified in the "# Repetitions" column. So for the first row in the table, the data in the table consists of 0, repeated twice, followed by 1, repeated 2, with this pattern continuing until the first block is full. We then see how many rows are in the first block.

Data Type	Values	# Repetitions	Actual Rows per Block
int2	0/1	2	5591810
int2	0/1	3	5591811
int2	0/1	4	5591812
int2	0/1	62	5591842
int2	0/1	63	5591817
int2	0/1	64	13420416
int2	0/1	65	5694016
int2	65/119	2	1524994
int2	65/119	3	1524993
int2	65/119	4	1524996
int2	65/119	62	1525014
int2	65/119	63	1525041
int2	65/119	64	13420416
int2	65/119	65	1567800

As we can see, the number of values stored changes as the values change, but the number of rows store does not change (not meaningfully - just by a very few rows, probably something to do with how much overhead is consumed in the first block) as the number of repetitions occurs (with the exception of 64 repetitions, because then method #2 kicks in, as 64 is an integer power of 2).

If Gorilla encoding has been used as defined in its white paper, with a single bit for every repetition of a value, it would perform far better than `AZ64` is doing here, where `AZ64` is re-using the full Gorilla encoded bit-pattern for the repeated value for every row.

The upshot of this in terms of using `AZ64` is that with 63 or fewer repetitions, there is no compression improvement over single repetitions.

Encoding Method #3 : Runlength Encoding with Positive Integer Power of 2 Repetitions

The next encoding mode presented by `AZ64` comes into use when the number of repetitions is a positive integer power of 2, between 64 and 16384 (inclusive both ends).

For these particular number of repetitions, a “storage unit” is written to disk, the size of which is the data type size in bytes times two, plus one byte, where the storage unit contains the full value being stored (and as such the value being stored no longer matters, as it did before with encoding methods #1 and #2), and the storage unit records the number of repetitions of the value (which must be a positive integer power of 2, between 64 and 16384 inclusive both ends - if it is not, this encoding method is not used).

This encoding method can store very large numbers of rows per block, and as such in fact determines the maximum possible rows per block for `AZ64`.

We can directly compute the maximum number of values per block (which is when we have 16384 repetitions per storage unit) and these values have been found, empirically, in the test suite, to be *exactly correct*.

Data Type	Data Type Size	Storage Unit Size	Number Storage Units	Max Rows per Block
int2	2	5	209694	3,435,626,496
int4	4	9	116496	1,908,670,464
int8	8	17	61674	1,010,466,816
numeric(38)	16	33	31771	520,536,064

I am astounded by this behaviour.

1. I may be wrong, I’ve not checked materialization behaviour in Redshift for some time, but I think it is dangerous. It allows an extremely large number of rows in a single block, and Redshift used to and I believe still does basically materialize rows on a per-block basis (so if you need one row in a block, you get all of them), so that if Redshift attempts to materialize such as block, then the cluster may well grind to a halt - a couple of billion rows usually brings most systems to their knees. Such blocks then are booby-traps, and you need to make sure when using `AZ64` that your data is such that no such blocks end up being encoded.
2. The simplest runlength encoder design would store the entire value being encoded, followed by a count of the number of repetitions. The maximum number of repetitions allowed here is 16384, which is 15 bits. If we think then about say `int8` we’re talking 64 bits for the value and 15 bits for the count, which is 79 bits. With this encoding method though, `int8` uses 136 bits.

   The smallest storage unit seen is for `int2`, at 40 bits. A runlength encoder would need 16 bits for the `int2` and 15 bits for the length encoding, a total of 39 bits, so even in this case, method #3 is inferior.

   As a method for storing runlength information, method #3 makes absolutely no sense.
3. As an encoding method, despite being more complex and less effective than the simplest runlength encoder, it nevertheless compresses profoundly

more effectively than either of the other runlength-like compression methods used in `AZ64` (method #2 and #4). Apart from the problem of massively large row counts (which could be handled with some code to limit the maximum number of rows), the power-of-2 runlength encoding method is fantastically more efficient than methods #2 and #4; why are the other methods in existence at all, particular method #4 with it baroque design which depends on the value being encoded, the number of repetitions *and* the bit-pattern of the number of repetitions? and why is this method in existence but limited to positive integer powers of 2 repetitions only? why would you have all three methods in the first place, and then use the best one only very rarely? and why would you have *any* of them, when the simplest runlength encoder is better than all of them? exactly which drugs were the dev team snorting when they did this work?

Encoding Method #4 : Gorilla/Runlength Encoding with 65 or More Repetitions

The next encoding mode presented by `AZ64` comes into use when the number of repetitions is 65 or greater, and is *not* a positive integer power of 2, between 64 and 16384 inclusive both ends.

I think the first 64 repetitions receive no extra compression at all, just as with 63 or fewer repetitions, but that the additional repetitions over 64 obtain some form of runlength compression.

With this method, both the value being encoded, and the number of repetitions, affect compression, but, surprisingly, of the number of repetitions, both the number itself *but also its bit-pattern* affect compression, such that increasing the number of repetitions often *reduces* compression because the bit-pattern has become less favourable, and the magnitude of effect of the bit-pattern can be extremely large - many times greater than the number of repetitions itself.

We can examine the actual results found from the test script, for `int4`.

Note the number of repetitions tested is not contiguous. This is to reduce the time taken to run the test script.

Data Type	Values	# Repetitions	Actual Rows per Block
int4	0/1	64 (0b1000000)	7455744
int4	0/1	65 (0b1000001)	4251072
int4	0/1	66 (0b1000010)	4308096
int4	0/1	67 (0b1000011)	4306358
int4	0/1	68 (0b1000100)	4421440
int4	0/1	69 (0b1000101)	4359744
int4	0/1	70 (0b1000110)	4414592
int4	0/1	71 (0b1000111)	4411328
int4	0/1	72 (0b1001000)	4645512
int4	0/1	90 (0b1011010)	4854656
int4	0/1	91 (0b1011011)	4846296
int4	0/1	92 (0b1011100)	4946624
int4	0/1	93 (0b1011101)	4883008

Data Type	Values	# Repetitions	Actual Rows per Block
int4	0/1	94 (0b1011110)	4927808
int4	0/1	95 (0b1011111)	4918720
int4	0/1	96 (0b1100000)	5920800
int4	65/119	64 (0b1000000)	7455744
int4	65/119	65 (0b1000001)	1433770
int4	65/119	66 (0b1000010)	1470348
int4	65/119	67 (0b1000011)	1469243
int4	65/119	68 (0b1000100)	1545708
int4	65/119	69 (0b1000101)	1504200
int4	65/119	70 (0b1000110)	1541050
int4	65/119	71 (0b1000111)	1538854
int4	65/119	72 (0b1001000)	1705968
int4	65/119	90 (0b1011010)	1870848
int4	65/119	91 (0b1011011)	1863953
int4	65/119	92 (0b1011100)	1948652
int4	65/119	93 (0b1011101)	1894503
int4	65/119	94 (0b1011110)	1932452
int4	65/119	95 (0b1011111)	1924700
int4	65/119	96 (0b1100000)	3050048
int4	65/119	97 (0b1100001)	1954560

64 repetitions invokes method #2, but after that we're looking at the behaviour of method #4.

What we see is that both the values being stored, the number of repetitions, and the *bit pattern of the number of repetitions*, affect the number of rows per block.

Something Gorilla-like is happening with the number of repetitions.

This can be seen clearly by looking at 96 repetitions, where we then have a large number of trailing 0 bits; the number of rows stored per block suddenly shoots up to 3,050,048, from the 1,924,700 of 95 repetitions.

This is though different to method #2, because with the powers of 2 (which is method #2), the number of rows per block does *not* change when the values being encoded change, whereas with the non-powers of 2, such as 96, the number of rows stored *does* change depending on the values being encoded.

This number of repetitions is usually less important than the bit-pattern of the number of repetitions can be much more important than the actual number of repetitions. For values 0/1 we see the following;

Data Type	Values	# Repetitions	Actual Rows per Block
int4	0/1	96 (0b1100000)	5920800
int4	0/1	135 (0b10000111)	5718870
int4	0/1	136 (0b10001000)	6199696

For example, 96 repetitions gives 5,920,800 rows per block, a value we do not manage to exceed until we get to 136 repetitions, with 6,199,696 rows per block.

Needless to say, it is not expected that increasing the number of repetitions *reduces* compression.

Of all the encoding methods in `AZ64`, this is the method where I've least pinned down what's going on, and this is due to a complete lack of motivation, because what's going on looks to be crazy, and so there is no value in figuring it out. What should have been implemented is the usual runlength method of one value, plus a count of repetitions, plus, should you want to limit the number of repetitions so blocks do not have too many rows, an arbitrary maximum number of repetitions.

Conclusions

The `AZ64` encoder appears to implement four different compression methods, each of which is used in different circumstances.

1. When a value differs to the value in the previous row, the Gorilla compression method from Facebook (2015) is used. Gorilla encoding is based on storing for each row the result of an `XOR` of a value with the value in the previous row, but removing from the `XOR` result all contiguous leading and trailing 0 bits; which is to say, the more there are identical contiguous leading and trailing bits, in the two values, the better.

   Note however the Gorilla specification encodes repeated values in 1 bit, but that this behaviour has been removed from the `AZ64` implementation and replaced with three different runlength encoding methods, described below.

2. When a value repeats, from 2 to 63 times, the full Gorilla encoding for the value is used in each row (rather than storing 1 bit). This is a lost compression opportunity.

3. When a value repeats, and the number of repetitions is a positive integer power of 2, from 64 to 16384 inclusive both ends, a "storage unit" (from 5 to 33 bytes inclusive both ends in length, depending on the data type) is used to encode a copy of the value and a 15 bit counter of the number of repetitions.

   This allows billions of rows per block, which I think highly dangerous when materializing rows (I've yet to investigate this, but I currently think rows are materialized on a per block basis, so needing to materialize any row in a block means all its rows are materialized), and in fact also overflows the signed 32 bit `num_values` column in 'STV_BLOCKLIST', which becomes negative.

4. When a value repeats, more than 64 times but the number of repetitions is not a positive integer power of 2, the number of values per block is based on the value being stored, the number of repetitions, and the bit pattern of the number of repetitions, where Gorilla-like encoding is occurring on

the value as well as the number of repetitions, such that increasing the number of repetitions often *decreases* the number of rows stored per block.

I have not pinned down exactly how this method works, because it seems crazy, and I simply can find no motivation within myself to figure it out when there's so many more pressing investigations to work on.

When it comes to using `AZ64`, observe the following rules;

1. Never use `AZ64` with interleaved tables, as the values in rows are basically random, which is the worst use case for this compression method.
2. Never use `AZ64` with unsorted tables, for the same reason (unless you control your data load so it is in fact ordered, even though the table is not).
3. Never use `AZ64` with unsorted columns (outside of the sortkey) in compound sorted tables.
4. Never use `AZ64` with data which repeats, as `runlength` encoding is better.

Note that Redshift's default encoding selection behaviour ignores all these rules.

If you have been creating tables using `CREATE TABLE AS`, or using `CREATE TABLE` but without specifying encodings, or using `COPY` to load data with `COMPUPDATE` in used in a non-sampling mode (I've not checked with sampling), you will find `AZ64` is used for all data types it supports (with the exception of columns in sortkeys, which are are `raw`). Recently AWS also introduced an option to have Redshift select column encodings automatically on an ongoing basis, changing encodings as it sees fit; I absolutely would not use this functionality until it has been investigated and found to be safe.

In short, `AZ64` performs when well the `XOR` of the value in the current row and the previous row has as many contiguous leading and trailing 0 bits as possible. That's not quite intuitive to work with, because for example with `int2`, -32678 and 0 works almost perfectly (1 bit stored per value), but -32678 and -1 works appallingly (15 bits stored per value). You need to understand how the compression method works to use it correctly; there's never any getting away from this, not for any compression method.

I strongly recommend `AZ64` is not used when any of its runlength encoding methods would come into play.

The only information I have ever seen from AWS about `AZ64` is in this blog post, here.

As we can now directly see, comparing `AZ64` to `ZSTD` and `LZO` is wrong, as they are different types of compression method. `AZ64` works extremely well on values which when `XOR`ed with the value in the previous row have many contiguous leading and trailing 0 bits in the `XOR` result, but increasingly poorly as we move from this to data which is more and more random, where-as `ZSTD` and `LZO` are general purpose compression methods which work reasonably well on all data.

For `AZ64` to have done so much better, as described in the blog post, the data must have been carefully chosen to be highly suitable for `AZ64` and reporting only this test scenario, and so presenting it as the normal scenario, is unethical.

Furthermore, I have as-yet unpublished benchmarks showing `AZ64` to be at best a couple of percentage points faster than `LZO` or `ZSTD`.

I assert that the information AWS have published, to the extent I think I now understand `AZ64`, has no credibility.

There is, as far as I can see, no connection at all between the blog post and the reality of the encoding method.

I call upon AWS to publish their benchmark method and the data used.

Revision History

v1

- Initial release.

v2

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

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: {'method_1': {'int2': {-32768: {-1: {1: 541121},
                                                              0: {1: 5591809}},
                                                    0: {1: {1: 5591809}},
                                                    51: {60: {1: 1863937}}},
                                           'int4': {-2147483648: {-1: {1: 262081},
                                                                   0: {1: 4193856}},
                                                    0: {1: {1: 4193856}},
                                                    51: {60: {1: 1677505}}},
                                           'int8': {-9223372036854775808: {-1: {1: 129025},
                                                                            0: {1: 2795904}},
                                                    0: {1: {1: 2795904}},
                                                    51: {60: {1: 1397952}}},
                                           'numeric(38,0)': {0: {1: {1: 1677504}},
                                                             51: {60: {1: 1048448}}}},
                              'method_2': {'int2': {0: {1: {2: 5591810,
                                                            3: 5591811,
                                                            4: 5591812,
                                                            62: 5591842,
                                                            63: 5591817,
                                                            64: 13420416,
                                                            65: 5694016}},
                                                    65: {119: {2: 1524994,
                                                               3: 1524993,
                                                               4: 1524996,
                                                               62: 1525014,
                                                               63: 1525041,
                                                               64: 13420416,
                                                               65: 1567800}}},
                                           'int4': {0: {1: {2: 4193856,
                                                            3: 4193856,
                                                            4: 4193856,
                                                            62: 4193856,
                                                            63: 4193856,
                                                            64: 7455744,
                                                            65: 4251072}},
                                                    65: {119: {2: 1397952,
                                                               3: 1397952,
                                                               4: 1397952,
                                                               62: 1397952,
                                                               63: 1397952,
                                                               64: 7455744,
                                                               65: 1433770}}},
                                           'int8': {0: {1: {2: 2795904,
                                                            3: 2795904,
                                                            4: 2795904,
                                                            62: 2795904,
                                                            63: 2795904,
                                                            64: 3947136,
                                                            65: 2821248}},
                      65: {119: {2: 1198210,
                                 3: 1198209,
                                 4: 1198212,
                                 62: 1198212,
                                 63: 1198260,
                                 64: 3947136,
                                 65: 1224470}}},
             'numeric(38,0)': {0: {1: {2: 1677504,
                                       3: 1677504,
                                       4: 1677504,
                                       62: 1677504,
                                       63: 1677504,
                                       64: 2033344,
                                       65: 1686592}},
                               65: {119: {2: 931968,
                                          3: 931968,
                                          4: 931968,
                                          62: 931968,
                                          63: 931968,
                                          64: 2033344,
                                          65: 947765}}}},
'method_3': {'int2': {0: {1: {64: 13420416,
                              128: 26840832,
                              256: 53681664,
                              512: 107363328,
                              16384: -859340800}},
                      65: {119: {64: 13420416,
                                 128: 26840832,
                                 256: 53681664,
                                 512: 107363328,
                                 16384: -859340800}}},
             'int4': {0: {1: {64: 7455744,
                              128: 14911488,
                              256: 29822976,
                              512: 59645952,
                              16384: 1908670464}},
                      65: {119: {64: 7455744,
                                 128: 14911488,
                                 256: 29822976,
                                 512: 59645952,
                                 16384: 1908670464}}},
             'int8': {0: {1: {64: 3947136,
                              128: 7894272,
                              256: 15788544,
                              512: 31577088,
                              16384: 1010466816}},
                      65: {119: {64: 3947136,
                                 128: 7894272,
                                 256: 15788544,
                                 512: 31577088,
                                 16384: 1010466816}}},
             'numeric(38,0)': {0: {1: {64: 2033344,
                                       128: 4066688,
                                       256: 8133376,
                                            512: 16266752,
                                            16384: 520536064}},
                                    65: {119: {64: 2033344,
                                               128: 4066688,
                                               256: 8133376,
                                               512: 16266752,
                                               16384: 520536064}}}},
 'method_4': {'int2': {0: {1: {64: 13420416,
                               65: 5694016,
                               66: 5796780,
                               67: 5793624,
                               68: 6003856,
                               69: 5890624,
                               70: 5991232,
                               71: 5985229,
                               72: 6424640}}},
              'int4': {0: {1: {64: 7455744,
                               65: 4251072,
                               66: 4308096,
                               67: 4306358,
                               68: 4421440,
                               69: 4359744,
                               70: 4414592,
                               71: 4411328,
                               72: 4645512,
                               90: 4854656,
                               91: 4846296,
                               92: 4946624,
                               93: 4883008,
                               94: 4927808,
                               95: 4918720,
                               96: 5920800,
                               97: 4953499,
                               98: 4996928,
                               99: 4987323,
                               100: 5083500,
                               101: 5020224,
                               102: 5062400,
                               103: 5052253,
                               104: 5255016,
                               105: 5083470,
                               106: 5124480,
                               107: 5113856,
                               108: 5206144,
                               109: 5143552,
                               110: 5183424,
                               111: 5172489,
                               112: 5591824,
                               113: 5200640,
                               114: 5239488,
                               115: 5228130,
                               116: 5316800,
                               117: 5254976,
                               118: 5292800,
                                                    119: 5281152,
                                                    120: 5470320,
                                                    121: 5306752,
                                                    122: 5343616,
                                                    123: 5331804,
                                                    124: 5417088,
                                                    125: 5356160,
                                                    126: 5392170,
                                                    127: 5379974,
                                                    128: 14911488,
                                                    129: 5464698,
                                                    130: 5563350,
                                                    131: 5549422,
                                                    132: 5766592,
                                                    133: 5634146,
                                                    134: 5734530,
                                                    135: 5718870,
                                                    136: 6199696,
                                                    137: 5803594,
                                                    138: 5905710,
                                                    139: 5888318,
                                                    140: 6116096,
                                                    190: 8131050,
                                                    191: 8091142,
                                                    192: 22367232,
                                                    193: 8175866,
                                                    194: 8302230}},
                                            65: {119: {64: 7455744,
                                                       65: 1433770,
                                                       66: 1470348,
                                                       67: 1469243,
                                                       68: 1545708,
                                                       69: 1504200,
                                                       70: 1541050,
                                                       71: 1538854,
                                                       72: 1705968,
                                                       90: 1870848,
                                                       91: 1863953,
                                                       92: 1948652,
                                                       93: 1894503,
                                                       94: 1932452,
                                                       95: 1924700,
                                                       96: 3050048,
                                                       97: 1954560,
                                                       98: 1992732,
                                                       99: 1984257,
                                                       100: 2071040,
                                                       101: 2013435,
                                                       102: 2051648,
                                                       103: 2042387,
                                                       104: 2236728,
                                                       105: 2071040,
                                                       106: 2109312,
                                                       107: 2099340,
                                                           108: 2188096,
                                                           109: 2127360,
                                                           110: 2165900,
                                                           111: 2155176,
                                                           112: 2609488,
                                                           113: 2182595,
                                                           114: 2221120,
                                                           115: 2209840,
                                                           116: 2300164,
                                                           117: 2236689,
                                                           118: 2275276,
                                                           119: 2263380,
                                                           120: 2466960,
                                                           121: 2289683,
                                                           122: 2328370,
                                                           123: 2315844,
                                                           124: 2407584,
                                                           125: 2341625,
                                                           126: 2380288,
                                                           127: 2367280,
                                                           128: 14911488,
                                                           129: 2404560,
                                                           130: 2455872,
                                                           131: 2441840,
                                                           132: 2562912,
                                                           133: 2479120,
                                                           134: 2531456,
                                                           135: 2516400,
                                                           136: 2795888,
                                                           137: 2553680,
                                                           138: 2606976,
                                                           139: 2590960,
                                                           140: 2718240,
                                                           190: 3589312,
                                                           191: 3560240,
                                                           192: 22367232,
                                                           193: 3597520,
                                                           194: 3664896}}}}}}},
 '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 1.0.35649'}}}
```