SA128 - A Smart Data Compression Technique for Columnar Databases Based on Characteristics of Data

by

Sukhpreet Singh Anand

A Thesis Presented in Partial Fulfillment
of the Requirements for the Degree
Master of Science

Approved April 2021 by the
Graduate Supervisory Committee:

Ajay Bansal, Chair
Robert Heinrichs
Javier Gonzalez-Sanchez

ARIZONA STATE UNIVERSITY

May 2021

ABSTRACT

Traditionally, databases have been categorized as either row-oriented or column-oriented databases. Row-oriented databases store each row of the table's data contiguously onto the disk whereas column-oriented databases store each column's data contiguously onto the disk. In recent years, columnar database management systems are becoming increasingly popular because deep and narrow queries are faster on them. Hence, column-oriented databases are highly optimized to be used with analytical (OLAP) workloads (Mike Freedman 2019). That is why they are very frequently used in business intelligence (BI), data warehouses, etc., which involve working with large data sets, intensive queries and aggregated computing. As the size of data keeps growing, efficient compression of data becomes an important consideration for these databases to optimize storage as well as improve query performance. Since column-oriented databases store data of the same data type contiguously, most modern compression techniques provide better compression ratios as compared to row-oriented databases.

This thesis introduces a new compression technique called SA128 for column-oriented databases that performs a column-wise compression of database tables. SA128 is a multi-stage compression technique which performs a column-wise compression followed by a table-wide compression of database tables. In the first stage, SA128 performs an analysis based on the characteristics of data (such as data type and distribution) and determines which combination of lossless-compression algorithms would result in the best compression ratio. In the second phase, SA128 uses an entropy encoding technique such as rANS (Duda, J., 2013) to further improve the compression ratio.

To my parents for believing in me and providing their unconditional support.

To my would-be wife for motivating me when I needed it the most.

ACKNOWLEDGMENTS

Several people have played an important role in the timely completion of this thesis and the related research associated with it. I would like to acknowledge them here.

I would like to thank Dr. Ajay Bansal, my thesis chair and advisor, for providing me with the encouragement to pursue this research. The completion of this research would have not been possible without his participation and suggestions during our weekly meetings.

I would like to thank Dr. Robert Heinrichs for his continuous involvement in my research work, providing valuable suggestions and pushing me to achieve thesis milestones on time.

I would like to thank Dr. Javier Gonzalez-Sanchez for supporting my research and serving as part of my thesis committee.

I would also like to thank my family, friends and people close to me for inspiring and motivating me along the way. This work would have not been possible without their support.

TABLE OF CONTENTS

Page

CHAPTER Page

APPENDIX Page

LIST OF TABLES

Table Page

LIST OF FIGURES

Figure Page

Figure Page

Figure Page

Chapter 1

INTRODUCTION

In traditional databases, as data is stored and queried at scale, data compression becomes significantly important both in terms of optimizing storage space as well as improving query performance. More compression means that lesser number of disk blocks need to be read into memory during each query operation. This reduction in disk I/O leads to an increase in query performance.

Traditional databases can be divided into row-oriented databases or column-oriented databases. The difference between the two is mostly concerned with how the data is stored onto disk. In row-oriented databases, the data tables are stored row by row, whereas in column-oriented databases the data tables are stored column by column. In a column-based database, each column value is stored contiguously onto disk. This storage format has several advantages during compression since the contiguous data values stored share similar properties such data type, range of values, etc. and sometimes even share some common characteristics as in the case of time-series data. Generic compression techniques in row-oriented databases fail to take into account this local property and common characteristics of data and thus compression on column-oriented databases can lead to better compression ratios.

A common problem with many compression algorithms is that they are data set specific. An algorithm which works well on one data set, may not work well on others. A common example of this is Delta encoding and Run-length encoding, Frame of Reference (Goldstein et al. 1998), etc. In Delta encoding, data sets which have a small difference (delta) between each contiguous value compress better than those with large delta values. In Run-length encoding, if our data set has a lot of repeated values stored contiguously, then this encoding works very well as opposed to datasets with lesser

number of contiguous repeated values. The same is true for Frame of Reference (Goldstein et al. 1998) technique. If the data set has all values that are large, then FOR compresses well, but if the data set contains several zero values, then it is not as good.

Another problem with most compression algorithms is that they are datatype specific. For example, Delta Encoding and Run-length Encoding only works well for Integers, XOR-based encoding works well for only floating-point numbers, LZ77 works well for string types, etc.

The goal of our thesis is to introduce a new lossless compression technique called SA128 for column-oriented databases which comprises of two stages. This current scope of this thesis mainly focuses on the space savings achieved using SA128 compression and not on improving the compression/decompression performance. Compression and decompression times can be significantly improved using parallel processing, computing using SIMD instructions and testing results on powerful server machines (for more details, refer to Chapter 8).

In the first stage, we analyse the data based on their data type. Based on the properties and distribution of data belonging to each data type, we use a smart selection process to select a combination of one or more state of the art compression algorithms that result in a good compression ratio for that dataset instead of blindly choosing a generic compression algorithm for the data set. In this stage, we divide each data column into one of 5 categories based on its data type. We use a combination of the following algorithms based on the below categories of data types to compress them (the data types considered below are some of the commonly used data types in PostgreSQL):

1. **Integers (integer/int4, smallint/int2, bigint/int8), date, time, timetz, timestamp and timestamptz:** For data columns belonging to this category, we divide the data column into blocks of 128 data values each. Data belonging to data types such as date, time, timetz, timestamp and timestamptz are all converted

to Integers so that all of them can be compressed using similar methods. For each block and based on the distribution of the values, we perform Delta encoding and Delta of Delta encoding over a Frame of reference on two separate copies of the original block. By this stage, we have three copies of the same block where one copy is unencoded and the other two copies are encoded using Delta encoding and Delta of Delta encoding over a Frame of reference respectively. In the next step, we use a zigzag encoding technique on data values belonging to all three copies of the block to map negative numbers to positive numbers. By the end of this stage, if there are repeated runs of contiguous values in the block, we use the Run-length encoding technique on all three copies of the block. The value for each of the runs for the data value is stored within the block itself and the index location to that run value is stored in an exception block for each copy of the block to remember the position of the run values. In the next step, we use bit packing technique where we encode all the values with the minimum number of bits required to encode the largest value in each block. To ensure that the largest value is not a large value, we use an exception technique in combination with bit packing. For this, we calculate the minimum number of bits (b_{min}) required to store a majority of the values in the block and the bits (b_{max}) required to store the largest element in the block. We select the optimum number of bits iteratively going from b_{max} to $b_{min.}$ At each iteration, all values requiring more than b bits of storage are broken by dividing them with the largest value which can be stored using b bits. The quotient and remainder pair (q, r) are stored as two separate values in place of the original value to ensure all values in the copy of the block can be represented using b bits. We use a second exception block for each copy of the block to store the index positions of the numbers which were broken down into a (q, r) pair. We use a third exception block for each copy of the block to store the index positions of all the NULL values in the block copy. In the end,

we compute the total size occupied by each of the three copies of the block along with the sizes of their respective exception blocks and choose the copy requiring the least amount of space as the final block. Information regarding which degree of Delta encoding was used to encode the data block is stored in the block header which can be used by the decoder using decompression.

2. **Decimal/Numeric:** Numeric types are fixed precision datatypes. They are often represented with a tuple (p, s) where p and s are the precision and scale of the data values respectively. For these data types, we follow a similar approach as the previous category. However, after dividing the data column into blocks of 128 values each and making three copies of each block, we further split each copy of the block into two sub-blocks where the first sub-block contains the part of the data value appearing before the decimal point and the second sub-block contains the part of the data value appearing after the decimal point. For each of the two sub-blocks, we follow the same compression process as the first category such as computing variants of Delta encoding over a Frame of reference, zigzag encoding, run-length encoding, bit packing with modulo technique and packing NULL values.
3. **Floats (real/float4, double precision/float8):** In the first step, we again divide the data column into blocks of 128 values each. In the second step, we use an approach similar to that used for numeric datatypes in second category. However, instead of using three copies of the block (along with the two sub-blocks for each) for computing variants of delta encoding, we maintain another fourth copy of the block. On this copy, we perform XOR-based encoding (Pelkonen et al. 2015) between the contiguous values in the block and we do so over a frame of reference. We perform zigzag encoding, run-length encoding, bit packing with exception technique and encode NULL values for all four copies of the block and select the block (and its exception blocks) with the smallest size as our final

block. Information regarding which encoding was used to encode the data (uncompressed, delta encoding, delta of delta encoding or XOR-based encoding) is stored in the block header which can be used by the decoder using decompression.

4. **Boolean:** For Boolean values, have only 3 distinct values, i.e., true, false and NULL. Therefore, the probability of having long repeated runs of the same value is very high in these columns. We again divide the column values into blocks of 128 values each, however this time we keep only a single copy of each block. For each block instead of trying variants of Delta encoding on each block to see which one compresses better, we directly use run-length encoding followed by bit packing and NULL handling to encode the block since delta encoding would not be much effective in a block containing only 0, 1 and NULL values.
5. **Character (char) and Character Varying (varchar):** For this category, we use a dictionary encoding mechanism and map each String to a unique integer. We then use their integer representation and encode it using the encoding procedure for the first category integer as discussed above. We then use LZ77 compression (Ziv J., Lempel A. 1977) to encode the dictionary strings and store them along with the integer blocks. This only works well when there are a high number of repeated string values in the data set. This means that the resulting dictionary will contain a smaller number of unique strings making compression more effective. If there are lesser number of repeated values, then we compress the entire column using LZ77 compression (Ziv J., Lempel A. 1977) since no other compression algorithm would lead to a decent compression on it.

In the second stage, we use a ranged variant of Asymmetric Numeral Systems (ANS) entropy encoding technique called rANS (Duda, J., 2013) to further compress the column compressed during the previous stage. With each insert, update and delete

operation on the database in the form of queries, the probability distributions of each symbol might frequently change over time. rANS (Duda, J., 2013) belong to the family of ANS algorithms and is highly suitable for environments with fast-changing symbol probability distributions such as databases with fast compression performance. The probability distribution of the compressed data is stored along with the compressed data so that it can be used by the decoder during decompression.

Structure of document: This thesis comprises of the following sections:

- Chapter 2 introduces the preparation of data sets used in taking the results of our compression technique and the preparation of this document.
- Chapter 3 discusses the background and history of progress in the area of data compression. The history behind some of the techniques used in our research have been discussed in detail.
- Chapter 4 discusses the design, structure and details of each component of our compression algorithm and techniques used with it.
- Chapter 5 discusses the design and logic behind our decompression algorithm encoded using the compression algorithm discussed in chapter 4.
- Chapter 6 describes the experimental setup, presents the results and performs a comparative analysis on storage and compression ratio in comparison with other state of the art compression algorithms.
- Chapter 7 continues our discussion on the results, lessons learned over the course of the project, the limitations and talk about the further improvements.
- Chapter 8 discusses the limitations of our approach and assumptions.
- Chapter 9 concludes the thesis, with ideas about future work.

Chapter 2

DATA SET PREPARATION

For verifying the results of our SA128 compression technique, we performed compression and decompression on two types of datasets:

1. A 1 GB TPC-DS benchmark data set (Transaction Processing Performance Council 2020) for data belonging to the first, second and fifth data type categories, i.e., for integers (smalling/int2, integer/int4, bigint/int8), identifier, date, timestamp, timestamptz, time, timez, decimal/numeric, char and varchar.

2. A generated dataset containing 4 different types of tables for data belonging to the third and fourth data type categories, i.e., for Boolean, real/float4 and double precision/float8.

This section provides a brief overview about the various aspects of both the above datasets and their purpose in accurately computing compression results for SA128.

2.1 Benchmark dataset

For our benchmark dataset, we will be using a 1GB TPC-DS dataset for calculating the compression ratio and space savings on data modelled from actual production data.

According to the TPC-DS documentation version 2.13.0 (Transaction Processing Performance Council 2020), the "TPC Benchmark™ DS (TPC-DS) is a decision support benchmark that models several generally applicable aspects of a decision support system, including queries and data maintenance. The benchmark provides a representative evaluation of the System Under Test's (SUT) performance as a general-purpose decision support system."

The TPC-DS benchmark (Transaction Processing Performance Council 2020) illustrates decision support systems which examine large volumes of data and provides relevant and objective performance data to industry users, they are expected to be accurate representations of system performance. Hence, this benchmark is good candidate for measuring the performance of SA128 against them.

TPC-DS (Transaction Processing Performance Council 2020) models the decision support functions of a retail product supplier, i.e., the sales and sales return process for an organization that employs three primary sales channels: stores, catalogs, and the Internet. The supporting schema contains vital business information, such as customer, order, and product data. The goal of selecting a retail business model is to assist the reader in relating intuitively to the components of the benchmark, without tracking that industry segment so tightly as to minimize the relevance of the benchmark.

2.1.1 TPC-DS Benchmark Schema Overview

The TPC-DS benchmark dataset (Transaction Processing Performance Council 2020) comprises of a total of 24 tables which include 7 fact tables and 17-dimension tables where:

1. 6 tables comprise of a pair of fact tables focused on the product sales and returns for each of the three channels.
2. 1 fact table that models the inventory for the catalog and internet sales channels.
3. 17-dimension tables that are associated with all sales channels.

2.1.2 Column Definition

For each of the above 24 tables, we create a definition file (which is a .dat file), where the naming convention of each file is in the form <table name>_def.dat. every column in a table, we have a row in the definition file corresponding to it, which stores its column definition. The column definition is in the following format:

Column name	Datatype	NULLs	Primary Key	Foreign Key

Table 2.1: Column Definition Format

Each of these five properties are explained below in detail:

Column name: The TPC-DS documentation (Transaction Processing Performance Council 2020) states that each column is uniquely named, and each column name begins with the abbreviation of the table in which it appears. Columns that are part of a business key are indicated with (B) appearing after the column name. A business key is neither a primary key nor a foreign key in the context of the data warehouse schema. It is only used to differentiate new data from update data of the source tables during the data maintenance operations.

Datatype: Each column employs one of the following datatypes:

a. **Identifier:** The column shall be able to hold any key value generated for that column.

b. **Integer:** The column shall be able to exactly represent integer values (i.e., values in increments of 1) in the range of at least (-2^{n-1}) to $(2^{n-1} - 1)$, where n is 64.

c. **Decimal (d, f):** The column shall be able to represent decimal values up to and including d digits, of which f shall occur to the right of the decimal place; the values can be either represented exactly or interpreted to be in this range.

d. **Char (N):** The column shall be able to hold any string of characters of a fixed length of N. If the string that a column of datatype char(N) holds is shorter than N characters, then trailing spaces shall be stored in the database or the database shall automatically pad with spaces upon retrieval such that a char_length() function will return N.

e. **Varchar (N):** The column shall be able to hold any string of characters of a variable length with a maximum length of N. Columns defined as "varchar(N)" may optionally be implemented as "char(N)".

f. **Date:** The column shall be able to express any calendar day between January 1, 1900 and December 31, 2199.

The datatypes do not correspond to any specific SQL-standard datatype. The definitions are provided to highlight the properties that are required for a particular column. The benchmark implementer may employ any internal representation or SQL datatype that meets those requirements. The implementation chosen by the test sponsor for a particular datatype definition shall be applied consistently to all the instances of that datatype definition in the schema, except for identifier columns, whose datatype may be selected to satisfy database scaling requirements.

NULLs: If a column definition includes an 'N' in the NULLs column, this column is populated in every row of the table for all scale factors. If the field is blank this column may contain NULLs.

Primary Keys: Columns that are part of the table's primary key are indicated in the column called Primary Key. If a table uses a composite primary key, then for convenience of reading the order of a given column in a table's primary key is listed in parentheses following the column name.

Foreign Keys: If the values in this column join with another column, the foreign columns name is listed in the Foreign Key field of the column definition.

Note: For our experiments and results, we will only be concerned with the first three properties, i.e., column name, datatype and NULLs for each column. The primary key and foreign key are not useful to us in our compression technique. Also, since the datatypes used in this benchmark dataset cover our first, second and fifth PostgreSQL datatype categories, we will only use this benchmark to test compression performance on these three categories only. Datatypes belonging to the third and fourth category will be tested using a different dataset.

2.1.3 Dataset scale

The TPC-DS benchmark (Transaction Processing Performance Council 2020) defines a set of discrete scaling points ("scale factors") based on the approximate size of the raw data produced by dsdgen.

The set of scale factors defined for TPC-DS is 1 TB, 3 TB, 10 TB, 30 TB and 100 TB for regular databases and 1 GB for a qualification database.

For our performance measurements for compression, we will be using a 1 GB TPC-DS dataset and calculate compression ratio for columns belonging to all 7 fact tables and 17-dimension tables.

2.2 Generated dataset

Our generated dataset consists of a large dataset (1 GB) each consisting of 4 tables each. Each table comprises of 3 columns each where each column belongs to a datatype from the set {Boolean, Real/Float4, Double Precision/Float8}, i.e., datatypes belonging to the third and fourth categories.

Details of the 4 generated tables are as follows:

1. **Table where all columns have non-decreasing values:** For the column with 'boolean' datatype, we have the first half of the values as all 'False' followed by the second half as all 'True'. For the columns with 'real' and 'double precision' datatypes, our values start from 0.0 and go all the way till 700000.0 with an increment of 0.1, where the 'real' values have up to 6 significant digits and 'double precision' values have up to 15 significant digits respectively.

2. **Table where all columns have non-increasing values:** For the column with 'boolean' datatype, we have the first half of the values as all 'True' followed by the second half as all 'False'. For the columns with 'real' and 'double precision' datatypes, our values start from 700000.0 and go all the way till 0.0 with a decrement of 0.1, where the 'real' values have up to 6 significant digits and 'double precision' values have up to 15 significant digits respectively.

3. **Table where all columns have random values over a small range:** For the column with 'boolean' datatype, we have randomly generated 'True' or 'False' values. For the columns with 'real' and 'double precision' datatypes, we generate random values which range between a small bound ranging from a value x to (x + 1) where x is chosen randomly. The 'real' values have up to 6 significant digits and 'double precision' values have up to 15 significant digits respectively.

4. **Table where all columns have random values over a large range:** For the column with 'boolean' datatype, we have randomly generated 'True' or 'False' values. For the columns with 'real' and 'double precision' datatypes, we generate random values which range between a large bound ranging from a value x to (x + 1000) where x is chosen randomly. The 'real' values have up to 6 significant digits and 'double precision' values have up to 15 significant digits respectively.

Chapter 3

PREVIOUS WORK AND RESEARCH QUESTIONS

Data compression (History of Lossless Data Compression Algorithms 2019) is a process by which the size occupied by a file or a piece of data is encoded in a way such that it uses fewer bits of storage to represent compared to the original file or data. Throughout history, there have been a tremendous amount if research in the area of data compression. Most of the compression algorithms today fall into either of the two categories – lossy compression and lossless compression.

In lossy compression, small and unimportant details are removed from a file or piece of data so that it required less amount of storage. This kind of compression is irreversible, which means that it is impossible for the decompression algorithm to restore the original file or data back again. Lossy compression algorithms are generally able to achieve very high compression ratios. Lossless compression algorithms on the other hand, compresses data in such a way that the decompression algorithm is always able to recover the original data back so that there is no loss of data. Lossless compression algorithms generally do not achieve compression ratios as high as lossy compression algorithms, but are extremely important in cases where our data is very important and loss to data cannot be tolerated. Most lossless compression algorithms rely on the principle that the data being compressed has large amounts of redundancy and non-random values. Hence, they can then be condensed using statistical modelling techniques.

3.1 Data Compression Techniques

In this section, we will discuss some of the common compression techniques used throughout history which are relevant to our thesis.

3.1.1 Delta Encoding

Delta Encoding reduces the amount of information required to represent a data object by only storing the difference (or delta) between the object and one or more reference objects (Joshua et al. 2020). This reduces the variance (range) of the values when the difference between contiguous values is small. Using this encoding, we can use fewer bits to represent the data point by only storing the delta from the previous data point. The following tables 3.1 and 3.2 show contents of our database tables before and after delta encoding where each value in a column is subtracted by the previous value in the column:

Time	CPU	Bytes	Temperature	Humidity
2021-03-25 20:00:00	140	6,843,472,947	28	60
2021-03-25 20:05:00	150	434,455,352	30	60
2021-03-25 20:10:00	150	434,231,335	30	60
2021-03-25 20:15:00	160	3,185,285,098	31	60

Table 3.1: Table Data Before Delta Encoding

Time	CPU	Bytes	Temperature	Humidity
2021-03-25 20:00:00	140	6,843,472,947	28	60
5 seconds	10	-6,409,017,595	2	0
5 seconds	0	-224,017	0	0
5 seconds	10	2,751,053,763	1	0

Table 3.2: Table Data after Delta Encoding

3.1.2 Delta of Delta Encoding

A large amount of data which is used today comprises of time-series data which is a series of data points collected over time intervals making it possible to track changes over time (Joshua et al. 2020). For time-series data, another variation of Delta encoding called Delta of Delta encoding is even more efficient in further reducing the data size. For example, if we consider the 'Time' column in Table 3.1 and 3.2, we can see that the time values are logged every 5 seconds. Therefore, instead of using Delta encoding, we compute a Delta of Delta encoding where we again compute Delta encoding for a second time over the Delta encoded table. If we apply Delta of Delta encoding to data in table 3.2, we get the below table 3.3:

Time	CPU	Bytes	Temperature	Humidity
2021-03-25 20:00:00	140	6,843,472,947	28	60
5 seconds	10	-6,409,017,595	2	0
0	-10	6,408,793,578	-2	0
0	10	-3,657,739,815	1	0

Table 3.3: Table Data after Delta of Delta Encoding

From the above table 3.3, we can see that after the first two rows, all of the following rows for the 'Time' column contains '0' values, which take an even lesser number of bits to represent than when the column was compressed using Delta encoding.

3.1.3 Run-length Encoding

In Run-length encoding (Joshua et al. 2020), if we have more than two number of repeats of the same value appearing contiguously within the data being compressed, we store one copy of the data value along with the number of repeats as a pair {run; value}.

For example, the below data can be compressed using run-length encoding in the following way:

Original Data: 11, 11, 12, 12, 12, 12, 12, 1, 1, 12, 12, 12, 12

Data after Run-length Encoding: {2, 11}, {5, 12}, {2, 1}, {4, 12}

The Run-length encoded data above required only 11 digits of storage:

Digits used: ([2, 1, 1, 5, 1, 2, 2, 1, 4, 1, 2])

For the original data, approximately 24 digits is required by an optimal series of variable length integers:

Digits used: ([1, 1, 1, 1, 1, 2, 1, 2, 1, 2, 1, 2, 1, 2, 1, 1, 1, 2, 1, 2, 1, 2, 1, 2])

Run-length encoding is highly useful for image data compression where several contiguous pixels share the same color, etc.

3.1.4 Zigzag Encoding

Zigzag encoding (Fürstenau 2015) is a technique which maps negative values to positive values while going back and forth (0 = 0, -1 = 1, 1 = 2, -2 = 3, 2 = 4, -3 = 5, 3 = 6, …). This technique is useful because it is hard to represent negative numbers using a small number of bits because of the sign bit in MSB position in their binary representation. For small numbers, although the magnitude of the numbers may be small, they require a large number of bits to represent making compression on negative numbers not efficient. Given below are the implementations of how the zigzag encoder and decoder work:

Encoding: (n >> bitlength - 1) ^ (n << 1)

Decoding: (n >>> 1) ^ - (n & 2)

Here, n is the number being encoded or decoded, the bitlength can be 32 for a 32-bit JAVA integer, '>>' is the arithmetic shifting operation (highest order bit is replicated), '^' is the XOR-operation, '>>>' is the non-arithmetic shifting operation (0-padding) and '-' is the unary negation operation

Therefore, this technique is useful to represent small negative numbers using a smaller number of bits by storing the sign bit in the LSB instead of MSB. An example of zigzag encoding can be given as follows:

Original value: -1

Zigzag encoding: $(-1 >> 7) \text{ ^ } (-1 << 1) = 1$

Zigzag decoding: $(1 >>> 1) \text{ ^ } - (1 \,\&\, 1) = -1$

3.1.5 Binary Packing

This technique is closely related to Frame-Of-Reference (Goldstein et al. 1998, Delbru et al. 2012) In Binary packing technique, the data values are partitioned into blocks (e.g., blocks containing 128 integers) (Daniel Lemire and Leonid Boytsov 2015). The range of values in the blocks are first coded and then all the values in the block are written in reference to the range of values.

For example, if the values in a block are integers in the range [1200, 1327], then they can be stored using 7 bits per integer, i.e., $\log_2(1327 - 1200 + 1) = 7$, as offsets from the number 1200 stored in binary notation.

This technique is very efficient if the lower and upper bound values in the block are large and their difference is small.

3.1.6 XOR-based encoding

XOR-based encoding (Pelkonen et al. 2015) is used for encoding floating point values. In this scheme, successive floating-point numbers are XOR-ed together, which means that only the different bits are stored. Techniques such as delta encoding don't generally work very well for floats, as they do not reduce the number of bits sufficiently. Floating point numbers are generally more difficult to compress than integers. Unlike fixed-length integers which often have a fair number of leading 0s, floating point numbers often use all of their available bits.

In this encoding, the first data value is stored with no compression. Subsequent data values are represented using their XOR-ed values, encoded using a bit packing scheme by removing the trailing and leading zeros in the XOR-ed representation. For example, refer to figure 3.1.

Decimal	Double Representation	XOR with previous
12	0x4028000000000000	
24	0x4038000000000000	0x0010000000000000
15	0x402e000000000000	0x0016000000000000
12	0x4028000000000000	0x0006000000000000
35	0x4041800000000000	0x0069800000000000

Decimal	Double Representation	XOR with previous
15.5	0x402f000000000000	
14.0625	0x402c200000000000	0x0003200000000000
3.25	0x400a000000000000	0x0026200000000000
8.625	0x4021400000000000	0x002b400000000000
13.1	0x402a333333333333	0x000b733333333333

Figure 3.1: An Example of XOR Based Encoding (Pelkonen et al. 2015)

3.1.7 Lempel-Ziv (LZ) Compression

Lempel-Ziv compression (History of Lossless Data Compression Algorithms 2019) is a contains a family of several compression algorithms and variants. It takes advantage of the large amounts of repetitive data in a file. Each time we hit one of these common words, we could just put a shorter code for this word. Some popular variants of Lempel-Ziv compression are LZW, LZ77, LZ78, LZMA, etc.

3.1.8 LZ77 Compression

LZ77 compression (Ziv J., Lempel A., 1977) works by looking ahead into the file. If it sees a pattern it recognizes, it will write the previous position of that match in a file instead of the actual data. LZ77 works by encoding scanned data values using a triple (o, l, c) where,

o: offset, represents the number of positions that we would need to move backwards in order to find the start of the matching string.

l: length, represents the length of the match.

c: character, represents the character that is found after the match.

Given below is an example of how data compressed using LZ77 compression would look like:

Original data: a b a b c b a b a b a a

LZ77 encoding: (0, 0, a), (0, 0, b), (2, 2, c), (4, 3, a), (2, 2, a)

3.1.9 Asymmetric Numeral Systems (ANS)

ANS (Duda, J., 2013) is a lossless and entropy encoding compression algorithm. Its input is a list of symbols from some finite set and its output is a finite integer. Each symbol s has a fixed known probability p_s of occurring in the list. The algorithm tries to assign each list a unique integer so that the more probable lists get smaller integers.

We convert each symbol to a number from 0 to B−1 (where B is the number of symbols), add a leading 1 to avoid ambiguities caused by leading zeros, and interpret the list as an integer written in a base-B positional system.

An example of how the encoder and decoder works for its simplest variant (Roman Cheplyaka 2017) is given below:

Encoding: $f(s, n) = s + n.B$

Decoding: $g(n) = (n \bmod B, [n/B])$

where, s is the symbol being encoded, B is the number of symbols and n is the current state of the encoded number prior to encoding symbol s.

There are several popular variants of ANS such as tANS, rANS, uANS. ANS offers several important advantages in comparison to other entropy encoders such as Huffman Coding (HUFFMAN, D. A., 1952) and Arithmetic Coding (RISSANEN, J., AND LANGDON, G. G., 1979). Huffman Coding is generally very fast but does not compress close to the entropy limit for the data and Arithmetic Coding compresses close to the entropy limit but is slow compared to Huffman coding. ANS offers the

best of both worlds by being efficient both in terms of degree of compression as well as performance.

3.2 Research Questions

There are two major concerns with most of the compression algorithms available today:

1. Most compression algorithms are generic in nature, i.e., they do not adapt with respect to the data characteristics to get a better compression ratio.

   For example:

   a. Delta encoding is not favorable if the difference between adjacent values is large.

   b. Delta encoding and delta of delta encoding may in some cases increase the size of the original file.

   c. Frame of Reference or binary packing is not favorable if there is the minimum value in the data block is 0 or the range of values in a block is very large.

   d. ANS produces a large integer result if set of input symbols are large.

2. Most compression algorithms are suitable for particular data types.

   For example:

   a. Delta, Delta of Delta, Run-length are good for Integer types.

   b. XOR encoding is good for float types.

   c. LZ-type compression is good for String types.

Therefore, our thesis work caters to providing a solution to the following research questions:

1. Can we devise a smart compression algorithm which adapts its compression technique with respect to dataset characteristics?

2. Can we devise a smart compression algorithm which adapts its compression technique with respect to the datatype?

To answer the above questions, we devise a new compression technique called SA128 (covered in chapter 4 and 5) which adapts its compression technique with respect to both the datatype and data characteristics of our data.

An example of a database which tackles the second questions is TimescaleDB. TimescaleDB (Freedman 2019) turns a row-oriented database into a column-oriented format and adapts its compression strategy for each of the supported datatypes such as integers, floats, strings, etc. This approach results in 91-96% storage savings for TimescaleDB.

Chapter 4

SA128 COMPRESSION

In the chapter 3, we discussed some background and common methods used in the area of compression. In this section we will build on top of some of those techniques and explore a smart compression technique called SA128. SA128 compression takes place in two stages:

1. Compression Stage 1 - Column based compression stage: Uses a smart compression technique which uses common data characteristics of columns for compression and adapts its compression logic with respect to the data distribution.

2. Compression Stage 2 – rANS entropy encoding stage: Uses an rANS variant of Asymmetric Numeral Systems (Duda, J., 2013) to compress the set of symbols received as output from stage 1 into an integer.

4.1 Supported Data Types

SA128 supports most of the commonly used datatypes in databases. For our implementation, we have used the datatypes available in PostgreSQL (PostgreSQL 13 Documentation, 2021) as many columnar databases are built on top of PostgreSQL or follow similar data type conventions. Note that certain data types may differ in their definition from database to database. The internal details of the algorithm can be extended to support these differences.

The data types which we will be compressing are:

1. Integer/Int4 – Stores whole numbers using 4 bytes of storage with a range of values from -2147483648 to +2147483647 (PostgreSQL 13 Documentation, 2021). The integer type is the most common choice as it offers the best balance between range, storage size and performance.

2. Smallint/Int2 – Stores whole numbers using 2 bytes of storage with a range of values from -32768 to +32767 (PostgreSQL 13 Documentation, 2021). The smallint type is generally only used if disk space is at a premium.

3. Bigint/Int8 – Stores whole numbers using 8 bytes of storage with a range of values from -9223372036854775808 to +9223372036854775807 (PostgreSQL 13 Documentation, 2021). The bigint type should only be used if the range of the integer type is insufficient, because the latter is definitely faster. On very minimal operating systems the bigint type might not function correctly, because it relies on compiler support for eight-byte integers. On such machines, bigint acts the same as integer, but still takes up eight bytes of storage.

4. Date – Stores a date literal using 4 bytes of storage with a range of values between 4713 BC and 5874897 AD and a resolution of 1 day (PostgreSQL 13 Documentation, 2021). Date is accepted in almost any reasonable format, including ISO 8601, SQL-compatible, traditional POSTGRES, and others. The ISO 8601 is the recommended format which uses the 'yyyy-mm-dd' format for storing dates.

5. Timestamp – Timestamp (without time zone) uses 8 bytes of storage to store both the date and time values concatenated into a valid timestamp literal (PostgreSQL 13 Documentation, 2021). It has a range of values between 4713 BC and 294276 AD and a resolution of 1 millisecond (14 digits). It can accept an optional precision value of p with valid values between 0 and 6, which specifies the number of fractional digits retained in the second's field.

6. Timestamp with time zone/timestamptz – Timestamptz uses 8 bytes of storage to store the date and time values along with the timezone information, all concatenated into a valid timestamp literal (PostgreSQL 13 Documentation, 2021). It has a range of values between 4713 BC and 294276 AD and a resolution of 1 millisecond (14 digits). For timestamp with time zone, the internally stored value is always in UTC (Universal Coordinated Time, traditionally known as Greenwich Mean Time, GMT). An input value that has an explicit time zone specified is converted to UTC using the appropriate offset for that time zone. If no time zone is stated in the input string, then it is assumed to be in the time zone indicated by the system's timezone parameter, and is converted to UTC using the offset for the timezone zone.

7. Time – Time uses 8 bytes of storage to store a valid time of the day as a time literal. It has a range of values from 00:00:00 to 24:00:00 and a resolution 1 microsecond, i.e.,14 digits (PostgreSQL 13 Documentation, 2021).

8. Time with time zone/timetz – Timetz uses 12 bytes of storage to store a valid time of the day along with the time zone, both concatenated into a valid timetz literal. It has a range of values from 00:00:00+1459 to 24:00:00-1459 and a resolution 1 microsecond., 14 digits (PostgreSQL 13 Documentation, 2021). It can accept an optional precision value of p with valid values between 0 and 6, which specifies the number of fractional digits retained in the second's field.

9. Numeric/Decimal – The numeric or decimal types stores arbitrary precision numbers with a very large number of digits. It stores up to 131072 digits before the decimal point and up to 16383 digits after the decimal point (PostgreSQL 13 Documentation, 2021). They have the ability to perform calculations accurately, but are however slow compared to integer or floating-point types. Two values p and s, indicating the precision and scale can be defined for numeric types. The precision of a numeric is the total count of significant digits in the whole number, that is, the number of digits to both sides of the decimal point. The scale of a numeric is the count of decimal digits in the fractional part, to the right of the decimal point. The precision must be positive, the scale zero or positive. A numeric type can be defined with either both

the precision and scale values, with only a precision value (scale is considered as 0) or without precision and scale values (in this case it stores numbers with any precision and scale up to the implementation limit).

10. Real/Float4 – They are inexact and variable-precision numeric types requiring 4 bytes of storage and storing values having up to 6 digits of precision (PostgreSQL 13 Documentation, 2021). They are usually implementations of the IEEE standard 754 for single precision binary floating-point arithmetic.

11. Double Precision/Float8 - They are inexact and variable-precision numeric types requiring 8 bytes of storage and storing values having up to 15 digits of precision (PostgreSQL 13 Documentation, 2021). They are usually implementations of the IEEE standard 754 for double precision binary floating-point arithmetic.

12. Boolean – The Boolean type stores three states: 'true', 'false' and a third 'unknown' state which is represented by the SQL null value (PostgreSQL 13 Documentation, 2021). Possible string representation for true values are 'true', 'yes', 'on', '1'. Possible string representations for false values are 'false', 'no', 'off', '0'.

13. Character/Char – Stores fixed size strings up to n characters in length. The value n is defined along with the data type (PostgreSQL 13 Documentation, 2021). If the length of the string is less than n characters, the string is padded with empty spaces to make it equal to a size of n.

14. Character Varying/Varchar – Stores variable length strings with a limit n. Stores only the number of characters equal to the length of the string without padding extra spaces at the end (PostgreSQL 13 Documentation, 2021). If the length of the string exceeds n, then the string is truncated to accommodate a maximum length of n characters.

4.2 Compression Stage 1: Column Based Compression

In this stage, we sequentially compress each column of our database tables based on the category of datatypes the column falls under. We divide our data types listed in section 4.1 into 5 categories:

1. Category 1 – Integer/int4, SmallInt/Int2, BigInt/Int8, Date, Timestamp, Timestamptz, Time, Timetz.

2. Category 2 - Numeric/Decimal.

3. Category 3 - Real/Float4, Double Precision/Flaot8.

4. Category 4 – Boolean

5. Category 5 – Character/Char, Character Varying/Varchar

Depending on the Category the column data type falls under, a different compression strategy is used to compress the respective column. In the next sections, we will explain the inner details of each of the category-wise compression steps which are a part of Stage 1 compression.

4.2.1 Compression of Category 1 data types

The compression of category 1 datatypes take place in 12 sequential steps which are explained below:

Step 1 - Divide into blocks: In this step, all the data values belonging to the column being compressed is divided into blocks of 128 values. If there are N values in the column, the total number of blocks after division will be $\lfloor N/128 \rfloor$ values where each block contains 128 values each except the last block which contains $N\%128 + 1$ values (between 1 and 128). Therefore, there cannot be an empty block which contains 0 number of values.

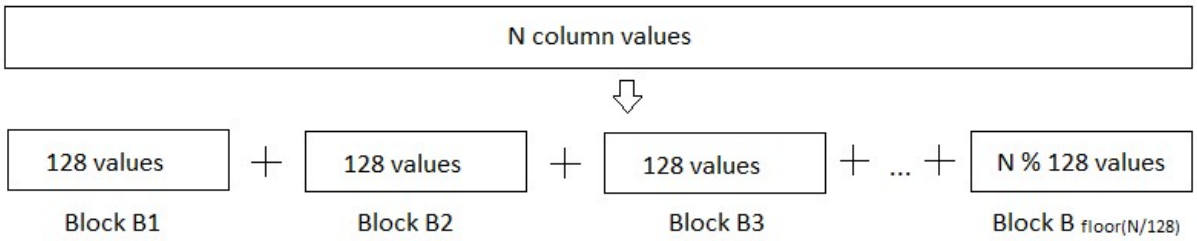

Figure 4.1: Illustration of Block Division Step in Category-1 Compression

The main reason behind choosing block sizes with 128 values are:

1. **Optimal storage savings –** Having several small sized blocks ensures that most of the blocks compress well and at the same time it makes sure the compression does not deteriorate due to poor compression of a small number of blocks. This ensures that the net average compression of all the blocks combined is high. In most

compression algorithms, blocks sizes are generally chosen to be multiples of 8 (to keep it as multiples of 1 byte). For the reasons above, SA128 has blocks of 128 values each. We did not choose block sizes with lesser than 128 values due to the overhead of metadata information in the block header which might often occupy more space if the number of data values are chosen to be 64, 32 or 16, etc.

2. **Optimal decompression performance** – If columns are divided into blocks containing small number of values, parallel processing and SIMD instructions can be used during decompression which can increase decompression performance. If there is a search key present in the column, then it can also increase query times tremendously since only selected blocks need to be decompressed instead of decompressing the entire column data.

Step 2 – Type conversion: If the data type of the column being compressed does not fall into the set of three integer types, i.e., $\{smallint, integer, bigint\}$, the values in every block is converted to an integer representation by removing all non-numeric characters from the literal. The integer representation requires 4 bytes of storage for date types, 8 bytes for timestamp, timestamptz and time types and 12 bytes for timez types. For example:

i. Let s be a literal of 'date' type, where s = '2021-12-11' (where the date value is stored in ISO 8601 format). This will be stored as a literal f = '20211211'.
ii. Let s be a literal of 'timestamp' type, where s = '2021-12-11 11:55:34.12313' (where the timestamp value is stored in ISO 8601 format). This will be stored as a literal f = '20211211115534123 13'.
iii. Let s be a literal of 'timestamptz' type, where s = '2021-12-11 11:55:34 -8:00' (where the timestamptz value is stored in ISO 8601 format). This will be stored as a literal f = '-20211211115534800'. Note that the negative sign for the timezone becomes the sign of the integer representation, thus making it negative.

iv. Let s be a literal of ‘time’ type, where s = ’11:55:34.12313’ (where the time value is stored in ISO 8601 format). This will be stored as a literal f = ‘11553412313’.

v. Let s be a literal of ‘timez’ type, where s = ’11:55:34.12313-08:00’ (where the time value is stored in ISO 8601 format). This will be stored as a literal f = ‘-115534123130800’. Note that the negative sign for the timezone becomes the sign of the integer representation, thus making it negative.

If the resultant integer representation has leading zeros, then the leading zeros are removed, i.e., the date literal ‘0010-12-11’ gets converted to ‘101211’.

Step 3 – Create block copies: For each block B_m containing 128 values, where 1 <= m <= ⌈N/128⌉, we create two more copies of it and call them B_m’ and B_m’’ respectively. In total, we have three identical blocks of 128 values each.

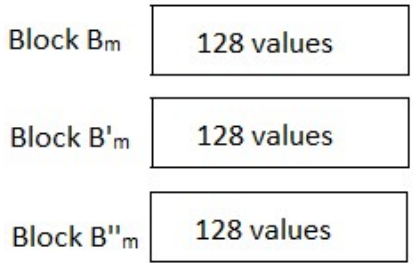

Figure 4.2: Illustration of Block Copy Creation Step in Category-1 Compression

Step 4 – Delta and Delta of Delta encoding: In this step, we leave block B_m uncompressed but apply Delta encoding on block copy B_m’ and Delta of Delta encoding on block copy B_m’’. During Delta encoding and Delta of delta encoding, if there are NULL values in the blocks B_m’ and B_m’’ respectively, we leave them as it is. During delta encoding on block B_m’, for every non-null value v, we replace it with its delta value d as $d = prev - v$ where $prev$ is the previously scanned non-null value in the block (the values $prev$ and v

do not necessarily need to be adjacent to each other as there can me any number of NULL values between them which remain unchanged). If the previously scanned non-null value $prev$ does not exist, we keep the value v unchanged. For example: [20, 15, NULL, 10] gets converted to [20, 5, NULL, 5]. Notice here that the first value remains unchanged because a non-null value does not exist prior to the first element.

During delta of delta encoding on block B_m', for every non-null value v, we replace it with its delta of delta value d' as $d' = (pprev - prev) - (prev - v)$ where $prev$ is the non-null value appearing before v and $pprev$ is the second previous non-null value before v. If $prev$ and $pprev$ do not have non-null values, then we keep the value v unchanged in the resultant encoding.

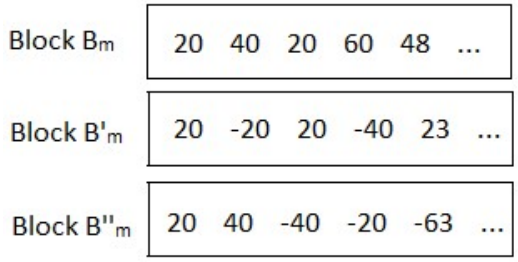

Figure 4.3: Illustration of Delta and Delta of Delta Encoding Step in Category-1 Compression

Step 5 – Zig-zag encoding: In this step, we use zig-zag encoding on all the three blocks B_m, B_m' and B_m'' to deal with negative numbers and represent them using positive numbers. We do this because it is more space efficient to store small negative numbers with a smaller number of bits by storing the sign bit in the LSB instead of MSB. Depending on the number of bits required to represent the integer representation of the data value after step 2, we represent it using a variable called 'bitlength'. The zig-zag encoder encodes the data value to a positive number based on the following equation:

$$(n >> bitlength - 1) \wedge (n << 1)$$

Where, n is the value being encoded, ">>" is the arithmetic right shift operation, "<<" is the arithmetic left shift operation and "^" is the XOR operation.

This mapping from negative to positive numbers is done in the following sequence:

$$[0 = 0, -1 = 1, 1 = 2, -2 = 3, 2 = 4, -3 = 5, 3 = 6, \dots]$$

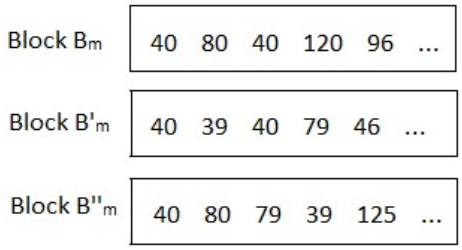

Figure 4.4: Illustration of Zig-zag Encoding in Category-1 Compression

Step 6 – Frame of Reference: In this step, we find the minimum values in each of the three blocks B_m, B_m' and B_m'', and represent them using t_m, t_m' and t_m'' respectively where, $t_m = \min(B_m)$, $t_m' = \min(B_m')$ and $t_m'' = \min(B_m'')$. We call t_m, t_m' and t_m'' the translated values for each of their respective blocks. We then subtract all data values in blocks B_m, B_m' and B_m'' by t_m, t_m' and t_m'' respectively. This brings down the magnitude of each number in the block such than a smaller number of bits can be used to represent them later. We store the values t_m, t_m' and t_m'' for our later steps where we will wrap them as part of our SA128 block header.

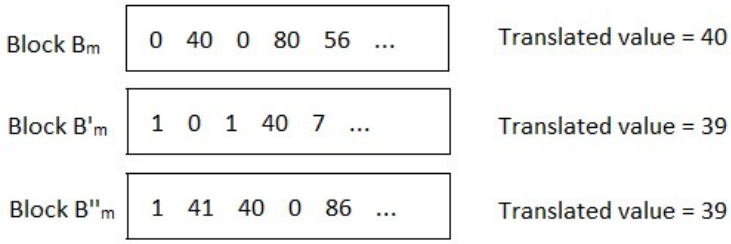

Figure 4.5: Illustration of Frame of Reference Step in Category-1 Compression

Step 7 – Run-length encoding: In this step, we try to further optimize the storage by computing a run-length encoding over all the three blocks B_m, B_m' and B_m''. For every sequence S of contiguous values within each of the three blocks, where $S = \{x_0, \dots, x_i, \dots, x_n \; where \; 0 \leq i < n \; and \; x_i = x_{i+1}\}$ we calculate the run value r, where $r = len(S)$ and len is the length of the sequence S. Note than all run values must be greater than 1, i.e., $r > 1$ since the minimum length of a sequence S is 2, i.e., $min(len(S)) = 2$. The sequence of data values $x_0, \dots, x_i, \dots, x_n$ in S is replaced with the pair $\{x_0, r\}$ in the block. We create an exception block for each of the blocks B_m, B_m' and B_m'' and call them $E1_m$, $E1_m$' and $E1_m$'' respectively. These exception blocks store the index location for each run value 'r' in their respective blocks.

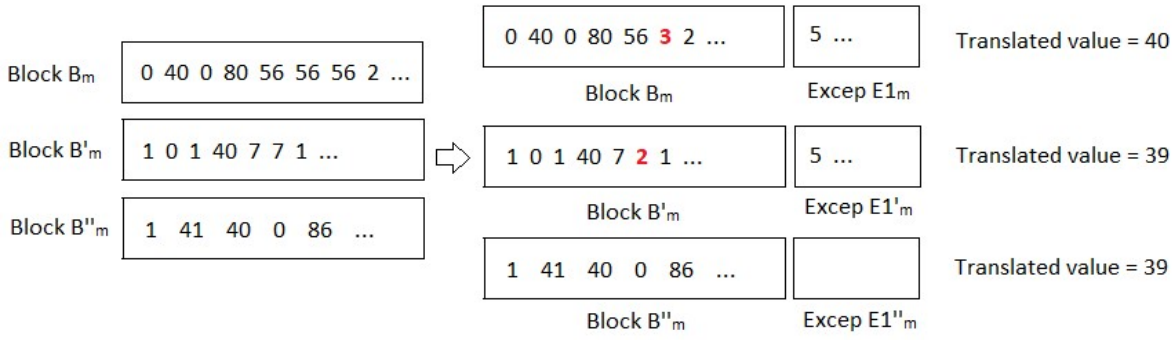

Figure 4.6: Illustration of Run Length Encoding Step in Category-1 Compression

Step 8 – Bit packing using modulo technique: In this step, we perform an important operation of the three blocks B_m, B_m' and B_m'' where we try to break down the largest values into smaller values. The intuition behind doing so is that the largest values in the blocks dictate the number of bits with which each value will be represented during bit packing. If the largest value can be represented using lesser number of bits, then so can each other value in the block. We do this by following a modulo technique for bit packing specifically designed to optimize the blocks in SA128. For each of the three blocks B_m, B_m' and B_m'', we find the minimum number of bits b_{min} (b_{min}' and b_{min}'' for the copies)

required to represent the majority of the values in the block (> 50% values in the block). We also find the number of bits b_{max} (b_{max}' and b_{max}'' for the copies) required to represent the largest value in the block. The space occupied by the block after bit packing is dictated by the value b_{max} (b_{max}' and b_{max}'' for the copies). Therefore, the total number of bits required to represent the values as well as the indices in their respective exception blocks are:

$$S_m = N_1 . b_{max} + N_2 . 8$$

$$S_m' = N_1' . b_{max}' + N_2' . 8$$

$$S_m'' = N_1'' . b_{max}'' + N_2'' . 8$$

where,

S_m = Number of bits required to represent the blocks B_m and $E1_m$,

S_m' = Number of bits required to represent the blocks B_m' and $E1_m'$,

S_m'' = Number of bits required to represent the blocks B_m' and $E1_m''$,

N_1 = Number of values in block B_m,

N_1' = Number of values in block B_m',

N_1'' = Number of values in block B_m'',

N_2 = Number of values in exception block $E1_m$,

N_2' = Number of values in exception block $E1_m'$,

N_2'' = Number of values in exception block $E1_m''$.

Here, the constant value 8 is the number of bits required to represent each index value in the exception blocks $E1_m$, $E1_m'$ and $E1_m''$ since the number of possible values lie between 0 to 127.

We see that the value b_{max} (b_{max}' and b_{max}'' for the copies) dictate the values S_m, S_m' and S_m'' respectively. We now explain how the modulo technique works on block B_m. The same technique is used for blocks B_m' and B_m'' as well. The modulo technique takes as input the block B_m and the exception block $E1_m$ and tries to break down large

values in the block B_m into two small values such than they can be represented using an optimal number of bit b where $b_{min} \leq b \leq b_{max}$. It returns us a modified block B_m, the optimal number of bits b required to represent B_m and a second exception block $E2_m$ which contains the indices of the values which are broken down into two smaller values.

Pseudo-code for bit packing using modulo technique:

$function\ bitPackingWithModulo(B_m, E1_m)$:

$E2_m = [\,];$

$N_1 = len(B_m);$

$N_2 = len(E1_m);$

$S_m = N_1 . b_{max} + N_2 . 8;$

$S_{prev} = S_m;$

$B_{prev} = B_m;$

$E2_{prev} = E2_m;$

$b_{max} = Bits\ required\ to\ represent\ the\ largest\ element\ in\ B_m;$

$b_{min} = Bits\ required\ to\ represent\ more\ than\ half\ the\ elements\ in\ B_m;$

$for\ b_{max} \geq b \geq b_{min}$:

$maxVal =$ *the largest value which can be represented using b bits*;

$B_m^{new} = [\,];$

$E2_m^{new} = [\,];$

$for\ 0 \leq i \leq len(B_m);$

$n = B_m[i];$

$if\ n > maxVal$:

$Push\ \left\{\left\lfloor \frac{n}{b} \right\rfloor, n\ \%\ b\right\} into\ B_m^{new};$

$Push\ i\ into\ E2_m^{new};$

$else$:

$Push\ n\ into\ B_m^{new}$;

$S_m^{new} = len(B_m^{new}) * b + len(E1_m^{new}) * 8 + len(E2_m^{new}) * 8$;

$if\ S_m^{new} > S_{prev}$:

$return\ \{B_{prev}, E2_{prev}, (b+1)\}$;

$else$:

$S_{prev} = S_m^{new}$;

$B_{prev} = B_m^{new}$;

$E2_{prev} = E2_m^{new}$;

$Return\ \{B_{prev}, E2_{prev}.b_{min}\}$;

What bit packing with modulo does is that it tries to represent the block B_m with an optimal number of bits b. The value of b is found out greedily from b_{max} to b_{min}. If there are any elements in the block which require more than b bits to represent, we break that number n down into a pair $\left\{\left\lfloor\frac{n}{b}\right\rfloor, n\ \%\ b\right\}$ which is the quotient and remainder when the number n is divided by b. This guarantees than the value $\left\lfloor\frac{n}{b}\right\rfloor \leq n$ and $n\ \%\ b < b$. Therefore, it requires a smaller number of bits to represent n compared to each of the two values $\left\lfloor\frac{n}{b}\right\rfloor$ and $n\ \%\ b$. While doing this for each value of b, we calculate the total number of bits required to represent the new block and its two exception blocks. If the size decreases, we keep trying for smaller and smaller values of b. If the size at any point increases, this means that too many numbers have been broken down into smaller values in our new block which has resulted in an overhead. This is the point we break and return the previous optimal value of b.

Step 9 – Packing NULL values: If the blocks B_m, B_m' and B_m'' contain NULL values, we replace all the NULL values by 0 and store the index positions of the NULL values in a third exception block $E3_m$ ($E3_m$' and $E3_m$' for the block copies).

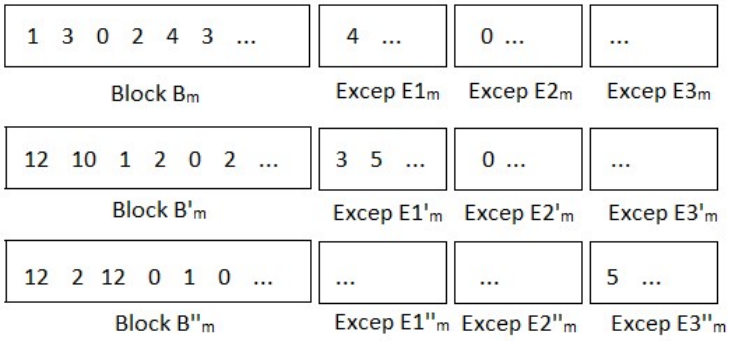

Figure 4.7: Illustration of NULL Packing Step in Category-1 Compression

Step 10- Bit packing of exception blocks: By this stage, for each of the blocks B_m, B_m' and B_m'', we have three exception blocks each, i.e., $E1_m$, $E2_m$ and $E3_m$ for block B_m, $E1_m$', $E2_m$' and $E3_m$' for block B_m' and $E1_m$'', $E2_m$'' and $E3_m$'' for block B_m''. To optimize the number of bits with which each of the exception blocks are represented, we use bit packing on each of the exception blocks. We represent each value in the three exception blocks using e_1, e_2 and e_3 bits (e_1', e_2', e_3' and e_1'', e_2'', e_3'' for the block copies respectively), where,

e_1 = Number of bits required to represent the largest value in $E1_m$,

e_2 = Number of bits required to represent the largest value in $E2_m$,

e_3 = Number of bits required to represent the largest value in $E3_m$,

e_1' = Number of bits required to represent the largest value in $E1_m$',

e_2' = Number of bits required to represent the largest value in $E2_m$',

e_3' = Number of bits required to represent the largest value in $E3_m$',

e_1'' = Number of bits required to represent the largest value in $E1_m$'',

e_2'' = Number of bits required to represent the largest value in $E2_m$'',

e_3'' = Number of bits required to represent the largest value in $E3_m$''.

Step 11 – Block copy selection: By this stage, the total number of bits S_m, S_m' and S_m'' required to represent each block B_m, B_m' and B_m'' and their three exception blocks respectively can be given below:

$$S_m = N_1 * b + N_2 * e_1 + N_3 * e_2 + N_4 * e_3$$

$$S_m{}' = N_1{}' * b' + N_2{}' * e_1{}' + N_3{}' * e_2{}' + N_4{}' * e_3{}'$$

$$S_m{}'' = N_1{}'' * b'' + N_2{}'' * e_1{}'' + N_3{}'' * e_2{}'' + N_4{}'' * e_3{}''$$

where,

$N_1, N_1' \; and \; N_1{}''$ are the number of elements in blocks B_m, B_m' and B_m'' respectively.

$b, b' \; and \; b''$ are the number of bits required to represent each element in blocks B_m, B_m' and B_m'' respectively.

$N_2, N_2' \; and \; N_2{}''$ are the number of elements in exception blocks $E1_m$, $E1_m$' and $E1_m$'' respectively.

$e_1, e_1{}' \; and \; e_1{}''$ are the number of bits required to represent each element in blocks $E1_m$, $E1_m$' and $E1_m$'' respectively.

$N_3, N_3' \; and \; N_3{}''$ are the number of elements in exception blocks $E2_m$, $E2_m$' and $E2_m$'' respectively.

$e_2, e_2{}' \; and \; e_2{}''$ are the number of bits required to represent each element in blocks $E2_m$, $E2_m$' and $E2_m$'' respectively.

$N_4, N_4' \; and \; N_4{}''$ are the number of elements in exception blocks $E3_m$, $E3_m$' and $E3_m$'' respectively.

$e_3, e_3{}' \; and \; e_3{}''$ are the number of bits required to represent each element in blocks $E3_m$, $E3_m$' and $E3_m$'' respectively.

In this stage, we select the optimal block (and its three exception blocks) where the total number of bits required to represent it is $S_m^{opt} = \min(S_m, S_m{}', S_m{}'')$ and reject the other two blocks (and their three exception blocks each). Let's call the optimal block B_m^{opt}

and the respective exception blocks $E1_m^{opt}$, $E2_m^{opt}$ and $E3_m^{opt}$. We also select the respective translated value (from step 6) as t_m^{opt}.

After selection of the optimal block B_m^{opt} and its exception blocks $E1_m^{opt}$, $E2_m^{opt}$ and $E3_m^{opt}$, we record the encoding with which the block was encoded with in step 4 using a variable Enc_m:

i. If the selected block was uncompressed, $Enc_m = 0$.

ii. If the selected block was encoded with delta encoding, $Enc_m = 1$.

iii. If the selected block was encoded with delta of delta encoding, $Enc_m = 2$.

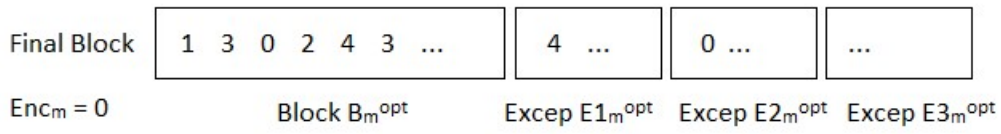

Figure 4.8: Illustration of NULL Packing Step in Category-1 Compression

Step 12 – Encode as SA128 block: We encode our selected block B_m^{opt} and exception blocks $E1_m^{opt}$, $E2_m^{opt}$ and $E3_m^{opt}$ by wrapping them within a category-1 SA128 block. The category-1 SA128 block consists of two parts:

i. A category-1 SA128 block header – Contains metadata information about the block.

ii. A category-1 SA128 block body – Contains the data encoded within the block.

The components of a category-1 SA128 block header and block body can be given below:

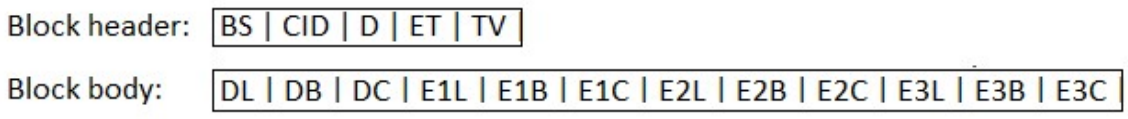

Figure 4.9: Components of SA128 Block for Category-1 Compression

i. **BS - Block size in bytes (32 bits):** This is a 32-bit representation for the size of the entire block (block header and block body. 32 bits are adequate to represent the size

of the entire block irrespective of the datatype category being encoded. We will elaborate on this further in the later sections.

ii. **CID - ColumnID (11 bits):** Stores the ID of the column being compressed. PostgreSQL tables are hard limited to a maximum of 1600 tables. Therefore, each column ID can be represented using 11 bits (bits required to represent 1600 is 11).

iii. **D - Datatype (4 bits):** There are 14 datatypes supported by SA128. Therefore, each datatype can be represented using 4 bits. The datatype IDs for each of the 14 supported datatypes is given by the table below:

Datatype	Datatype ID
SmallInt/Int2	0
Integer/Int4	1
Bigint/Int8	2
Date	3
Timestamp	4
Timestamp with timezone/Timestamptz	5
Time	6
Time with timezone/Timez	7
Numeric/Decimal	8
Real/Float4	9
Double Precision/Float8	10
Boolean	11
Character/Char	12
Character Varying/Varchar	13

Table 4.1: Mapping of Supported Datatypes to Datatype IDs

iv. **ET – Encoding Type (2 bits):** This stands for the Enc_m^{opt} value with which the block B_m^{opt} was encoded with. The following table describes the Enc_m^{opt} values for each type of encoding:

Encoding	Enc_m^{opt}
Uncompressed	0
Delta encoding	1
Delta of Delta encoding	2

Table 4.2: Mapping of Encoding Type to Enc_m^{opt} Values

The three possible Enc_m values can be encoded using 2 bits.

v. **TV – Translated Value (16/32/64 or 96 bits):** This stands for the translated value t_m^{opt} for the selected block B_m^{opt}. For Smallint/Int2 datatype, this value can be represented using 16 bits. For Integer/Int4 and Date datatypes, this value can be represented using 32 bits. For Bigint/Int8, Timestamp, Timestamptz and Time datatypes, this value can be represented using 64 bits. For timez datatype, this value can be represented using 96 bits. These 16/32/64 and 96 bits respectively are the number of bits required to represent the integer representation of each of these supported category 1 datatypes (as established in step 2).

vi. **DL – Data Length (8 bits):** The maximum number of elements present in B_m^{opt} in the worst case can be 256. This case arises when there are no runs with run value $r > 1$ after run-length encoding in step 7 and when all values are broken down into pairs of quotient and remainder after bit-packing with modulo in step 8. Since we start with 128 values in each block, this scenario could double the number of elements in our block leading to 256 values in the block. Therefore, 8 bits is adequate to represent a max block length of 256.

vii. **DB – Data Bits (8 bits):** Let b^{opt} is the number of bits required to represent each value in B_m^{opt}. In the worst case, b^{opt} can be equal to b_{max}^{opt} after step 8 which is equal

to the number of bits required to represent the largest value in B_m^{opt}. The largest possible category 1 value for timez type requires 96 bits to represent. Hence, 8 bits is sufficient to represent the value b^{opt}.

viii. **DC – Data Content (DL * DB bits):** This represents the number of bits required to represent the data values within the block B_m^{opt}. This number of bits required is given by the Data Length (DL) * Data Bits (DB).

ix. **E1L = Excep1 Length (8 bits):** During step 7, the largest index value that can be stored in the exception block $E1_m^{opt}$ is 127. Let us understand why this is the case with the help of the below two lemmas:

Lemma 4.2.1.1: The number of values in the block n becomes less than 128 if there is at least one sequence $S = \{x_0, \dots, x_i, \dots, x_n \; where \; 0 \leq i < n \; and \; x_i = x_{i+1}\}$ with the run value r > 2.

Explanation: A sequence S with run value r > 2 will be represented by 2 values, i.e., the pair $\{x_0, \text{r}\}$. This reduces the total number of values in the block leading to n < 128.

Lemma 4.2.1.2: The number of values in the block n is equal to 128 if there is no sequence $S = \{x_0, \dots, x_i, \dots, x_n \; where \; 0 \leq i < n \; and \; x_i = x_{i+1}\}$ with r > 2.

Explanation: For a sequence $S = \{x_0, \dots, x_i, \dots, x_n \; where \; 0 \leq i < n \; and \; x_i = x_{i+1}\}$ with run value r = 2, we replace the two repeated values in the sequence $x_0 \; and \; x_1$ with the pair $\{x_0, \text{r}\}$. This leads to no change in the number of values n in the block.

Lemma 4.2.1.3: There can be a maximum of 64 index values in an $E1_m^{opt}$ block.

Explanation: From lemmas Lemma 4.2.1.1 and Lemma 4.2.1.2, we can conclude that if we have a block where all distinct values form a sequence with run values r = 2, we can have a maximum of 64 such sequences where the 64 different index values will be {1, 3, 5, …, 127}.

Therefore 8 bits is adequate to represent a maximum index value of 127 in the $E1_m^{opt}$ block.

x. **E1B – Excep1 Bits (8 bits):** This represents the number of bits $e1_m^{opt}$ required to represent each value in the exception block $E1_m^{opt}$. From lemma 4.2.1.3, we can conclude that if we have a block where all distinct values form a sequence with run values r = 2, the indices required to represent each of those sequences are {1, 3, 5, …, 127}. Since 127 is the largest index that an $E1_m^{opt}$ block can contain, we can represent this value using a maximum of 8 bits.

xi. **E1C – Excep1 Content (E1L * E1B bits):** This represents the number of bits required to represent the index values within the exception block $E1_m^{opt}$. This number of bits required is given by the Excep1 Length (E1L) * Excep1 Bits (E1B).

xii. **E2L – Excep2 Length (8 bits):** There can be a maximum of 128 index values in exception block $E2_m^{opt}$. This can be proved using the following lemma:

Lemma 4.2.1.4: There can be a maximum of 128 index values in exception block $E2_m^{opt}$.

Explanation: This is because from lemma 4.2.1.2, if there is no sequence $S = \{x_0, \dots, x_i, \dots, x_n \; where \; 0 \leq i < n \; and \; x_i = x_{i+1}\}$ with r > 2, our block B_m^{opt} will contain exactly 128 values. In this block B_m^{opt} with 128 values, if all the values were broken down into a quotient and remainder pair in the bit packing with modulo step (step 8), this would double the values in the block, therefore leading to a maximum of 256 values. The number of index values required to represent the position of each broken value in B_m^{opt} is 128 which can be represented using 8 bits.

xiii. **E2B – Excep2 Bits (8 bits):** This represents the number of bits $e2_m^{opt}$ required to represent each value in the exception block $E2_m^{opt}$. From lemma 4.2.1.4, we can infer that the index values required to represent a maximum of 128 values in $E2_m^{opt}$ are {0,

2, …, 255}. Since 255 is the largest index value that can exist in $E2_m^{opt}$. Therefore $e2_m^{opt}$ can be adequately represented using 8 bits.

xiv. **E2C – Excep2 Content (E2L * E2B bits):** This represents the number of bits required to represent the index values within the exception block $E2_m^{opt}$. This number of bits required is given by the Excep2 Length (E2L) * Excep2 Bits (E2B).

xv. **E3L – Excep3 Length (8 bits):** This represents the length of the exception block $E3_m^{opt}$. The maximum possible length for an exception block $E3_m^{opt}$is 64. This can be explained with the help of the following lemma:

Lemma 4.2.1.5: There can be a maximum of 64 index values in exception block $E3_m^{opt}$.

Explanation: To have the maximum number of NULL values in B_m^{opt} in step 10, before the run length-encoding step (step 7), our block should have one of the following sequences with 128 values: {NULL, x_0, NULL, x_1, …, NULL, x_n} or {x_0, NULL, x_1, NULL, …, x_n, NULL} where x_0 to x_n are non-NULL integer representations of block values in B_m^{opt}. Note that we do not have any contiguous NULL values in both the above sequences. If we did, we would replace the contiguous NULL values with a single NULL value followed by the run value r after the run-length encoding in step 7. This would reduce the number of NULL values in block B_m^{opt}. Therefore, only for the above two sequences, we can have the maximum number of null values in our final block B_m^{opt} which contains a maximum of 64 indices to 64 null values. These 64 index values can be adequately represented using 8 bits.

xvi. **E3B – Excep3 Bits (8 bits):** This represents the number of bits $e3_m^{opt}$required to represent each value in the exception block $E3_m^{opt}$. The largest possible index value in $E3_m^{opt}$is 191 which can be represented using 8 bits. We can understand why this is the case using the below lemma:

Lemma 4.2.1.6: The largest possible index value in $E3_m^{opt}$ is 191.

Explanation: From the explanation provided for lemma 4.2.1.5, the two block sequences {NULL, x_0, NULL, x_1, …, NULL, x_n} or {x_0, NULL, x_1, NULL, …, x_n, NULL} before step 7 result in 64 NULL values in the block after step 7. During the bit packing using modulo in step 8, if all our values 64 values from x_0 to x_n are broken down into a quotient and remainder pair, this will result in a total of 128 non-NULL values and 64 NULL values (a total of 192 values). The index of the last NULL value in this case will be 191.

xvii. **E3C – Excep3 Content (E3L * E3B bits):** This represents the number of bits required to represent the index values within the exception block $E3_m^{opt}$. This number of bits required is given by the Excep3 Length (E3L) * Excep3 Bits (E3B).

4.2.2 Compression of Category 2 data types

The compression of category 2 datatypes take place in 4 sequential steps which are explained below:

Step 1 – Divide into blocks: In this step, all the data values belonging to the column being compressed is divided into blocks of 128 values. If there are N values in the column, the total number of blocks after division will be $\lfloor N/128 \rfloor$ values where each block contains 128 values each except the last block which contains $N\%128 + 1$ values (between 1 and 128). Refer to figure 4.1. Therefore, there cannot be an empty block which contains 0 number of values.

Step 2 – Divide into sub-blocks: Separate each block B_m containing 128 values into two sub-blocks $B1_m$ and $B2_m$, where 1 <= m <= ⌈N/128⌉. For each value m in the block B_m, $B1_m$ contains the part of the number m before the decimal point and $B2_m$ contains the part of the number m after the decimal point.

For example: If B_m contains the values [1.23, 4.43, 1.44, …], then,

$B1_m$ = [1, 4, 1, …] and $B2_m$ = [23, 43, 44, …] respectively.

Step 3 – Compress sub-blocks using Category 1 compression algorithm: Since numeric types have a fixed precision p and scale s, the size of values in $B1_m$ and $B2_m$ are comparable. Therefore, we compress each of the two sub-blocks $B1_m$ and $B2_m$ using category 1 compression from steps 3 to step 11 which includes creating block copies, delta and delta of delta encoding, zig-zag encoding, frame of reference, run-length encoding, bit packing using modulo technique, packing NULL values, bit packing exception blocks and block copy selection. Let the resultant sub-blocks after category 1 compression of $B1_m$ be $B1_m^{opt}$ and the resultant exception blocks be $E11_m^{opt}$, $E21_m^{opt}$ and $E31_m^{opt}$. Let the resultant sub-blocks after category 1 compression of $B2_m$ be $B2_m^{opt}$ and the resultant exception blocks be $E12_m^{opt}$, $E22_m^{opt}$ and $E32_m^{opt}$.

Step 4 – Encode as SA128 block: We encode our resultant sub-blocks $B1_m^{opt}$, $B2_m^{opt}$ and exception blocks $E11_m^{opt}$, $E21_m^{opt}$, $E31_m^{opt}$, $E12_m^{opt}$, $E22_m^{opt}$ and $E32_m^{opt}$ by wrapping them within a category-2 SA128 block. The category-2 SA128 block consists of two parts:

i. A category-2 SA128 block header – Contains metadata information about the block.
ii. A category-2 SA128 block body – Contains the data encoded within the block.

The components of a category-2 SA128 block header and block body can be given below:

Block header:	BS | CID | D | ET1 | TV1 | ET2 | TV2
Block body:	DL1 | DB1 | DC1 | DL2 | DB2 | DC2 |
	E1L1 | E1B1 | E1C1 | E1L2 | E1B2 | E1C2
	E2L1 | E2B1 | E2C1 | E2L2 | E2B2 | E2C2
	E3L1 | E3B1 | E3C1 | E3L2 | E3B2 | E3C2

Figure 4.10: Components of SA128 Block in Category-2 Compression

i. **BS – Block size in bytes (32 bits):** Stores the size of the category-2 block (block header and the block body) using 32-bits.

ii. **CID – Column ID (11 bits):** Uses 11 bits to represent the column ID of the column being compressed. Explanation of the bit representation is same as that provided for the 'CID' section in category-1 SA128 block header.

iii. **D – Datatype (4 bits):** Uses 4 bits to represent the data type on the column being compressed. Explanation of the bit representation is same as that provided for the 'D' section in category-1 SA128 block header.

iv. **ET1 – $B1_m^{opt}$ Encoding Type (2 bits):** Uses 2 bits to represent the encoding type (uncompressed, delta or delta of delta) for sub-block $B1_m^{opt}$. Explanation of the bit representation is same as that provided for the 'ET' section in category-1 SA128 block header.

v. **TV1 – $B1_m^{opt}$ Translated Value (t_1 bits):** Uses t_1 bits to represent the translated value $t1_m^{opt}$after frame of reference in step 6 on sub-block $B1_m^{opt}$, where $t_1 = \lceil \log_2(10^{(p-s)} - 1) \rceil$, p is the precision and s is the scale for the numeric type. The max length of a value in $B1_m^{opt}$ is (p – s) which is equal to the part of the number before the decimal point in the main block B_m^{opt}. The largest value possible for $t1_m^{opt}$ is equal to the largest value which can be formed with (p – s) digits, i.e., $10^{(p-s)} - 1$. Therefore, the number of bits required to represent the number $10^{(p-s)} - 1$ is $t_1 = \lceil \log_2(10^{(p-s)} - 1) \rceil$.

vi. **ET2 – $B2_m^{opt}$ Encoding Type (2 bits):** Uses 2 bits to represent the encoding type (uncompressed, delta or delta of delta) for sub-block $B2_m^{opt}$. Explanation of the bit representation is same as that provided for the 'ET' section in category-1 SA128 block header.

vii. **TV2 – $B2_m^{opt}$ Translated Value (t_2 bits):** Uses t_2 bits to represent the translated

value $t2_m^{opt}$after frame of reference in step 6 on sub-block $B2_m^{opt}$, where $t_2 = \lceil \log_2(10^s - 1) \rceil$ and s is the scale for the numeric type. The max length of a value in $B2_m^{opt}$ is s which is equal to the part of the number after the decimal point in the main block B_m^{opt}. The largest value possible for $t2_m^{opt}$ is equal to the largest value which can be formed with s digits, i.e., $10^s - 1$. Therefore, the number of bits required to represent the number $10^s - 1$ is $t_2 = \lceil \log_2(10^s - 1) \rceil$.

viii. **DL1 – $B1_m^{opt}$ Data Length (8 bits):** Uses 8 bits to represent the length of the $B1_m^{opt}$ block. Explanation of the bit representation is same as that provided for the 'DL' section in category-1 SA128 block body.

ix. **DB1 – $B1_m^{opt}$ Data Bits (b_1 bits):** Uses b_1 bits to represent the number of bits required for representing each value in the sub-block $B1_m^{opt}$, where $t_1 = \left\lceil \log_2(10^{(p-s)} - 1) \right\rceil$, p is the precision, s is the scale for the numeric type and $b_1 = \lceil \log_2(t_1) \rceil$.

x. **DC1 – $B1_m^{opt}$ Data Content (DL1 * DB1 bits):** Uses DL1 * DB1 bits to represent all the values in sub-block $B1_m^{opt}$.

xi. **DL2 – $B2_m^{opt}$ Data Length (8 bits):** Uses 8 bits to represent the length of the $B2_m^{opt}$ block. Explanation of the bit representation is same as that provided for the 'DL' section in category-1 SA128 block body.

xii. **DB2 – $B2_m^{opt}$ Data Bits (b_2 bits):** Uses b_2 bits to represent the number of bits required for representing each value in the sub-block $B2_m^{opt}$, where $t_2 = \lceil \log_2(10^s - 1) \rceil$, s is the scale for the numeric type and $b_2 = \lceil \log_2(t_2) \rceil$.

xiii. **DC2 – $B2_m^{opt}$ Data Content (DL2 * DB2 bits):** Uses DL2 * DB2 bits to represent all the values in sub-block $B2_m^{opt}$.

xiv. **E1L1 – $B1_m^{opt}$ Excep1 Length (8 bits):** Uses 8 bits to represent the length of the exception block $E11_m^{opt}$. Explanation of the bit representation is same as that

provided for the ‘E1L’ section in category-1 SA128 block body.

xv. **E1B1 – $\boldsymbol{B1_m^{opt}}$ Excep1 Bits (8 bits):** Uses 8 bits to represent the bit required to represent each index value in exception block $E11_m^{opt}$. Explanation of the bit representation is same as that provided for the ‘E1B’ section in category-1 SA128 block body.

xvi. **E1C1 – $\boldsymbol{B1_m^{opt}}$ Excep1 Content (E1L1 * E1B1 bits):** Uses E1L1 * E1B1 bits to represent all the indices in the exception block $E11_m^{opt}$.

xvii. **E2L1 – $\boldsymbol{B1_m^{opt}}$ Excep2 Length (8 bits):** Uses 8 bits to represent the length of the exception block $E21_m^{opt}$. Explanation of the bit representation is same as that provided for the ‘E2L’ section in category-1 SA128 block body.

xviii. **E2B1 – $\boldsymbol{B1_m^{opt}}$ Excep2 Bits (8 bits):** Uses 8 bits to represent the bit required to represent each index value in exception block $E21_m^{opt}$. Explanation of the bit representation is same as that provided for the ‘E2B’ section in category-1 SA128 block body.

xix. **E2C1 – $\boldsymbol{B1_m^{opt}}$ Excep2 Content (E2L1 * E2B1 bits):** Uses E2L1 * E2B1 bits to represent all the indices in the exception block $E21_m^{opt}$.

xx. **E3L1 – $\boldsymbol{B1_m^{opt}}$ Excep3 Length (8 bits):** Uses 8 bits to represent the length of the exception block $E31_m^{opt}$. Explanation of the bit representation is same as that provided for the ‘E3L’ section in category-1 SA128 block body.

xxi. **E3B1 – $\boldsymbol{B1_m^{opt}}$ Excep3 Bits (8 bits):** Uses 8 bits to represent the bit required to represent each index value in exception block $E31_m^{opt}$. Explanation of the bit representation is same as that provided for the ‘E3B’ section in category-1 SA128 block body.

xxii. **E3C1 – $\boldsymbol{B1_m^{opt}}$ Excep3 Content (E3L1 * E3B1 bits):** Uses E3L1 * E3B1 bits to represent all the indices in the exception block $E31_m^{opt}$.

xxiii. **E1L2 – $B1_m^{opt}$ Excep1 Length (8 bits):** Uses 8 bits to represent the length of the exception block $E12_m^{opt}$. Explanation of the bit representation is same as that provided for the 'E1L' section in category-1 SA128 block body.

xxiv. **E1B2 – $B2_m^{opt}$ Excep1 Bits (8 bits):** Uses 8 bits to represent the bit required to represent each index value in exception block $E12_m^{opt}$. Explanation of the bit representation is same as that provided for the 'E1B' section in category-1 SA128 block body.

xxv. **E1C2 – $B2_m^{opt}$ Excep1 Content (E1L2 * E1B2 bits):** Uses E1L2 * E1B2 bits to represent all the indices in the exception block $E12_m^{opt}$.

xxvi. **E2L2 – $B2_m^{opt}$ Excep2 Length (8 bits):** Uses 8 bits to represent the length of the exception block $E22_m^{opt}$. Explanation of the bit representation is same as that provided for the 'E2L' section in category-1 SA128 block body.

xxvii. **E2B2 – $B2_m^{opt}$ Excep2 Bits (8 bits):** Uses 8 bits to represent the bit required to represent each index value in exception block $E22_m^{opt}$. Explanation of the bit representation is same as that provided for the 'E2B' section in category-1 SA128 block body.

xxviii. **E2C2 – $B2_m^{opt}$ Excep2 Content (E2L2 * E2B2 bits):** Uses E2L2 * E2B2 bits to represent all the indices in the exception block $E22_m^{opt}$.

xxix. **E3L2 – $B2_m^{opt}$ Excep3 Length (8 bits):** Uses 8 bits to represent the length of the exception block $E32_m^{opt}$. Explanation of the bit representation is same as that provided for the 'E3L' section in category-1 SA128 block body.

xxx. **E3B2 – $B2_m^{opt}$ Excep3 Bits (8 bits):** Uses 8 bits to represent the bit required to represent each index value in exception block $E32_m^{opt}$. Explanation of the bit representation is same as that provided for the 'E3B' section in category-1 SA128 block body.

xxxi. **E3C2 – $B2_m^{opt}$ Excep3 Content (E3L2 * E3B2 bits):** Uses E3L2 * E3B2 bits to represent all the indices in the exception block $E32_m^{opt}$.

4.2.3 Compression of Category 3 data types

The compression of category 3 datatypes take place in 6 sequential steps which are explained below:

Step 1 – Divide into blocks: In this step, all the data values belonging to the column being compressed is divided into blocks of 128 values. If there are N values in the column, the total number of blocks after division will be $\lfloor N/128 \rfloor$ values where each block contains 128 values each except the last block which contains $N\%128 + 1$ values (between 1 and 128). Refer to figure 4.1. Therefore, there cannot be an empty block which contains 0 number of values.

Step 2 – Create block copies: For each block B_m containing 128 values, where 1 <= m <= ⌈N/128⌉, we create three more copies of it and call them B_m', B_m'' and B_m''' respectively. In total, we have four identical blocks of 128 values each.

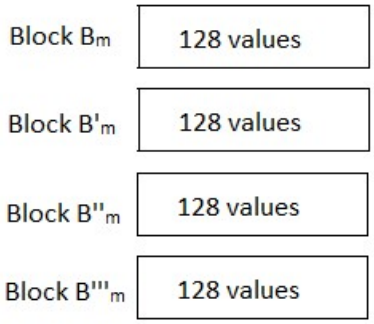

Figure 4.11: Illustration of Block Copy Creation Step in Category-3 Compression

Step 3 – Compress using category-1 compression algorithm: For blocks B_m, $B_m{}'$ and $B_m{}''$, follow steps 3 to 11 of the category-1 compression algorithm and we get the final encoded sub-blocks $B1_m$ and $B1_m$ along with their exception blocks $E11_m$, $E21_m$, $E31_m$ and $E12_m$, $E22_m$, $E32_m$ respectively.

Step 4 – Compress using XOR-based encoding variant: In this step, we compute a XOR between all contiguous values in block $B_m{}'''$. During XOR encoding, if there are NULL values in the block B_m''', we leave them as it is. During XOR encoding on block B_m''', for every non-null value v, we replace it with its XOR value d as $d = prev \wedge v$ where $prev$ is the previously scanned non-null value in the block (the values $prev$ and v do not necessarily need to be adjacent to each other as there can me any number of NULL values between them which remain unchanged). If the previously scanned non-null value $prev$ does not exist, then v is the first 32-bit or 64-bit value in $B_m{}'''$ (32-bit for real datatype and 64-bit for double precision datatype). Therefore, we keep the first non-NULL value v unchanged and compute XOR over the following elements. For example: [5.5, 5.6, NULL, 5.1] gets converted to [5.5, 5.562684646268e-310, NULL, 2.781342323134e-309]. Notice here that the first value remains unchanged because a non-null value does not exist prior to the first element. The first 32-bit or 64-bit value (for real and double precision datatypes respectively) in $B_m{}'''$ is left unchanged. We store the first non-NULL value f_m^{val} and its index position f_m^{idx} in $B_m{}'''$ We then remove f_m^{val} from block $B_m{}'''$. If the block contains all NULL values, then $f_m^{val} = 0$ and $f_m^{idx} = 128$.

Next, we scan through all the XOR-ed values in the block $B_m{}'''$ and find the total number of leading zeros l_m^{zeros} and trailing zeros t_m^{zeros} common to all the XOR-ed values in block $B_m{}'''$ in its 32-bit or 64-bit floating point representation (32-bit for real datatype and 64-bit for double precision datatype). Once we have the values $l_m^{zeros}$and t_m^{zeros}, we remove l_m^{zeros} number of leading zeros from all the non-NULL values and t_m^{zeros} number of trailing zeros in block $B_m{}'''$. For each non-NULL value, the bits remaining after

trimming the leading and trailing zeros are converted to an integer representation and stored in B_m''' in place of the original non-NULL value. The below diagram demonstrates this for real datatype values. In case of double precision datatypes, each value below will be represented using its 64-bit floating point representation instead of the 32-bit floating point representation for real type.

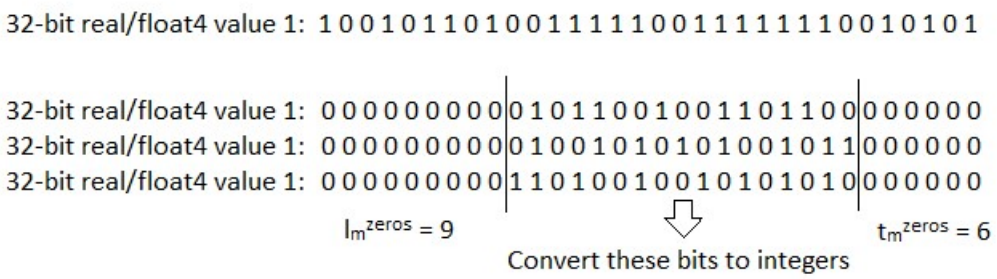

Figure 4.12: Illustration of XOR-Encoding Variant Step in Category-3 Compression

Next, we perform the operations from step 5 to step 10 of category 1 compression on block B_m''', which includes zig-zag encoding, frame of reference, run-length encoding, bit packing using modulo, packing NULL values, bit backing of exception blocks. The final block which we get after this step is $B3_m$ and exception blocks $E13_m$, $E23_m$ and $E33_m$.

Step 5 – Block selection: In this step, we choose between selecting the two sub-blocks $B1_m$, $B2_m$ received from step 3 and the block $B3_m$ received in step 4 respectively. (along with the exception blocks $E11_m$, $E21_m$, $E31_m$, $E12_m$, $E22_m$, $E32_m$, $E13_m$, $E23_m$, $E33_m$ respectively). We choose the block which takes the least number of bits to represent. Let the number of bits required to represent sub-blocks $B1_m$ and $B2_m$ be $S1_m$ and the number of bits required to represent block $B3_m$ be $S2_m$ respectively. $S1_m$ and $S2_m$ can be expressed as follows:

$$S1_m = N1_1 * b1_m + N1_2 * e11_m + N1_3 * e21_m + N1_4 * e31_m +$$
$$N2_1 * b2_m + N2_2 * e12_m + N2_3 * e22_m + N2_4 * e32_m$$

$$S2_m = N3_1 * b3_m + N3_2 * e13_m + N3_3 * e23_m + N3_4 * e33_m$$

where,

$N1_1$= Number of elements in block $B1_m$,

$b1_m$= Number of bits required to represent each value in $B1_m$,

$N1_2$ = Number of elements in exception block $E11_m$,

$e11_m$ = Number of bits required to represent each index value in $E11_m$,

$N1_3$ = Number of elements in exception block $E21_m$,

$e21_m$ = Number of bits required to represent each index value in $E21_m$,

$N1_4$ = Number of elements in exception block $E31_m$,

$e31_m$ = Number of bits required to represent each index value in $E31_m$,

$N2_1$= Number of elements in block $B2_m$,

$b2_m$= Number of bits required to represent each value in $B2_m$,

$N2_2$ = Number of elements in exception block $E12_m$,

$e12_m$ = Number of bits required to represent each index value in $E12_m$,

$N2_3$ = Number of elements in exception block $E22_m$,

$e22_m$ = Number of bits required to represent each index value in $E22_m$,

$N2_4$ = Number of elements in exception block $E32_m$,

$e32_m$ = Number of bits required to represent each index value in $E32_m$,

$N3_1$= Number of elements in block $B3_m$,

$b3_m$= Number of bits required to represent each value in $B3_m$,

$N3_2$ = Number of elements in exception block $E13_m$,

$e13_m$ = Number of bits required to represent each index value in $E13_m$,

$N3_3$ = Number of elements in exception block $E23_m$,

$e23_m$ = Number of bits required to represent each index value in $E23_m$,

$N3_4$ = Number of elements in exception block $E33_m$,

$e33_m$ = Number of bits required to represent each index value in $E33_m$,

We select between the two sub-blocks $B1_m$, $B2_m$ and the block $B3_m$ (and their exception blocks) which occupy the least number of bits $S_m = \min(S1_m, S2_m)$. If sub-blocks $B1_m$, $B2_m$ were selected, we call the selected sub-blocks $B1_m^{opt}$ and $B2_m^{opt}$ respectively and its exception blocks $E11_m^{opt}$, $E21_m^{opt}$ and $E31_m^{opt}$ respectively. We use two variables called $Enc1_m$ and $Enc2_m$ to store which encoding was used to encode the selected sub-blocks $B1_m^{opt}$ and $B2_m^{opt}$.

If the sub-blocks $B1_m^{opt}$ and $B2_m^{opt}$ were selected, then $Enc1_m$ and $Enc2_m$ have three possible values specifying the encoding used on that block:

i. $Enc1_m/Enc2_m = 0$: Uncompressed.
ii. $Enc1_m/Enc2_m = 1$: Delta encoding.
iii. $Enc1_m/Enc2_m = 2$: Delta of delta encoding.

If the block selected as the final block is $B3_m$, we call the selected block $B3_m^{opt}$. We use a variable $Enc3_m$ to represent which encoding was used in block $B3_m^{opt}$. $Enc3_m$ has only one possible value specifying the encoding used on that block:

i. $Enc3_m = 3$: XOR-encoding

Step 6 – Encode as category-3 SA128 block: We encode our selected block B_m^{opt} and exception blocks $E1_m^{opt}$, $E2_m^{opt}$ and $E3_m^{opt}$ by wrapping them within a category-3 SA128 block. The category-3 SA128 block consists of two parts:

i. A category-3 SA128 block header block – Contains metadata information about the block.
ii. A category-3 SA128 block body – Contains the data encoded within the block.

The components of a category-3 SA128 block header and block body depend on which blocks were selected in step 6. If the sub-blocks $B1_m^{opt}$ and $B2_m^{opt}$ were selected, then the components of a category-3 SA128 block header and block body are same as the

category-2 block header and block body. Refer to figure 4.10 for details regarding each component.

If the blocks $B3_m^{opt}$ was selected in step 5, then the components of a category-3 SA128 block header and block body can be given below:

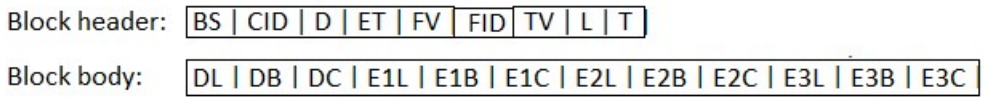

Figure 4.13: Components of SA128 Block for $Enc1m$ = 3 in Category-3 Compression

i. **BS – Block size in bytes (32 bits):** Stores the size of the category-3 block (block header and the block body) using 32-bits.
ii. **CID – Column ID (11 bits):** Uses 11 bits to represent the column ID of the column being compressed. Explanation of the bit representation is same as that provided for the 'CID' section in category-1 SA128 block header.
iii. **D – Datatype (4 bits):** Uses 4 bits to represent the data type on the column being compressed. Explanation of the bit representation is same as that provided for the 'D' section in category-1 SA128 block header.
iv. **ET – Encoding Type (2 bits):** Uses 2 bits to represent the encoding type $Enc3_m$ (which is 3 for XOR encoding) for sub-block $B3_m^{opt}$.
v. **FV – First Value (32 or 64 bits):** Uses 32 bits (for real data type) or 64-bits (for double precision datatype) to represent the first non-NULL value f_m^{val} in block $B3_m^{opt}$. If all values in $B3_m^{opt}$ are null, then $f_m^{val} = 0$ is stored.
vi. **FID – Index of first value (8 bits):** Uses 8 bits to represent the index of the first non-NULL value in block $B3_m^{opt}$. The largest possible index value in block $B3_m^{opt}$containing 128 values before step 4 is 127. Therefore, 8 bits are adequate to

represent the index of the first non-NULL value. If all values in $B3_m^{opt}$ are NULL, then a value of f_m^{idx} = 128 is stored.

vii. **TV – Translated Value (32 or 64 bits):** Uses 32 bits (for real data type) or 64-bits (for double precision datatype) to represent the translated value t_m received after the frame of reference with modulo operation at the end of step 4.

viii. **L – Leading Zeros (6 bits):** Uses 6 bits to represent the value l_m^{zeros} computed in step 4 representing the number of leading zeros. l_m^{zeros} can have a maximum value of 64 (for double precision data type) which can be adequately represented using 6 bits.

ix. **T – Trailing Zeros (6 bits):** Uses 6 bits to represent the value t_m^{zeros} computed in step 4 representing the number of trailing zeros. t_m^{zeros} can have a maximum value of 64 (for double precision data type) which can be adequately represented using 6 bits.

x. **DL – Data Length (8 bits):** Uses 8 bits to represent the length of the $B3_m^{opt}$ block. Explanation of the bit representation is same as that provided for the 'DL' section in category-1 SA128 block body.

xi. **DB – Data Bits (6 bits):** Uses 6 bits to represent the number of bits required to represent each data value in sub-block $B3_m^{opt}$. Since the maximum number of bits required to represent a data value is 64 bits (in the case of double precision datatype), 6 bits are adequate to represent the number of bits.

xii. **DC – Data Content (DL * DB bits):** Uses DL * DB bits to represent all the values in sub-block $B3_m^{opt}$.

xiii. **E1L – Excep1 Length (8 bits):** Uses 8 bits to represent the length of the exception block $E13_m^{opt}$. Explanation of the bit representation is same as that provided for the 'E1L' section in category-1 SA128 block body.

xiv. **E1B – Excep1 Bits (8 bits):** Uses 8 bits to represent the number of bits required to represent each index value in exception block $E13_m^{opt}$. Explanation of the bit representation is same as that provided for the 'E1B' section in category-1 SA128 block body.

xv. **E1C – Excep1 Content (E1L * E1B bits):** Uses E1L * E1B bits to represent all the indices in the exception block $E13_m^{opt}$.

xvi. **E2L – Excep2 Length (8 bits):** Uses 8 bits to represent the length of the exception block $E23_m^{opt}$. Explanation of the bit representation is same as that provided for the 'E2L' section in category-1 SA128 block body.

xvii. **E2B – Excep2 Bits (8 bits):** Uses 8 bits to represent the number of bits required to represent each index value in exception block $E23_m^{opt}$. Explanation of the bit representation is same as that provided for the 'E2B' section in category-1 SA128 block body.

xviii. **E2C – Excep2 Content (E2L * E2B bits):** Uses E2L * E2B bits to represent all the indices in the exception block $E23_m^{opt}$.

xix. **E3L – Excep3 Length (8 bits):** Uses 8 bits to represent the length of the exception block $E33_m^{op}$. Explanation of the bit representation is same as that provided for the 'E3L' section in category-1 SA128 block body.

xx. **E3B – Excep3 Bits (8 bits):** Uses 8 bits to represent the number of bits required to represent each index value in exception block $E33_m^{opt}$. Explanation of the bit representation is same as that provided for the 'E3B' section in category-1 SA128 block body.

xxi. **E3C – Excep3 Content (E3L * E3B bits):** Uses E3L * E3B bits to represent all the indices in the exception block $E33_m^{opt}$.

4.2.4 Compression of Category 4 data types

The compression of category-4 datatypes takes place in 3 sequential steps which are explained below:

Step 1 – Divide into blocks: In this step, all the data values belonging to the column being compressed is divided into blocks of 128 values. If there are N values in the column, the total number of blocks after division will be $\lfloor N/128 \rfloor$ values where each block contains 128 values each except the last block which contains $N\%128 + 1$ values (between 1 and 128). Therefore, there cannot be an empty block which contains 0 number of values.

Step 2 – Encode run-length, bit-packing and NULL packing: Execute steps 7 to 10 of category-1 compression algorithm on a single block copy B_m. We call our resultant block after all the above operations as B_m^{opt} and the resultant exception blocks are $E1_m^{opt}, E2_m^{opt}$ and $E3_m^{opt}$ respectively.

Step 3 – Encode as category-4 SA128 block: We encode our selected block B_m^{opt} and exception blocks $E1_m^{opt}$, $E2_m^{opt}$ and $E3_m^{opt}$ by wrapping them within a category-4 SA128 block. The category-4 SA128 block consists of two parts:

i. A category-4 SA128 block header – Contains metadata information about the block.
ii. A category-4 SA128 block body – Contains the data encoded within the block.

The components of a category-4 SA128 block header and block body can be given below:

Figure 4.14: Components of SA128 Block in Category-4 Compression

i. **BS – Block size in bytes (32 bits):** Stores the size of the category-4 block (block header and the block body) using 32-bits.

ii. **CID – Column ID (11 bits):** Uses 11 bits to represent the column ID of the column being compressed. Explanation of the bit representation is same as that provided for the 'CID' section in category-1 SA128 block header.

iii. **D – Datatype (4 bits):** Uses 4 bits to represent the data type on the column being compressed. Explanation of the bit representation is same as that provided for the 'D' section in category-1 SA128 block header.

iv. **DL – Data Length (8 bits):** Uses 8 bits to represent the length of the B_m^{opt} block. Explanation of the bit representation is same as that provided for the 'DL' section in category-1 SA128 block body.

v. **DB – Data Bits (1 bit):** Uses 1 bit to represent the number of bits required to represent each data value in sub-block B_m^{opt}. Since the maximum number of bits required to represent a data value of 'true' and 'false' is 1 bit (0 for false and 1 for true), 1 bit is adequate to represent the number of bits.

vi. **DC – Data Content (DL * DB bits):** Uses DL * DB bits to represent all the values in sub-block B_m^{opt}.

vii. **E1L – Excep1 Length (8 bits):** Uses 8 bits to represent the length of the exception block $E1_m^{opt}$. Explanation of the bit representation is same as that provided for the 'E1L' section in category-1 SA128 block body.

viii. **E1B – Excep1 Bits (8 bits):** Uses 8 bits to represent the number of bits required to represent each index value in exception block $E1_m^{opt}$. Explanation of the bit representation is same as that provided for the 'E2B' section in category-1 SA128 block body.

ix. **E1C – Excep1 Content (E1L * E1B bits):** Uses E1L * E1B bits to represent all the indices in the exception block $E1_m^{opt}$.

x. **E2L – Excep2 Length (8 bits):** Uses 8 bits to represent the length of the exception block $E2_m^{opt}$. Explanation of the bit representation is same as that provided for the 'E1L' section in category-1 SA128 block body.

xi. **E2B – Excep2 Bits (8 bits):** Uses 8 bits to represent the number of bits required to represent each index value in exception block $E2_m^{opt}$. Explanation of the bit representation is same as that provided for the 'E2B' section in category-1 SA128 block body.

xii. **E2C – Excep2 Content (E2L * E2B bits):** Uses E2L * E2B bits to represent all the indices in the exception block $E2_m^{opt}$.

xiii. **E3L – Excep3 Length (8 bits):** Uses 8 bits to represent the length of the exception block $E3_m^{opt}$. Explanation of the bit representation is same as that provided for the 'E1L' section in category-1 SA128 block body.

xiv. **E3B – Excep3 Bits (8 bits):** Uses 8 bits to represent the number of bits required to represent each index value in exception block $E3_m^{opt}$. Explanation of the bit representation is same as that provided for the 'E2B' section in category-1 SA128 block body.

xv. **E3C – Excep3 Content (E3L * E3B bits):** Uses E3L * E3B bits to represent all the indices in the exception block $E3_m^{opt}$.

4.2.5 Compression of Category 5 data types

The compression of category-5 datatypes takes place in 3 sequential steps which are explained below:

Step 1 – Create dictionary: Read all the string values one by one from the character or character varying type column being compressed. For each distinct string value, store them

onto a dictionary. The dictionary capacity, D_{cap} depends on the in-memory capacity of the system. The ideal size of the dictionary should be, $D_{cap} = 2^{8n} - 1$ where $n > 0$. In our implementation for SA128, we have implemented the dictionary capacity as $D_{cap} = 2^{16} - 1 = 65535$. In the dictionary, we map each unique string to an unsigned integer value starting between 0 and 65535 (D_{cap}).

Step 2 – LZ77 compression or LZ77 with dictionary compression: We can have three possible cases:

i. **Case 1:** If the number of distinct strings $N_{distinct} > D_{cap}$, this means that the dictionary is not large enough to represent all the strings in the column being compressed.

ii. **Case 2:** Let the size of all the column string values be C_{size} and let the size of the dictionary be D_{size}, Therefore,

$C_{size} = \sum_{i=0}^{N} L_{max} * 8$ (for character/char type) or,

$C_{size} = \sum_{i=0}^{N} L_{s_i} * 8 + 16$ (for character varying/varchar type)

$D_{size} = \sum_{i=0}^{D_{cap}} \log_2(D_{cap}) * L_{s_i} * 8$

where,

L_{max} = The max length n provided in the datatype definition,

L_{s_i} = Length of string at index position I,

For character/char type, the size of a string is the size of all its characters (included padded spaces at the end of the string to make it reach L_{max} length). Each character occupies 1 byte which when multiplied by 8 gives the size of the string in bits.

For character varying/varchar type, the size of a string is the size of all its characters + 16 extra bits.

In this case, if $C_{size} < D_{size}$, this means that there aren't too many repeated values in the column and storing them in a dictionary for dictionary compression only increases the storage requirement. This scenario is often encountered if the string column being compressed is part of a normalized table. When such a column is compressed, the low redundancy of the column makes it have near distinct values. This when combined with the size of mapped unsigned integers in the dictionary tends to increase the size of the resultant dictionary D_{size}.

iii. **Case 3:** $N_{distinct} \leq D_{cap}$ and $C_{size} \geq D_{size}$: This case arises when the number of distinct strings is less than the capacity of the dictionary and the size of the column is less than the dictionary size. The second condition is often satisfied when the column belongs to a denormalized table. In such columns, there may be a large range of repeated values which make it a good candidate for dictionary-based compression.

Step 2.1 – LZ77 compression: If we fall into one of the scenarios specified in case 1 or case 2, we directly apply LZ77 compression (Wesam Manassra, 2020) to all our column values. We do this by dividing all the string values into blocks of size 65535 (D_{cap}). We compress each block B_m of size 65535 where $1 <= m <= \lceil N/65535 \rceil$ using LZ77 compression.

Step 2.2 – LZ77 with dictionary compression: If we fall into the scenario specified in case 3, we represent each string in the column being compressed by its unsigned integer representation from the dictionary. We then compress this sequence of unsigned integers using steps 1 to 11 of category-1 compression. Let us call final compressed block we get after this operation as B_m^{opt}, its exception blocks as $E1_m^{opt}, E2_m^{opt} and\ E3_m^{opt}$ respectively, the translated value after frame of reference step as t_m and the encoding used to encode the block in the delta and delta encoding step as Enc_m. We then compress the strings in the

dictionary (which are less than 65535 in number) using LZ77 compression (Wesam Manassra, 2020).

Step 3 – Encode as SA128 block: Our column compressed in step 2 in wrapped around a category-5 SA128 block. A category-5 SA128 block has two types:

i. A category-5 SA128 string block – If our column was compressed using LZ77 compression in step 2.1 or if our dictionary strings were compressed using LZ77 with dictionary compression in step 2.2, we use a category-5 SA128 string block to encode the dictionary or column strings.
ii. A category-5 SA128 integer block – If our column was compressed using LZ77 with dictionary compression in step 2.2, we use a category-5 SA128 integer block which encodes the unsigned integers in the dictionary from step 2.2. This block is only used for columns compressed using LZ77 using dictionary compression in step 2.2.

The category-5 SA128 block consists of two parts:

i. A category-5 SA128 block header – Contains metadata information about the block.
ii. A category-5 SA128 block body – Contains the data encoded within the block.

The components of a category-5 SA128 string block can be given below:

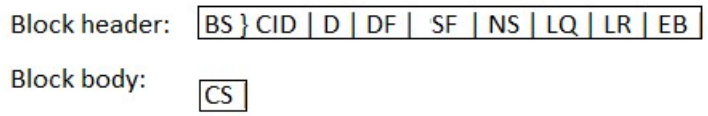

Figure 4.15: Components of SA128 String Block in Category-5 Compression

i. **BS – Block size in bytes (32 bits):** Stores the size of the category-5 block (block header and the block body) using 32-bits.
ii. **CID – Column ID (11 bits):** Uses 11 bits to represent the column ID of the column being compressed. Explanation of the bit representation is same as that provided for

the ‘CID’ section in category-1 SA128 block header.

iii. **D – Datatype (4 bits):** Uses 4 bits to represent the data type on the column being compressed. Explanation of the bit representation is same as that provided for the ‘D’ section in category-1 SA128 block header.

iv. **DF – Dictionary Flag (1 bit):** Uses 1 bit to represent whether the block is a category-4 SA128 string block or a category-4 SA128 integer block. We use bit ‘0’ to represent a category-5 SA128 integer block and bit ‘1’ to represent a category-5 SA128 string block.

v. **SF – Storage Flag (1 bit):** Uses 1 bit to represent whether to hold the dictionary in memory or not. If this bit is set to ‘0’, then the dictionary is discarded after encoding the category-5 SA128 string block. If this bit is set to ‘1’, then the dictionary is stored in memory until the next category-5 SA128 integer block has been encoded since the dictionary is required to extract the unsigned integer representations of the strings.

vi. **NS – Number of Strings (16 bits):** Uses 16 bits to store the number of strings being compressed as part of the category-5 SA128 string block. This value can range between 1 to 65535 (D_{cap}).

vii. **LQ – Length of Quotient (16 bits):** Uses 16 bits to represent the value L_q, where $L_q = \lfloor L_C / D_{cap} \rfloor$ and L_C = Length of the LZ77 compressed string and D_{cap} = Capacity of the dictionary (a value of 65535 in our case).

viii. **LR – Length of Remainder (16 bits):** Uses 16 bits to represent the value L_r, where $L_r = L_C \ \% \ D_{cap}$ and L_C = Length of the LZ77 compressed string and D_{cap} = Capacity of the dictionary (a value of 65535 in our case).

ix. **EB – Exempt Bits (3 bits):** Uses 3 bits to represent the additional bits appended to the binary representation of the compressed string (CS) to make the category-5 SA128 string block occupy a whole number of bytes, i.e., the number of bits added to the end of the binary representation of the block to make the entire length of the binary representation a multiple of 8 (1 byte). This is represented by the value $E_b =$

L_b % 8 where L_b = Length of the category-5 SA128 string block. Since $0 \leq E_b < 8$, we can represent this quantity using 3 bits.

x. **CS – Compressed String (LQ * D_{cap} (or 65535) + LR – EB bits):** Represents the LZ77 compressed string in bits.

The components of a category-5 SA128 integer block can be given below:

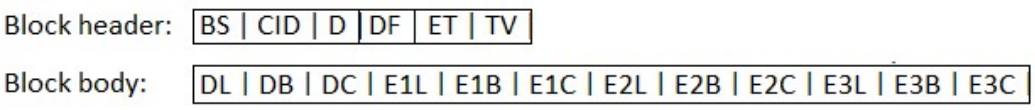

Figure 4.16: Components of SA128 Integer Block in Category-5 Compression

i. **BS – Block size in bytes (32 bits):** Stores the size of the category-5 block (block header and the block body) using 32-bits.

ii. **CID – Column ID (11 bits):** Uses 11 bits to represent the column ID of the column being compressed. Explanation of the bit representation is same as that provided for the 'CID' section in category-1 SA128 block header.

iii. **D – Datatype (4 bits):** Uses 4 bits to represent the data type on the column being compressed. Explanation of the bit representation is same as that provided for the 'D' section in category-1 SA128 block header.

xi. **DF – Dictionary Flag (1 bit):** Uses 1 bit to represent whether the block is a category-5 SA128 string block or a category-5 SA128 integer block. We use bit '0' to represent a category-5 SA128 integer block and bit '1' to represent a category-5 SA128 string block.

xii. **ET – Encoding Type (2 bits):** Uses 2 bits to represent the Enc_m encoding type value for the category-5 SA128 integer block. Explanation of the bit representation is same as that provided for the 'ET' section in category-1 SA128 block body.

xiii. **TV – Translated Value (16 bits):** Uses 16 bits to represent the translated value t_m

received after the frame of reference with modulo operation at the end of step 2.2. Explanation of the bit representation is same as that provided for the 'TV' section in category-1 SA128 block body.

xiv. **DL – Data Length (8 bits):** Uses 8 bits to represent the length of the B_m^{opt} block. Explanation of the bit representation is same as that provided for the 'DL' section in category-1 SA128 block body.

xv. **DB – Data Bits (4 bits):** Uses 4 bits to represent the number of bits required to represent each data value in sub-block B_m^{opt}. Since the maximum number of bits required to represent a data value is 16 bits (unsigned integers between 0 and 65535), 4 bits are adequate to represent the number of bits.

xvi. **DC – Data Content (DL * DB bits):** Uses DL * DB bits to represent all the values in sub-block $B3_m^{opt}$.

xvii. **E1L – Excep1 Length (8 bits):** Uses 8 bits to represent the length of the exception block $E1_m^{opt}$. Explanation of the bit representation is same as that provided for the 'E1L' section in category-1 SA128 block body.

xviii. **E1B – Excep1 Bits (8 bits):** Uses 8 bits to represent the number of bits required to represent each index value in exception block $E1_m^{opt}$. Explanation of the bit representation is same as that provided for the 'E1B' section in category-1 SA128 block body.

xix. **E1C – Excep1 Content (E1L * E1B bits):** Uses E1L * E1B bits to represent all the indices in the exception block $E1_m^{opt}$.

xx. **E2L – Excep2 Length (8 bits):** Uses 8 bits to represent the length of the exception block $E2_m^{opt}$. Explanation of the bit representation is same as that provided for the 'E1L' section in category-1 SA128 block body.

xxi. **E2B – Excep2 Bits (8 bits):** Uses 8 bits to represent the number of bits required to represent each index value in exception block $E2_m^{opt}$. Explanation of the bit

representation is same as that provided for the 'E2B' section in category-1 SA128 block body.

xxii. **E2C – Excep2 Content (E2L * E2B bits):** Uses E2L * E2B bits to represent all the indices in the exception block $E2_m^{opt}$.

xxiii. **E3L – Excep3 Length (8 bits):** Uses 8 bits to represent the length of the exception block $E3_m^{opt}$. Explanation of the bit representation is same as that provided for the 'E1L' section in category-1 SA128 block body.

xxiv. **E3B – Excep3 Bits (8 bits):** Uses 8 bits to represent the number of bits required to represent each index value in exception block $E3_m^{opt}$. Explanation of the bit representation is same as that provided for the 'E2B' section in category-1 SA128 block body.

xxv. **E3C – Excep3 Content (E3L * E3B bits):** Uses E3L * E3B bits to represent all the indices in the exception block $E3_m^{opt}$.

4.3 Compression Stage 2: rANS Entropy Encoding

In this stage we further compress all the SA128 blocks compressed at the end of Stage 1 using the ranged asymmetric numeral systems (rANS) entropy encoding technique. This stage takes place in four steps:

Step 1 – Preparing list of input symbols: In this step, we scan each byte of data (8 bits) one by one from the binary representation of all the compressed SA128 blocks. We convert each scanned byte of information into an ASCII character (between 0 to 255). This ASCII character is inserted into our list of input symbols (I) to be compressed using rANS. We also push each unique input symbol to our set of input symbols ($\sum$) to be compressed by rANS.

Step 2 – Computing probability distribution of input symbols: In this step, we compute the probability distribution $F(x)$, of the set of input symbols ($\sum$). For each symbol $x \in \sum$, we calculate the frequency, $f(x)$ of x such that $f(x) = \frac{(Number\ of\ times\ x\ occuers\ in\ I)}{(Total\ number\ of\ input\ symbols\ in\ \sum)}$. The probability distribution is saved in an internal database table and occupies 2048 bytes (2 KB) of storage. The probability distribution used to encode the symbols are needed during decompression to recover back the symbols.

Step 3 – Divide symbols into blocks and encode using rANS: In this step, we divide the list of input symbols 'I' into blocks containing 1024 input symbols each. For each symbol $x \in B_m$, where B_m is a block being encoded, we pass x and its frequency $f(x)$ as arguments to the rANS encoder (Fedor Glazov, 2020) to encode symbol x. After all symbols have been encoded, we receive the compressed data as a list of 32-bit integers $'O'$.

Step 4 – Encode as stage-2 SA128 block: For each block of 1024 symbols being encoded using rANS, we encapsulate it within a stage-2 SA128 block. The stage-2 SA128 block consists of two parts:

i. A stage-2 SA128 block header – Contains metadata information about the block.
ii. A stage-2 SA128 block body – Contains the data encoded within the block.

The components of a category-4 SA128 string block can be given below:

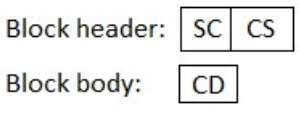

Figure 4.17: Components of Stage-2 SA128 Block in Stage-2 Compression

i. **SC – Symbol Count (16 bits):** Stores the number of symbols encoded by the stage-2 SA128 block. Since block B_m can have between 1 to 1024 symbols, therefore 16 bits are adequate to represent this quantity.

ii. **CS – Compressed Size (16 bits):** Uses 16 bits to represent the size of the compressed data using rANS in the block B_m.

iii. **CD – Compressed Data (CS bits):** Stores the final encoded data using CS number of bits. Convert each compressed data received as a 32-bit integer. Store the binary representation of the stage-2 SA128 block and store them onto our disk as our final compressed data for the block.

Chapter 5

SA128 DECOMPRESSION

The SA128 decompression is more straightforward compared to the SA128 compression. The decompression takes place in two stages:

i. Decompression Stage 1 – rANS entropy decoding stage: Uses an rANS variant of Asymmetric Numeral Systems (Duda, J., 2013) to decompress table data represented as a compressed integer and converts them to a list of ASCII symbols.

ii. Decompression Stage 2 – Column based decompression stage: Decompresses the list of symbols representing the encoded SA128 blocks for all columns in the table and retrieves the original data values for each column in the table with the supported datatype.

5.1 Decompression Stage 1: rANS Entropy Decoding

In this stage we decompress all the stage-2 SA128 blocks compressed at the end of section 4.3 using the ranged asymmetric numeral systems (rANS) entropy decoding technique. This stage takes place in four steps:

Step 1 – Read compressed blocks: In this step, we read each encoded block from section 4.3 in streamlined fashion. We read the first 16 bits signifying the Symbol Count (SC) and convert it into its integer representation. Read the next 16 bits signifying the Compressed Size (CS) of the data and convert it into its integer representation. Next, read CS number of bits from the stream and signifying the Compressed Data (CD) of the block. This process is repeated for all compressed blocks until there are no more bits to be read from the stream.

Step 2 – Decode stage-2 SA128 block: In this step, we fetch our probability distribution $F(x)$ for the encoded symbols saved in the internal database table during encoding of stage-2 SA128 block. Since the number of symbols encoded in the block is equal to SC, we perform an rANS decode (Fedor Glazov, 2020) operation SC number of times. In each

iteration, we pass the probability distribution $F(x)$ and the compressed data CD to the decoder so that the original symbols can be retrieved.

Step 3 – Convert symbols to bytes: In this step, each symbol is converted back from its ASCII representation to its byte representation (8 bits) and stored in memory for the next compression stage.

5.2 Decompression Stage 2: Column-based Decompression

In this stage, we sequentially decompress each column of our database tables from their encoded SA128 blocks for all 5 data type categories. The decompression stage 2 takes place using the following steps:

Step 1 – Read compressed blocks: In this step, we iteratively read all stage-1 SA128 blocks belonging to all of the 5 categories from our stream of decoded bits at the end of section 5.1. For each SA128 block, we read the first 32 bits signifying the overall Block Size (BS) and convert it into its integer representation. Next, read BS bits from the stream which contains the binary representation B_s of entire compressed SA128 block. Next, we read 11 bits from B_s signifying the Column ID (CID) which the compressed block belongs to and convert it into its integer representation. Next, we read 4 bits from B_s signifying the datatype of the column encoded by the block and convert it into its integer representation. Using the ID value in D, we can retrieve the datatype of the column represented by the block using the below table and determine the category which this SA128 block falls under:

Datatype ID (D)	**Datatype**
0	SmallInt/Int2
1	Integer/Int4
2	Bigint/Int8
3	Date
4	Timestamp
5	Timestamp with timezone/Timestamptz
6	Time
7	Time with timezone/Timez
8	Numeric/Decimal

9	Real/Float4
10	Double Precision/Float8
11	Boolean
12	Character/Char
13	Character Varying/Varchar

Table 5.1: Mapping of Datatype IDs to Supported Datatypes

The category of the SA128 block can be determined in the following way:

i. If the value D lies between 0 to 7, then our SA128 block is a category-1 SA128 block.

ii. If the value D is, then our SA128 block is a category-2 SA128 block.

iii. If the value D lies between 9 to 10, then our SA128 block is a category-3 SA128 block.

iv. If the value D is 11, then our SA128 block is a category-4 SA128 block.

v. If the value D lies between 12 to 13, then our SA128 block is a category-5 SA128 block.

Depending on which category the SA128 belongs to using the above logic, we follow a different decompression approach explained in the next sections (section 5.2.1 to 5.2.5). This way, we follow this process iteratively until there are no more bits to be read from the stream.

5.2.1 Decompression of Category 1 data types

If the SA128 block being decoded belongs to category 1, we follow the following steps to decode it:

Step 1 – Unpack Block Contents: Read 2 bits from the block stream B_s and convert it into its integer representation, which signifies the encoded value Enc_m of the current m^{th} data block B_m being decoded where $1 \leq m \leq \lfloor N/128 \rfloor + 1$ and N is the total number of blocks encoded using stage-1 compression. Next, we unpack the translated value (t_m) using the following rules:

i. If D is 0, read 16 bits from B_s and convert it into its integer representation which represents the translated value t_m for block B_m.

ii. If D is 1 or 3, read 32 bits from B_s and convert it into its integer representation which represents the translated value t_m for block B_m.

iii. If D is 2, 4, 5 or 6, read 64 bits from B_s and convert it into its integer representation which represents the translated value t_m for block B_m.

iv. If D is 7, read 96 bits from B_s and convert it into its integer representation which represents the translated value t_m for block B_m.

Read 8 bits from B_s and convert it into its integer representation which represents the block length N_1 of the compressed data in data block B_m. Lastly, read 8 bits from B_s and convert it into its integer representation which represents the block bits b_m of the compressed data in block B_m. Read the next $N_1 * b_m$ bits from B_s, where every b_m bits read is converted to its integer representation and stored in B_m.

Read the next 8 bits from B_s and convert it to its integer representation which signifies the length N_2 of the Exception 1 block $E1_m$ for block B_m. Read the next 8 bits from B_s and convert it to its integer representation which signifies the exception block bits $e1_m$ of the Exception 1 block $E1_m$ for block B_m. Next, read $N_2 * e1_m$ bits from B_s, where every $e1_m$ bits read is converted to its integer representation and stored in $E1_m$.

If there are bits available in B_s, read the next 8 bits from B_s and convert it to its integer representation which signifies the length N_3 of the Exception 2 block $E2_m$ for block B_m. Read the next 8 bits from B_s and convert it to its integer representation which signifies the exception block bits $e2_m$ of the Exception 2 block $E2_m$ for block B_m. Next, read $N_3 * e2_m$ bits from B_s, where every $e2_m$ bits read is converted to its integer representation and stored in $E2_m$.

If there are bits available in B_s, read the next 8 bits from B_s and convert it to its integer representation which signifies the length N_4 of the Exception 3 block $E3_m$ for block B_m. Read the next 8 bits from B_s and convert it to its integer representation which signifies the exception block bits $e3_m$ of the Exception 3 block $E3_m$ for block B_m. Next, read $N_4 * e3_m$ bits from B_s, where every $e3_m$ bits read is converted to its integer representation and stored in $E3_m$.

Step 2 – Unpack NULLs: In this step, for each value $i \in E3_m$, we replace the '0' value at index position i in B_m with NULL, i.e., $B_m[i] = NULL$.

Step 3 – Bit Unpacking using Modulo Technique: In this step, we find the maximum value $maxVal$ which can be represented using b_m bits. The value $maxVal$ can be represented as:

$$maxVal = 2^{b_m} - 1$$

For each value $i \in E2_m$. We perform the below transformations on block B_m:

i. $B_m[i] = B_m[i] * maxVal + B_m[i+1]$
ii. $Remove\ value\ at\ B_m[i+1]\ and\ shift\ elements\ left\ after\ index\ i+1\ by\ one.$
iii. $Decrease\ size\ of\ block\ N_1 by\ 1, i.e., N_1 = N_1 - 1.$

Step 4 – Decode Run-length: In this step, for each value $i \in E1_m$, we perform run-length decoding using the below transformations on block B_m:

i. $r = B_m[i]$
ii. Remove value at $B_m[i]\ and\ shift\ elements\ left\ after\ index\ i+1\ by\ one.$
iv. $Decrease\ size\ of\ block\ N_1 by\ 1, i.e., N_1 = N_1 - 1.$
v. Insert the value $B_m[i-1]\ into\ position\ i, r\ number\ of\ times.$
vi. $Increase\ size\ of\ block\ N_1 by\ r, i.e., N_1 = N_1 + r.$

Step 5 – Translate Frame of Reference: In this step, for each value at index position i in B_m, we convert the value by performing the below transformation on block B_m:

i. $B_m[i] = B_m[i] + t_m$

Step 6 – Zigzag decode: In this step, for each value at index position i in B_m, we convert the value by performing the below transformation on block B_m:

i. $B_m[i] = (B_m[i] >>> 1)\ \hat{}\ -(B_m[i]\ \&\ 1)$, where '>>>' is the non-arithmetic shift operation (0 – padding), '-' is the unary negation operation and '^' in the XOR operation. This transformation decodes positive numbers back to negative numbers using zigzag decoding.

Step 7 – Decode Delta and Delta of Delta Encoding: In this step, for each value at index position i in B_m, we perform delta or delta of delta decoding based on the encoding type Enc_m. The decoding technique used can be determined using the below table:

Enc_m	Decoding
0	Uncompressed
1	Delta decoding
2	Delta of Delta decoding

Table 5.2: Mapping of Enc_m values to Encoding Type

We perform the following transformations on our block based on the Enc_m values:

i. If $Enc_m = 0$: Keep B_m unchanged.

ii. If $Enc_m = 1$:

   a. If $prev$ exists, then $B_m[i] = prev - B_m[i]$, where $prev$ is the previous non-NULL value occurring before $B_m[i]$. There can be multiple number of NULL values between $prev$ and $B_m[i]$.
   b. If $prev$ does not exist, then $B_m[i]$ is left unchanged.

iii. If $Enc_m = 2$:

   a. If $prev$ and $pprev$ exist, then $B_m[i] = B_m[i] - pprev + (2 * prev)$, where $prev$ is the previous non-NULL value and pprev is the second previous non-NULL value occurring before $B_m[i]$ respectively.
   b. If $prev$ and $pprev$ do not exist, then $B_m[i]$ is left unchanged.

Step 8 – Convert block values from integer representation to datatype representation: In this step, for non-integer datatypes with D values 3 to 7, we convert them back to the original datatype format depending on the D value. For example:

i. Let s be an integer value which needs to be converted into a literal f of 'date' type, where s = '20211211'. This will be converted to f = '2021-12-11' (where the date value is stored in ISO 8601 format).

ii. Let s be an integer value which needs to be converted into a literal f of 'timestamp' type, where s = '2021121111553412313'. This will be converted to f = '2021-12-11 11:55:34.12313' (where the timestamp value is stored in ISO 8601 format).

iii. Let s be an integer value which needs to be converted into a literal f of 'timestamptz' type, where s = '-20211211115534800'. This will be converted to f = '2021-12-11 11:55:34 -8:00' (where the timestamptz value is stored in ISO 8601 format). Note that the negative sign for the integer value becomes the sign of the time zone in f.

iv. Let s be an integer value which needs to be converted into a literal f of 'time' type, where s = '11553412313'. This will be converted to f = '11:55:34.12313' (where the time value is stored in ISO 8601 format).

v. Let s be an integer value which needs to be converted into a literal f of 'timez' type, where s = '-115534123130800'. This will be converted to f = '11:55:34.12313-08:00' (where the time value is stored in ISO 8601 format). Note that the negative sign for the integer value becomes the sign of the time zone in f.

Step 9 – Store data block values in table column: Each data value in block B_m, contains between 128 values (except for the last block which contains N % 128 values where N is the number of values in the column). We store these values to our table column with $id = CID$ in the order in which they are decompressed. For the future blocks, the next batch of decoded values are appended to the table column after the previous block values till all the values of the column have been decompressed and stored back into the database.

5.2.2 Decompression of Category 2 data types

If the SA128 block being decoded belongs to category 2, we follow the following steps to decode it:

Step 1 – Unpack block contents: For each block B_m being decoded where $1 \leq m \leq \lfloor N/128 \rfloor + 1$ and N is the total number of blocks encoded using stage-1 compression, we have two sub-blocks, $B1_m$ and $B2_m$ respectively. These sub-blocks represent the part of the numeric/decimal type value before and after the decimal point respectively. We also calculate the bits per value for both the sub-blocks as $t1_{bits}$ and $t2_{bits}$, where $t1_{bits} = \log_2(10^{p-s} - 1), t2_{bits} = \log_2(10^s - 1)$. Here, p is the precision and s is the scale of the numeric/decimal type respectively.

Read 2 bits from the block stream B_s and convert it into its integer representation, which signifies the encoded value $Enc1_m$ of the sub-block $B1_m$. Read $t1_{bits}$ bits from the block stream B_s and convert it into its integer representation, which signifies the translated value $t1_m$ of the sub-block $B1_m$. Again, read 2 bits from the block stream B_s and convert it into its integer representation, which signifies the encoded value $Enc2_m$ of the sub-block $B2_m$. Read $t2_{bits}$ bits from the block stream B_s and convert it into its integer representation, which signifies the translated value $t2_m$ of the sub-block $B2_m$.

Read 8 bits from B_s and convert it into its integer representation which represents the block length $N1_1$ of the compressed data in sub-block $B1_m$. Read $\log_2\lceil t1_{bits} \rceil$ bits from B_s and convert it into its integer representation which represents the block bits $b1_m$ of the compressed data in sub-block $B1_m$. Read the next $N1_1 * b1_m$ bits from B_s, where every $b1_m$ bits read is converted to its integer representation and stored in $B1_m$. Read 8 bits from B_s and convert it into its integer representation which represents the block length $N2_1$ of the compressed data in sub-block $B2_m$. Read $\log_2\lceil t2_{bits} \rceil$ bits from B_s and convert it into its integer representation which represents the block bits $b2_m$ of the compressed data in

sub-block $B2_m$. Read the next $N2_1 * b2_m$ bits from B_s, where every $b2_m$ bits read is converted to its integer representation and stored in $B2_m$.

Read the next 8 bits from B_s and convert it to its integer representation which signifies the length $N1_2$ of the Exception 1 block $E11_m$ for sub-block $B1_m$. Read the next 8 bits from B_s and convert it to its integer representation which signifies the exception block bits $e11_m$ of the Exception 1 block $E11_m$ for sub-block $B1_m$. Next, read $N1_2 * e11_m$ bits from B_s, where every $e11_m$ bits read is converted to its integer representation and stored in $E11_m$. Read the next 8 bits from B_s and convert it to its integer representation which signifies the length $N2_2$ of the Exception 1 block $E12_m$ for sub-block $B2_m$. Read the next 8 bits from B_s and convert it to its integer representation which signifies the exception block bits $e12_m$ of the Exception 1 block $E12_m$ for sub-block $B2_m$. Next, read $N2_2 * e12_m$ bits from B_s, where every $e12_m$ bits read is converted to its integer representation and stored in $E12_m$.

Read the next 8 bits from B_s and convert it to its integer representation which signifies the length $N1_3$ of the Exception 2 block $E21_m$ for sub-block $B1_m$. Read the next 8 bits from B_s and convert it to its integer representation which signifies the exception block bits $e21_m$ of the Exception 2 block $E21_m$ for sub-block $B1_m$. Next, read $N1_3 * e21_m$ bits from B_s, where every $e21_m$ bits read is converted to its integer representation and stored in $E21_m$. Read the next 8 bits from B_s and convert it to its integer representation which signifies the length $N2_3$ of the Exception 2 block $E22_m$ for sub-block $B2_m$. Read the next 8 bits from B_s and convert it to its integer representation which signifies the exception block bits $e22_m$ of the Exception 2 block $E22_m$ for sub-block $B2_m$. Next, read $N2_3 * e22_m$ bits from B_s, where every $e22_m$ bits read is converted to its integer representation and stored in $E22_m$.

Read the next 8 bits from B_s and convert it to its integer representation which signifies the length $N1_4$ of the Exception 3 block $E31_m$ for sub-block $B1_m$. Read the next 8 bits from B_s and convert it to its integer representation which signifies the exception block bits $e31_m$ of the Exception 3 block $E31_m$ for sub-block $B1_m$. Next, read $N1_4 * e31_m$ bits from B_s, where every $e31_m$ bits read is converted to its integer representation and stored in $E31_m$. Read the next 8 bits from B_s and convert it to its integer representation which signifies the length $N2_4$ of the Exception 3 block $E32_m$ for sub-block $B2_m$. Read the next 8 bits from B_s and convert it to its integer representation which signifies the exception block bits $e32_m$ of the Exception 3 block $E32_m$ for sub-block $B2_m$. Next, read $N2_4 * e32_m$ bits from B_s, where every $e32_m$ bits read is converted to its integer representation and stored in $E32_m$.

Step 2 – Decompress sub-blocks using Catetogy-1 decompression: In this step, we decompress each of the two sub-blocks $B1_m$ and $B2_m$ along with their respective exception blocks $E11_m$, $E21_m$, $E31_m$ and $E12_m$, $E22_m$, $E32_m$ respectively using steps 2 to 7 of category-1 decompression (in section 5.2.1).

Step 3 – Retrieve numeric/decimal data values from sub-blocks: In this step, for each value at index position i in $B1_m$ and $B2_m$, we create our data block B_m which contains the

reconstructed decimal values by appending the corresponding data values in $B1_m$ and $B2_m$ separated by a decimal point. We do this by performing the following transformation:

i. $B_m[i] = concat(concat(str(B1_m), "."), str(B2_m).zfill(s))$, where concat(s1, s2) is the concatenation operation which concatenates two strings s1 and s2 provided as arguments to it, str(n) is a function which converts an integer n to string, zfill(s) is a function which prepends leading '0's to $B2_m$ so that $str(B2_m)$ can have a length equal to the scale s of the numeric/decimal type. The zfill(s) function is used to retrieve the lost 0s during conversion of the fractional part to integer during encoding.

Step 4 – Store data block values in table column: Each data value in block B_m, contains between 128 values (except for the last block which contains N % 128 values where N is the number of values in the column). We store these values to our table column with $id = CID$ in the order in which they are decompressed. For the future blocks, the next batch of decoded values are appended to the table column after the previous block values till all the values of the column have been decompressed and stored back into the database.

5.2.3 Decompression of Category 3 data types

If the SA128 block being decoded belongs to category 3, we follow the following steps to decode it:

Step 1 – Unpack block contents: Read 2 bits from the block stream B_s and convert it into its integer representation, which signifies the encoded value Enc_m of the current m^{th} data block B_m being decoded where $1 \leq m \leq \lfloor N/128 \rfloor + 1$ and N is the total number of blocks encoded using stage-1 compression.

We now have two scenarios based on the value of Enc_m:

i. **Scenario 1:** If Enc_m is 0, 1 or 2:

The block contains two sub-blocks $B1_m$ and $B2_m$. Let $Enc1_m$ *be* Enc_m.

The decompression process for scenario 1 can is shown using the flowchart below:

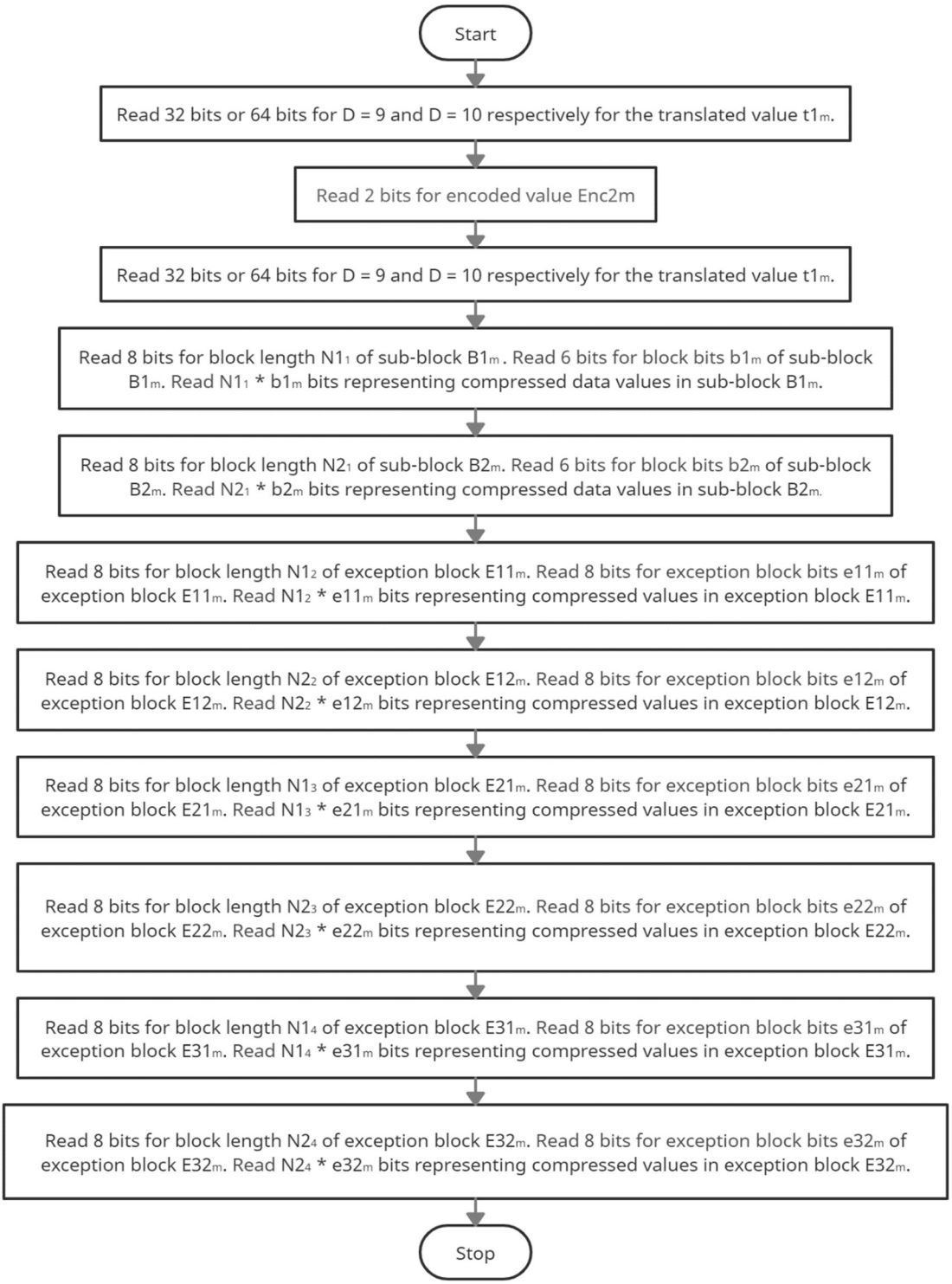

Figure 5.1: Steps for Scenario-1 of Category-3 Decompression

ii. **Scenario 2:** If Enc_m is 3:

Let $Enc3_m$ *be* Enc_m.

The decompression process for scenario 1 can is shown using the flowchart below:

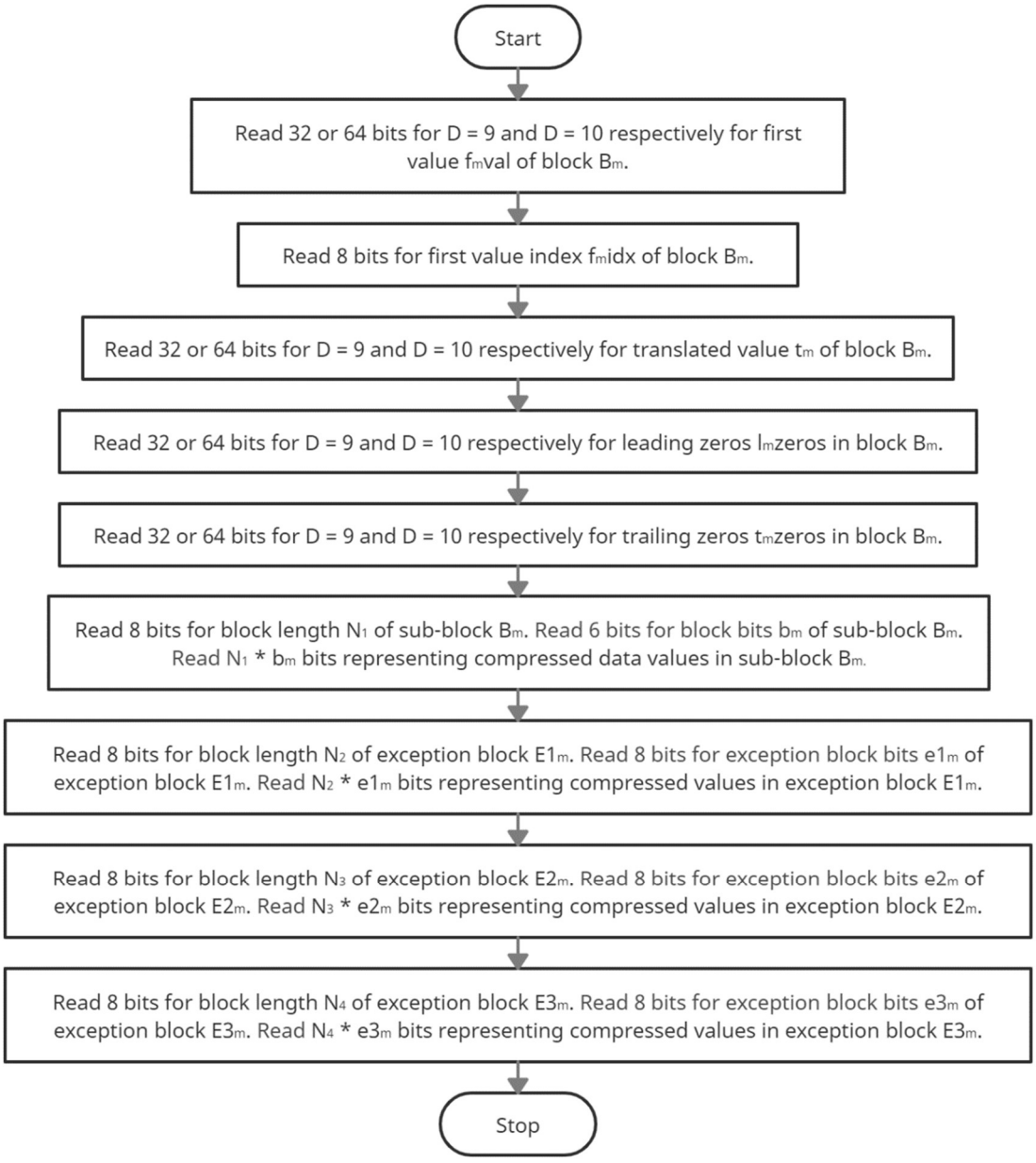

Figure 5.2: Steps for Scenario-2 of Category-3 Decompression

Step 2 – Scenario based decompression: Depending on which scenario was encountered in step 1, we perform the below actions:

i. **For scenario 1:** Decompress the sub-blocks $B1_m$ and $B2_m$ along with their exception blocks using steps 2 and 3 of category-2 decompression (in section 5.2.2) and get the final decoded data block B_m.

ii. **For scenario 2:** For each data value at index i in B_m, perform the XOR-decoding using the following transformations on the data block B_m:

   a. Convert $B_m[i]\ to\ its\ binary\ representation\ using\ at\ most\ (M - (l_m^{zeros} + t_m^{zeros}))$, where M = 32 if D = 9 and M = 64 if D = 10.
   b. Prepend l_m^{zeros} number of zeros to the binary representation of $B_m[i]$.
   c. Append t_m^{zeros} number of zeros to the binary representation of $B_m[i]$.
   d. Convert $B_m[i]$ to its 32-bit or 64-bit floating point representation for D = 9 or D = 10 respectively.
   e. Insert value f_m^{val} at index position $f_m^{idx}\ in\ B_m$.
   f. Decode using XOR by the transformation, $B_m[i] = B_m[i]\ \hat{}\ B_m[i-1]$ where $i > 0$ and '^' is the XOR operator.

   After XOR-decoding, we get our final decoded data block B_m.

Step 3 – Store data block values in table column: Each data value in block B_m, contains between 128 values (except for the last block which contains N % 128 values where N is the number of values in the column). We store these values to our table column with $id = CID$ in the order in which they are decompressed. For the future blocks, the next batch of decoded values are appended to the table column after the previous block values till all the values of the column have been decompressed and stored back into the database.

5.2.4 Decompression of Category 4 data types

If the SA128 block being decoded belongs to category 4, we follow the following steps to decode it:

Step 1 – Unpack Block Contents: Read 2 bits from the block stream B_s and convert it into its integer representation, which signifies the encoded value Enc_m of the current m^{th} data block B_m being decoded where $1 \leq m \leq \lfloor N/128 \rfloor + 1$ and N is the total number of blocks encoded using stage-1 compression.

Read 8 bits from B_s and convert it into its integer representation which represents the block length N_1 of the compressed data in data block B_m. Read 1 bit from B_s and convert it into its integer representation which represents the block bits b_m of the compressed data in block B_m. Read the next $N_1 * b_m$ bits from B_s, where every b_m bits read is converted to its integer representation and stored in B_m.

Read the next 8 bits from B_s and convert it to its integer representation which signifies the length N_2 of the Exception 1 block $E1_m$ for block B_m. Read the next 8 bits from B_s and convert it to its integer representation which signifies the exception block bits $e1_m$ of the Exception 1 block $E1_m$ for block B_m. Next, read $N_2 * e1_m$ bits from B_s, where every $e1_m$ bits read is converted to its integer representation and stored in $E1_m$.

If there are bits available in B_s, read the next 8 bits from B_s and convert it to its integer representation which signifies the length N_3 of the Exception 2 block $E2_m$ for block B_m. Read the next 8 bits from B_s and convert it to its integer representation which signifies the exception block bits $e2_m$ of the Exception 2 block $E2_m$ for block B_m. Next, read $N_3 * e2_m$ bits from B_s, where every $e2_m$ bits read is converted to its integer representation and stored in $E2_m$.

If there are bits available in B_s, read the next 8 bits from B_s and convert it to its integer representation which signifies the length N_4 of the Exception 3 block $E3_m$ for block B_m. Read the next 8 bits from B_s and convert it to its integer representation which signifies the exception block bits $e3_m$ of the Exception 3 block $E3_m$ for block B_m. Next, read $N_4 * e3_m$ bits from B_s, where every $e3_m$ bits read is converted to its integer representation and stored in $E3_m$.

Step 2 – Decompress using category-1 decompression: Follow steps 2 to 4 of category-1 decompression (in section 5.2.1) which gives the final data block B_m.

Step 3 – Convert block values to Boolean: For each data value with index i in B_m, we perform the following transformations on the data block:

i. If $(B_m[i] == 0)$, assign $B_m[i] = 'f'$.
ii. If $(B_m[i] == 1)$, assign $B_m[i] = 't'$.

Step 4 – Store data block values in table column: Each data value in block B_m, contains between 128 values (except for the last block which contains N % 128 values where N is the number of values in the column). We store these values to our table column with $id = CID$ in the order in which they are decompressed. For the future blocks, the next batch of decoded values are appended to the table column after the previous block values till all the values of the column have been decompressed and stored back into the database.

5.2.5 Decompression of Category 5 data types

If the SA128 block being decoded belongs to category 5, we follow the following steps to decode it:

Step 1 – Unpack Block Contents: Read 1 bit from the block stream B_s and convert it into its integer representation, which signifies the dictionary flag (DF) of the current m^{th} data

block B_m being decoded where $1 \leq m \leq \lfloor N/128 \rfloor + 1$ and N is the total number of blocks encoded using stage-1 compression. If DF = 0, then our block is a category-5 SA128 integer block. If DF = 1, then our block is a category-5 SA128 string block.

Depending on our DF value, we have two scenarios for decoding our category-5 SA128 block:

i. Scenario 1 – For DF = 1:

Read 1 bit from the block stream B_s and convert it into its integer representation, which represents the storage flag (SF) flag. If SF = 0, the strings being decompressed will written disk after the block has been decoded using LZ77 decompression (Wesam Manassra, 2020). If SF = 1. The decoded strings are not written onto the disk but are held in memory as a dictionary and are used for decompressing all the following category-5 SA128 integer blocks since the dictionary is required for LZ77 with dictionary decompression in category-5 SA128 integer blocks.

Read 16 bits from the block stream B_s and convert it into its integer representation, which represents the number of strings (NS) compressed in the block B_m.

Read 16 bits from the block stream B_s and convert it into its integer representation, which represents the length of quotient value (L_q) where $L_q = \lfloor L_C/D_{cap} \rfloor$, L_C is the length of the LZ77 compressed data and $D_{cap} = 65535$.

Read 16 bits from the block stream B_s and convert it into its integer representation, which represents the length of remainder value (L_r) where $L_r = L_C \,\%\, D_{cap}$, L_C is the length of the LZ77 compressed data and $D_{cap} = 65535$.

Read 3 bits from the block stream B_s and convert it into its integer representation, which represents the number of exempt bits E_b, where $E_b = L_b \,\%\, 8$, L_b = Length of the category-4 SA128 string block.

Read $C_b = L_q * 65535 + L_r - E_b$ bits from the block stream B_s and convert it into its integer representation, which represents the compressed string data.

ii. Scenario 1 – For DF = 0:

Read 2 bits from the block stream B_s and convert it into its integer representation, which signifies the encoded value Enc_m of our block B_m.

Read 16 bits from B_s and convert it into its integer representation which represents the translated value t_m for block B_m.

Read 8 bits from B_s and convert it into its integer representation which represents the block length N_1 of the compressed data in data block B_m. Next, read 4 bits from B_s

and convert it into its integer representation which represents the block bits b_m of the compressed data in block B_m. Read the next $N_1 * b_m$ bits from B_s, where every b_m bits read is converted to its integer representation and stored in B_m.

Read the next 8 bits from B_s and convert it to its integer representation which signifies the length N_2 of the Exception 1 block $E1_m$ for block B_m. Read the next 8 bits from B_s and convert it to its integer representation which signifies the exception block bits $e1_m$ of the Exception 1 block $E1_m$ for block B_m. Next, read $N_2 * e1_m$ bits from B_s, where every $e1_m$ bits read is converted to its integer representation and stored in $E1_m$.

If there are bits available in B_s, read the next 8 bits from B_s and convert it to its integer representation which signifies the length N_3 of the Exception 2 block $E2_m$ for block B_m. Read the next 8 bits from B_s and convert it to its integer representation which signifies the exception block bits $e2_m$ of the Exception 2 block $E2_m$ for block B_m. Next, read $N_3 * e2_m$ bits from B_s, where every $e2_m$ bits read is converted to its integer representation and stored in $E2_m$.

If there are bits available in B_s, read the next 8 bits from B_s and convert it to its integer representation which signifies the length N_4 of the Exception 3 block $E3_m$ for block B_m. Read the next 8 bits from B_s and convert it to its integer representation which signifies the exception block bits $e3_m$ of the Exception 3 block $E3_m$ for block B_m. Next, read $N_4 * e3_m$ bits from B_s, where every $e3_m$ bits read is converted to its integer representation and stored in $E3_m$.

Step 2 – Decode using LZ77 or LZ77 with dictionary decompression: Depending on which scenario from step 1 our category-5 SA128 block falls under, we either execute either LZ77 decompression or LZ77 with dictionary decompression:

i. For scenario 1:

If DF = 1 and SF = 0, the compressed data bits C_b are decoded using LZ77 decompression and stored in our final data block B_m^{final}.

If DF = 1 and SF = 1, the compressed data bits C_b are decoded using LZ77 decompression and are held in memory within a dictionary $Dict_m$ where the dictionary holds all unique strings mapped to an unsigned integer which is incremented from 0 to 65535.

ii. For scenario 2:

If DF = 0, then decompress the block B_m and its exception blocks using steps 2 to 7 of category-1 decompression (in section 5.2.1) to get the final data block B_m^{final}.

Replace each value at index i in B_m^{final} with its mapped string value in the dictionary.

$$B_m^{final}[i] = Dict[B_m^{final}[i]]$$

Step 3 – Store data block values in table column: Each data value in blockB_m^{final}, contains between 128 values (except for the last block which contains N % 128 values where N is the number of values in the column). We store these values to our table column with $id = CID$ in the order in which they are decompressed. For the future blocks, the next batch of decoded values are appended to the table column after the previous block values till all the values of the column have been decompressed and stored back into the database.

Chapter 6

EXPERIMENTS AND RESULTS

In this section, we go over the system configuration of the experimental setup used for collecting results and conducting experiments.

6.1 System Configurations

The system configuration used for collecting results for SA128 compression are summarized in the table below:

System Manufacturer	LENOVO
System Model	81N7
Operating System	Windows 10 Home Single Language 64-bit (10.0, Build 19042)
Memory	8192GB RAM
Processor	Intel® Core™ i5-8265U CPU @ 1.60 GHz

Table 6.1: System Configurations

6.2 Languages and Software Used

The programming languages and software libraries used for implementation of SA128 compressor and decompressor and conducting experiments are provided below:

1. Python 3.9.0: For developing SA128 compressor (stage-1 + stage-2) and SA128 decompressor (stage-1 and stage-2).
2. Shell: For executing CLI commands for executing the SA128 compressor and decompressor on data files.
3. Microsoft Visual Studio 2019: For building the TPC-DS version v2.13.0rc1 source code to generate benchmark data sets.

4. Libraries: LZ77-Compressor (Wesam Manassra, 2020), Zstd 1.4.8.1 (Sergey Dryabzhinsky and Anton Shuk, 2020), Snappy 1.1.3 (Snappy 2015), Lzma 9.38 (Igor Pavlov, 2015), Zlib 1.2.11 (Mark Adler, 2017), LZ4 3.1.3 (Jonathan Underwood, 2021), Brotli 1.0.9 (Brotli, 2020), Python-rANSCoder (Fedor Glazov, 2020).

6.3 Results

We divide the result section in two parts based on the datasets used for testing compression results. The first dataset used for result collection is a 1 GB TPC-DS Benchmark Dataset and the second dataset used for result collection is a generated dataset discussed in chapter 2.

6.3.1 Results on 1GB TPC-DS Benchmark Dataset (Dataset – 1)

For Dataset – 1, we segregate results based into the following categories based on size of the tables in the dataset:

1. Large sized tables (> 100 MB): These consist of tables such as catalog_sales, inventory, store_sales and web_sales.
2. Medium sized tables (1 MB – 100 MB): There consist of tables such as catalog_page, catalog_returns, customer, customer_address, customer_demographics, date_dim, household_demographics, item, store_returns, time_dim and web_returns.
3. Small sized tables (< 1 MB): These consist of tables such as call_center, income_band, promotion, reason, ship_mode, store, warehouse, web_page and web_site.

6.3.2 Results for Large Tables

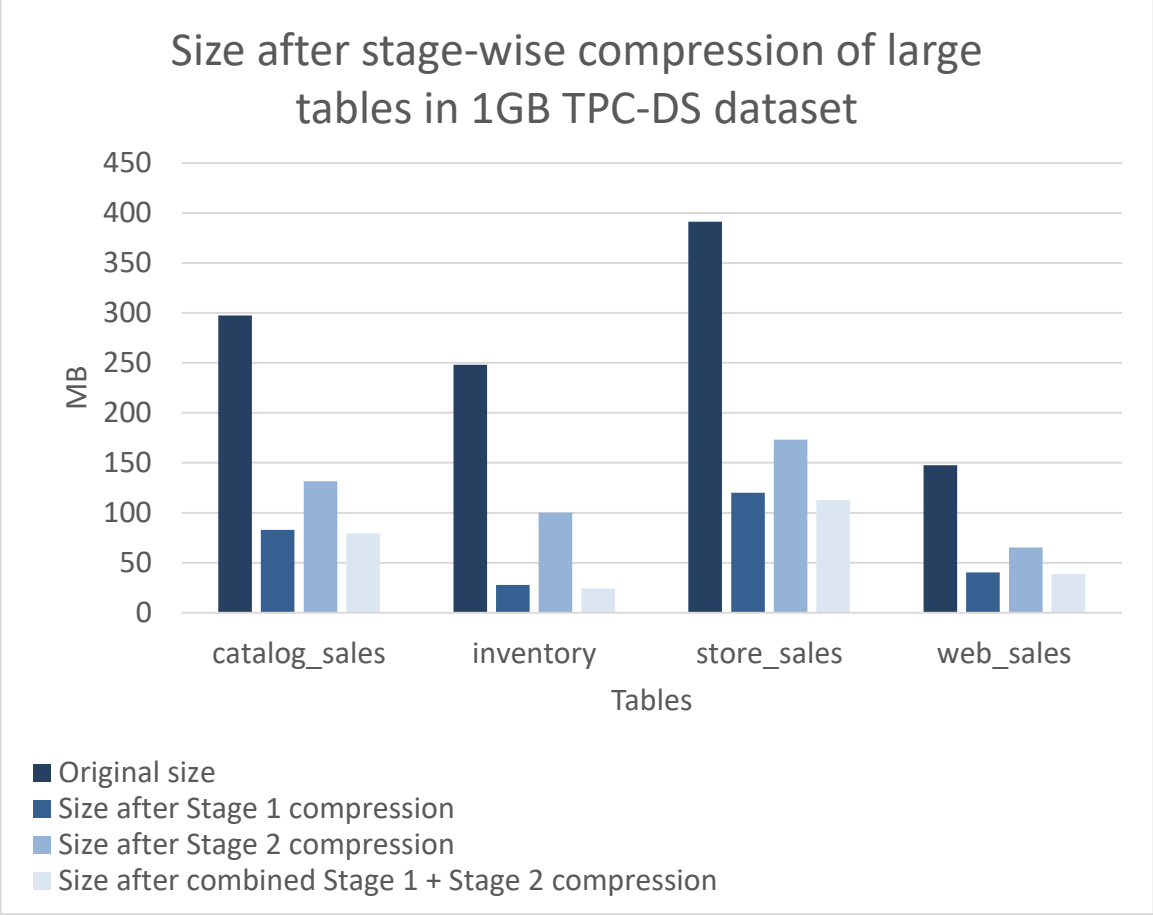

Figure 6.1: Comparative Analysis of Space Reduction Achieved on Large Tables in 1 GB TPC-DS Benchmark Dataset Using SA128 Compression for different stages of compression, i.e., stage-1 compression, stage-2 compression and combined stage-1 + stage-2 compression. The above graph shows that we achieved the highest compression ratio for all the 4 large tables by combining stage-1 and stage-2 compression in SA128 compression compared to compressing our tables using stage-1 compression and stage-2 compression separately. The space reduction achieved are 77.34% for catalog_sales, 90.25% for inventory, 71.19% for store_sales, 73.82% for web_sales. Refer to appendix A, B and C for more details regarding the data collected.

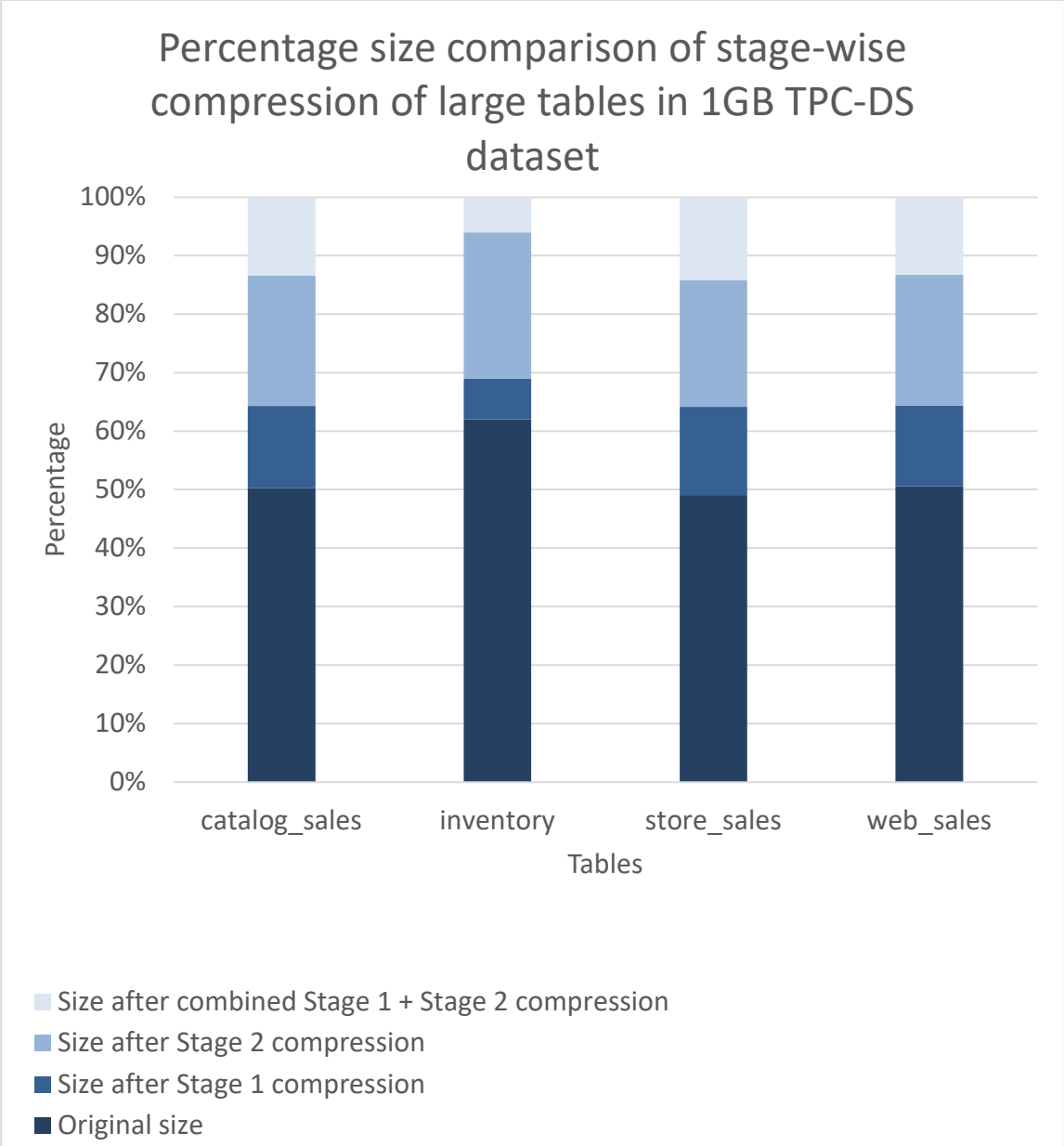

Figure 6.2: Percentage Comparison of Space Reduction Achieved on Each Large Table in 1 GB TPC-DS Benchmark Dataset Using SA128 Compression for different stages of compression, i.e., stage-1 compression, stage-2 compression and combined stage-1 + stage-2 compression. Refer to appendix A, B and C for more details regarding the data collected.

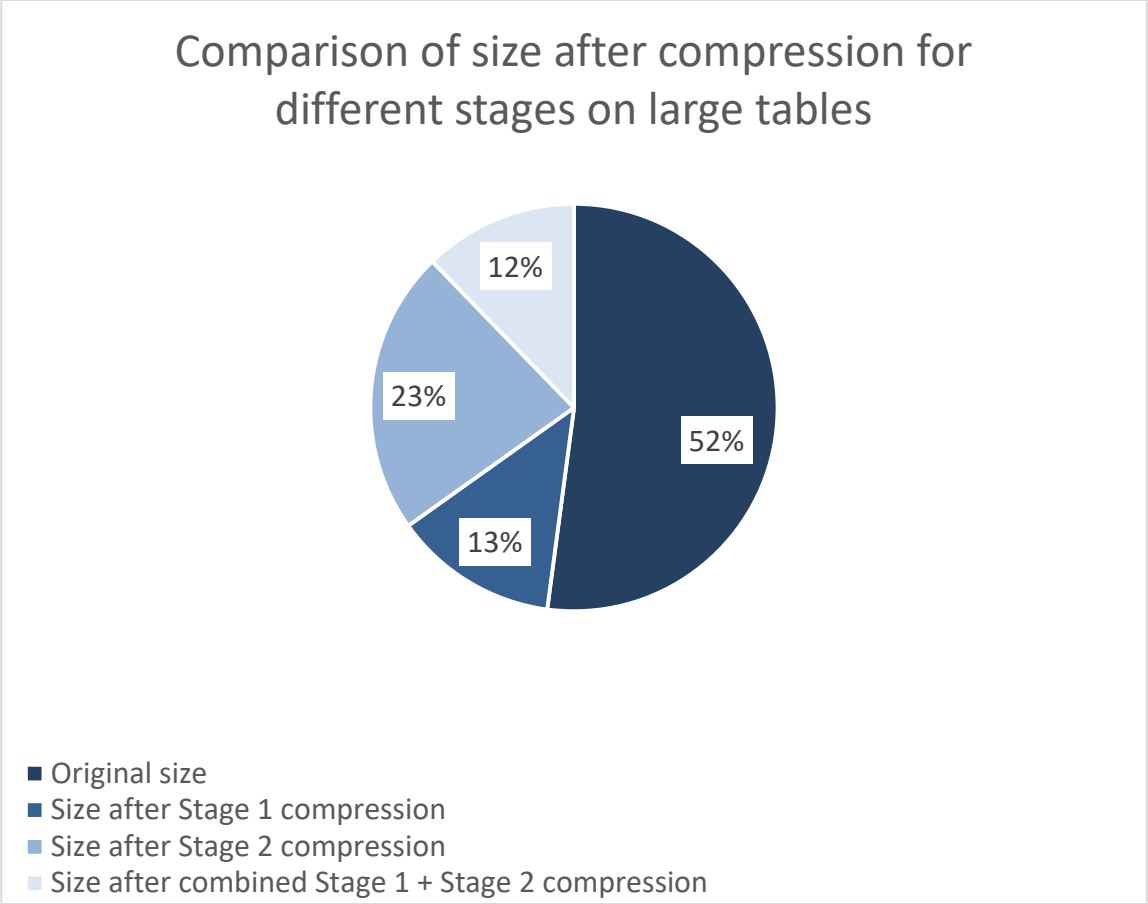

Figure 6.3: Combined Space Reduction Achieved for All Large Tables in 1 GB TPC-DS Benchmark Dataset Using SA128 Compression for different stages of compression, i.e., stage-1 compression, stage-2 compression and combined stage-1 + stage-2 compression. The above graph shows that for large tables, the SA128 algorithm achieved space savings up to 77% using both stage-1 and stage-2 compression, 75% using only stage-1 compression and 55.76% using only stage-2 compression. Refer to Appendix A, B and C for more details regarding the data collected.

6.3.3 Results for Medium Tables

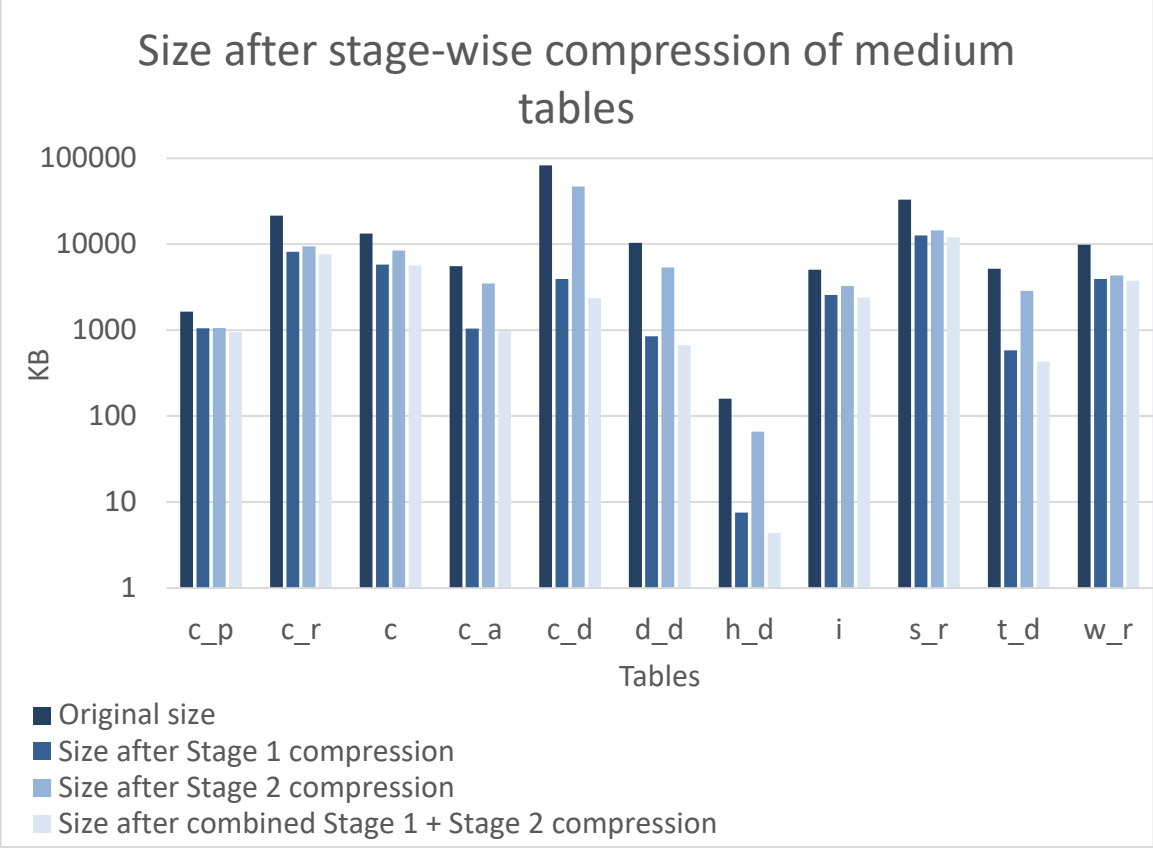

Figure 6.4: Comparative Analysis of Space Reduction Achieved on Medium Sized Tables in 1 GB TPC-DS Benchmark Dataset Using SA128 Compression for different stages of compression, i.e., stage-1 compression, stage-2 compression and combined stage-1 + stage-2 compression. The above graph shows that we achieved the highest compression ratio for all the 11 tables by combining stage-1 and stage-2 compression in SA128 compression compared to compressing our tables using stage-1 compression and stage-2 compression separately. The space reduction achieved are 42.45% for catalog_page, 64.43% for catlog_returns, 57.66% for customer, 82.44% for customer_address, 97.14% for customer_demographics, 93.59% for date_dim, 97.26% for household_demographics, 52.76% for item, 63.68% for store_returns, 91.7% for time_dim and 61.99% for web_returns. Refer to appendix A, B and C for more details regarding the data collected.

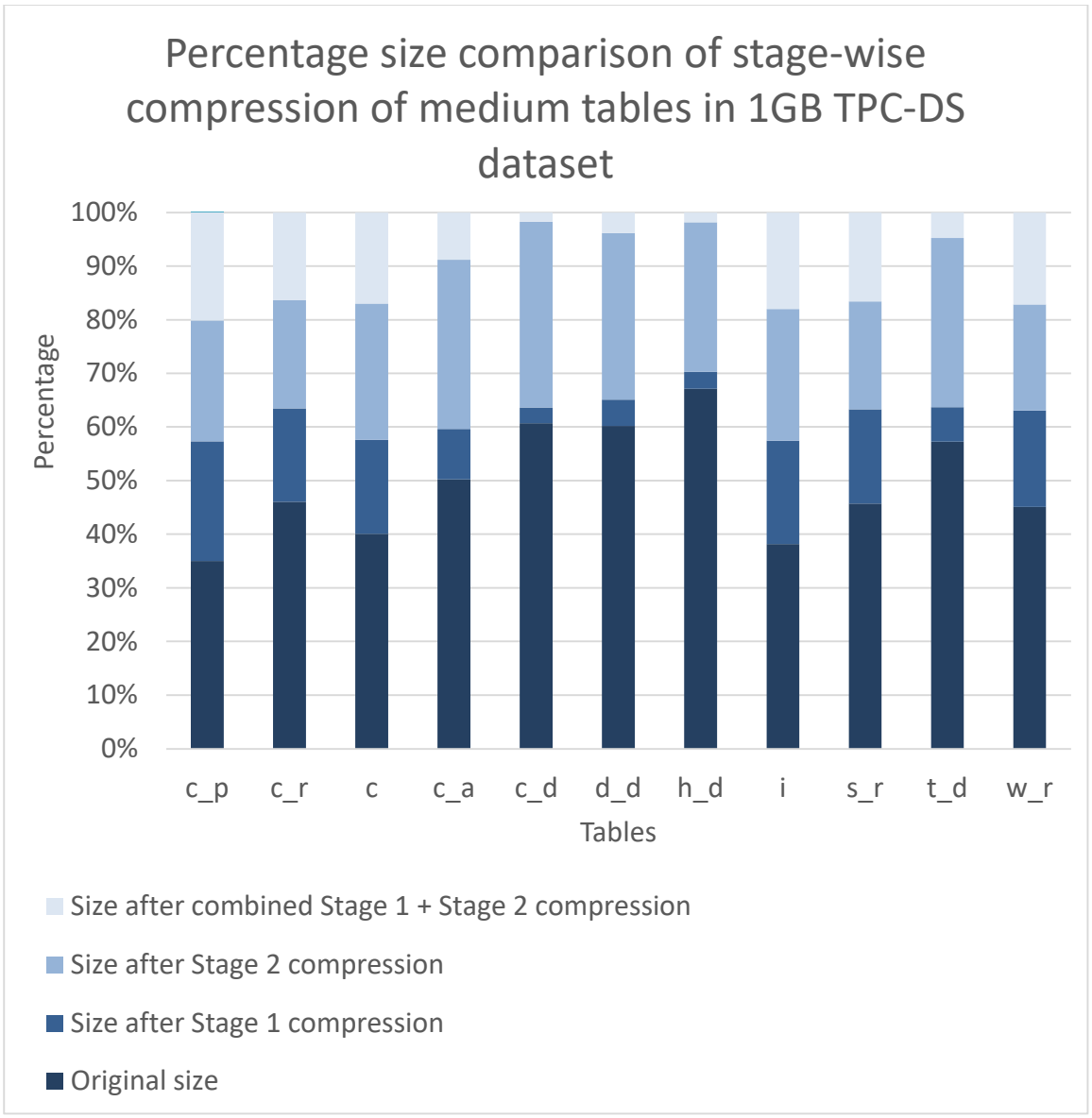

Figure 6.5: Percentage Comparison of Space Reduction Achieved on Each Medium Sized Table in 1 GB TPC-DS Benchmark Dataset Using SA128 Compression for different stages of compression, i.e., stage-1 compression, stage-2 compression and combined stage-1 + stage-2 compression. Refer to appendix A, B and C for more details regarding the data collected.

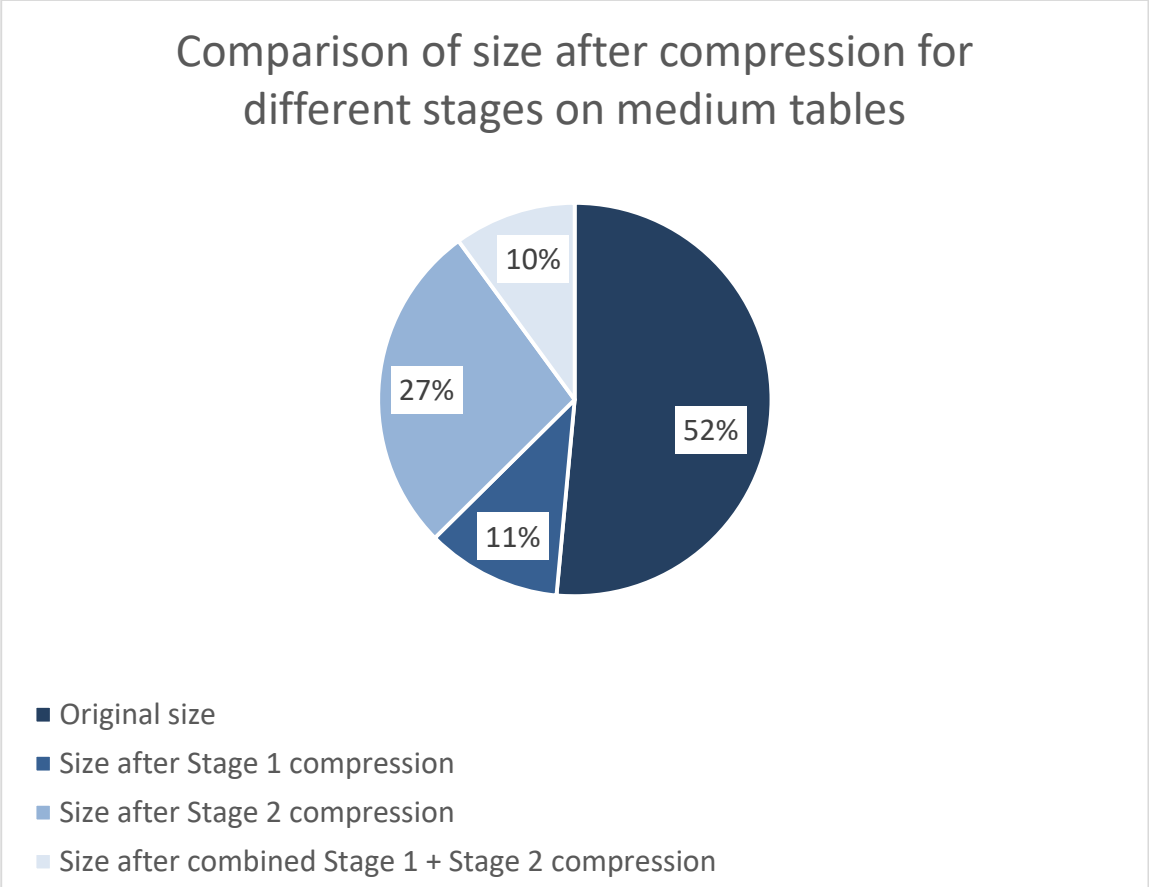

Figure 6.6: Combined Space Reduction Achieved for All Medium Sized Tables in 1 GB TPC-DS Benchmark Dataset Using SA128 Compression for different stages of compression, i.e., stage-1 compression, stage-2 compression and combined stage-1 + stage-2 compression. The above graph shows that for large tables, the SA128 algorithm achieved space savings up to 80.76% using both stage-1 and stage-2 compression, 79.8% using only stage-1 compression and 48.07% using only stage-2 compression. Refer to Appendix A, B and C for more details regarding the data collected.

6.3.4 Results for Small Tables

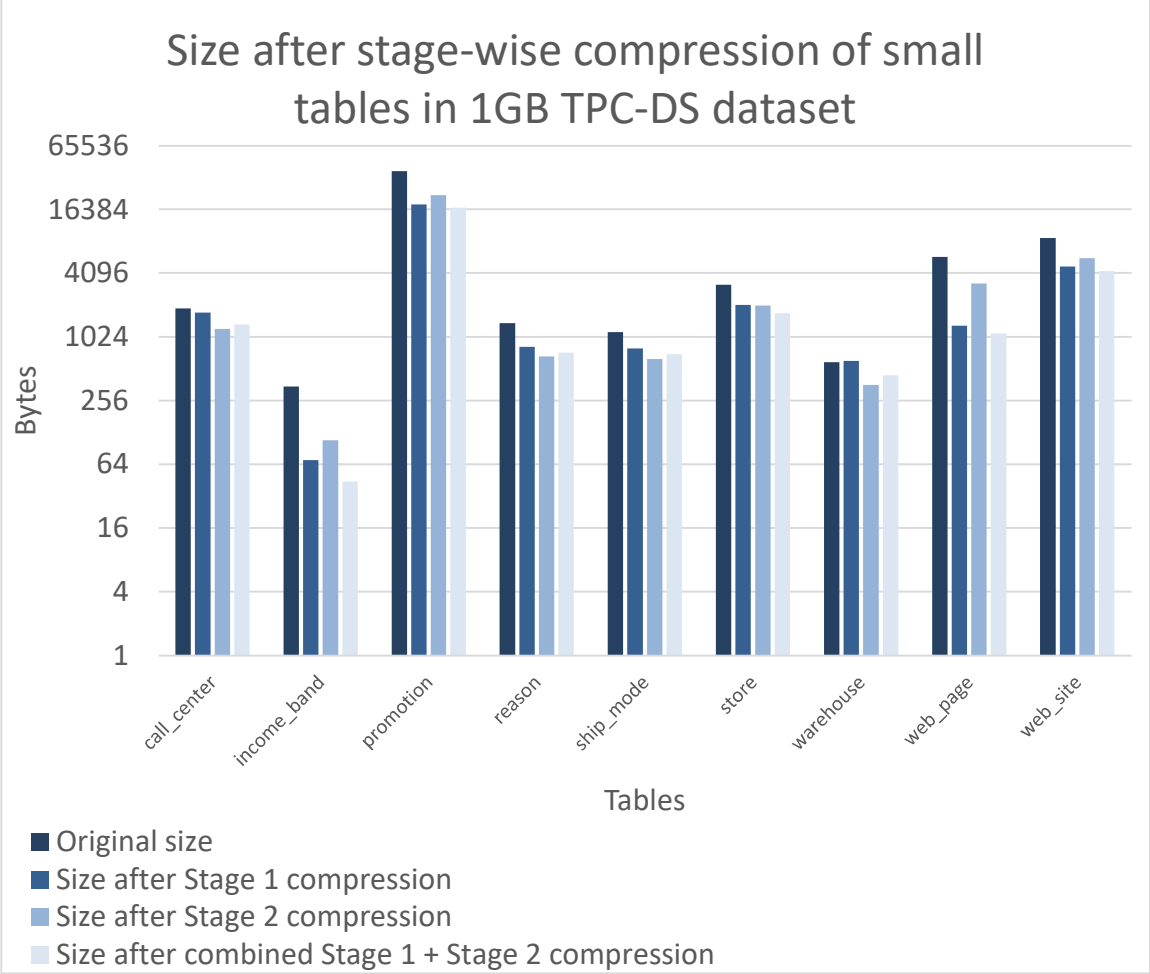

Figure 6.7: Comparative Analysis of Space Reduction Achieved on Small Sized Tables in 1 GB TPC-DS Benchmark Dataset Using SA128 Compression for different stages of compression, i.e., stage-1 compression, stage-2 compression and combined stage-1 + stage-2 compression. The above graph shows that for 5 tables we achieved the best compression ratio by combining stage-1 and stage-2 compression in SA128 whereas for 4 tables, we achieved the best compression ratio using only stage-2 compression in SA128. This is because for small tables, there is an extra overhead in storing the block header information during stage-1 compression which results in increasing the overall space than reducing it. Refer to appendix A, B and C for more details regarding the data collected.

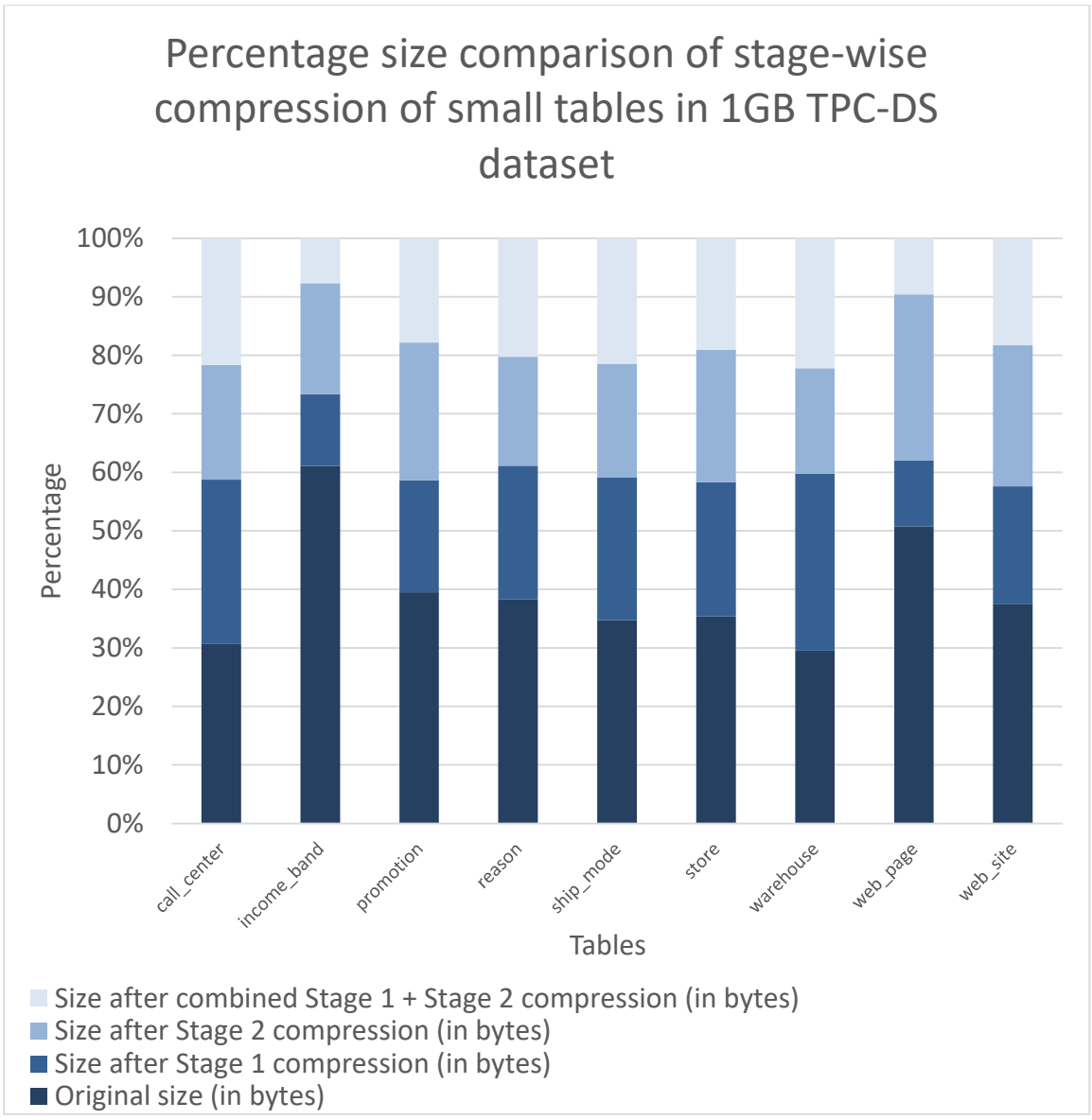

Figure 6.8: Percentage Comparison of Space Reduction Achieved on Each Small Sized Table in 1 GB TPC-DS Benchmark Dataset Using SA128 Compression for different stages of compression, i.e., stage-1 compression, stage-2 compression and combined stage-1 + stage-2 compression. Refer to appendix A, B and C for more details regarding the data collected.

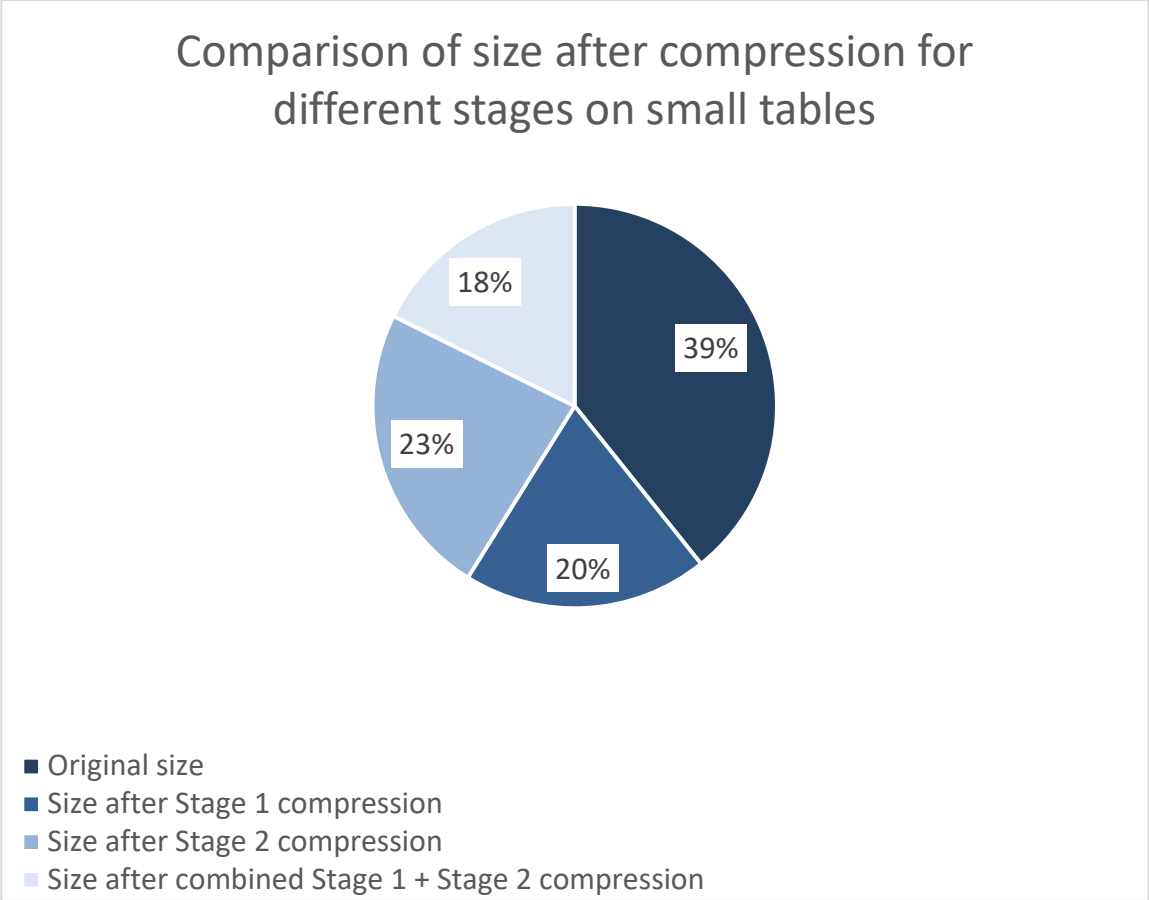

Figure 6.9: Combined Space Reduction Achieved for All Small Sized Tables in 1 GB TPC-DS Benchmark Dataset Using SA128 Compression for different stages of compression, i.e., stage-1 compression, stage-2 compression and combined stage-1 + stage-2 compression. The above graph shows that for large tables, the SA128 algorithm achieved space savings up to 53.84% using both stage-1 and stage-2 compression, 48.71% using only stage-1 compression and 41.0% using only stage-2 compression. Refer to Appendix A, B and C for more details regarding the data collected.

6.3.5 Datatype Specific Results

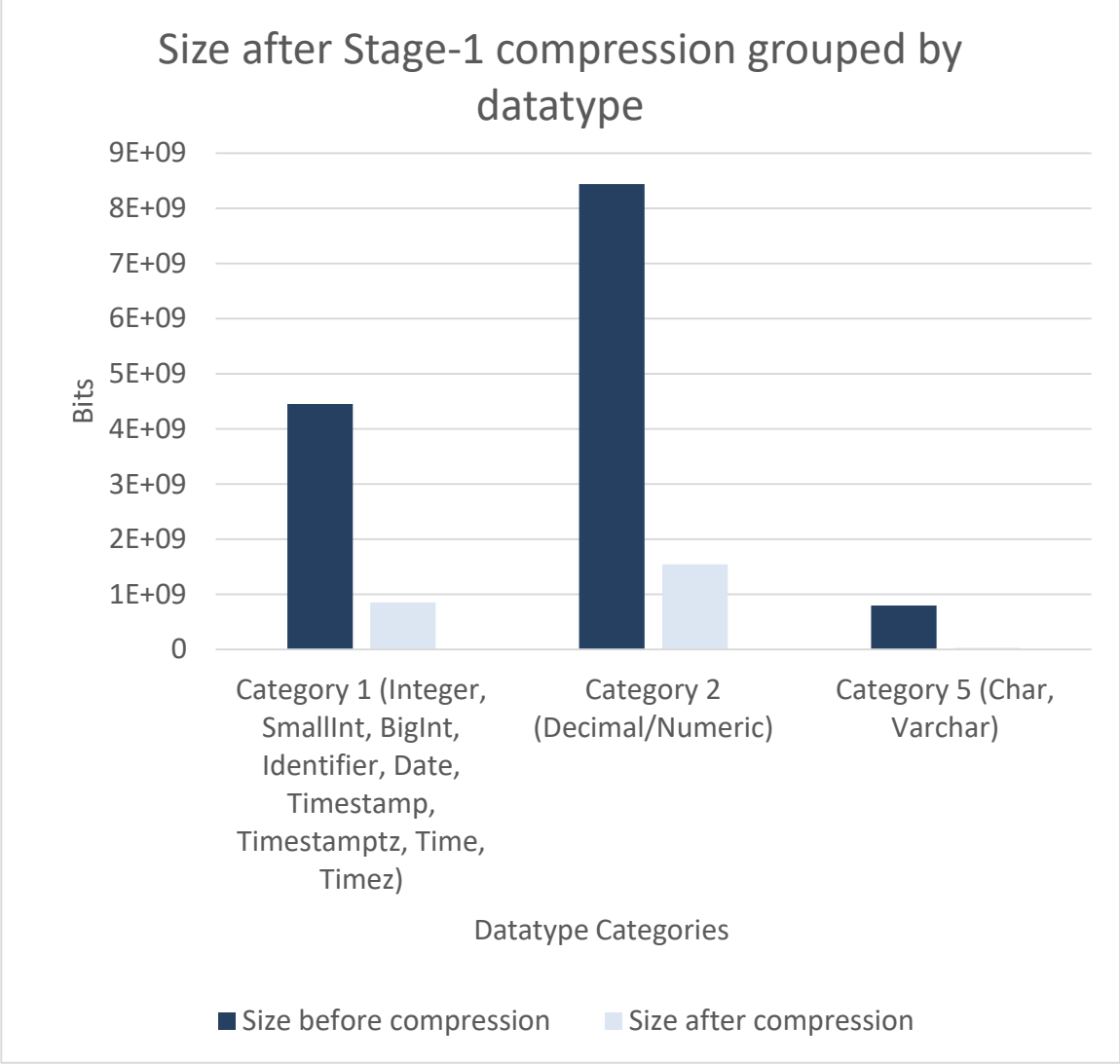

Figure 6.10: Space Reduction Achieved for Each of the Three Datatype Categories Available in 1 GB TPC-DS Benchmark Dataset. We achieve a space reduction of 96.21% for Category-5 datatypes, 81.7% for Category-2 datatypes and 80.9% for Category-1 datatypes. Therefore, compression for Category-5 > Category 2 > Category 1. Refer to Appendix D for more details regarding the data collected.

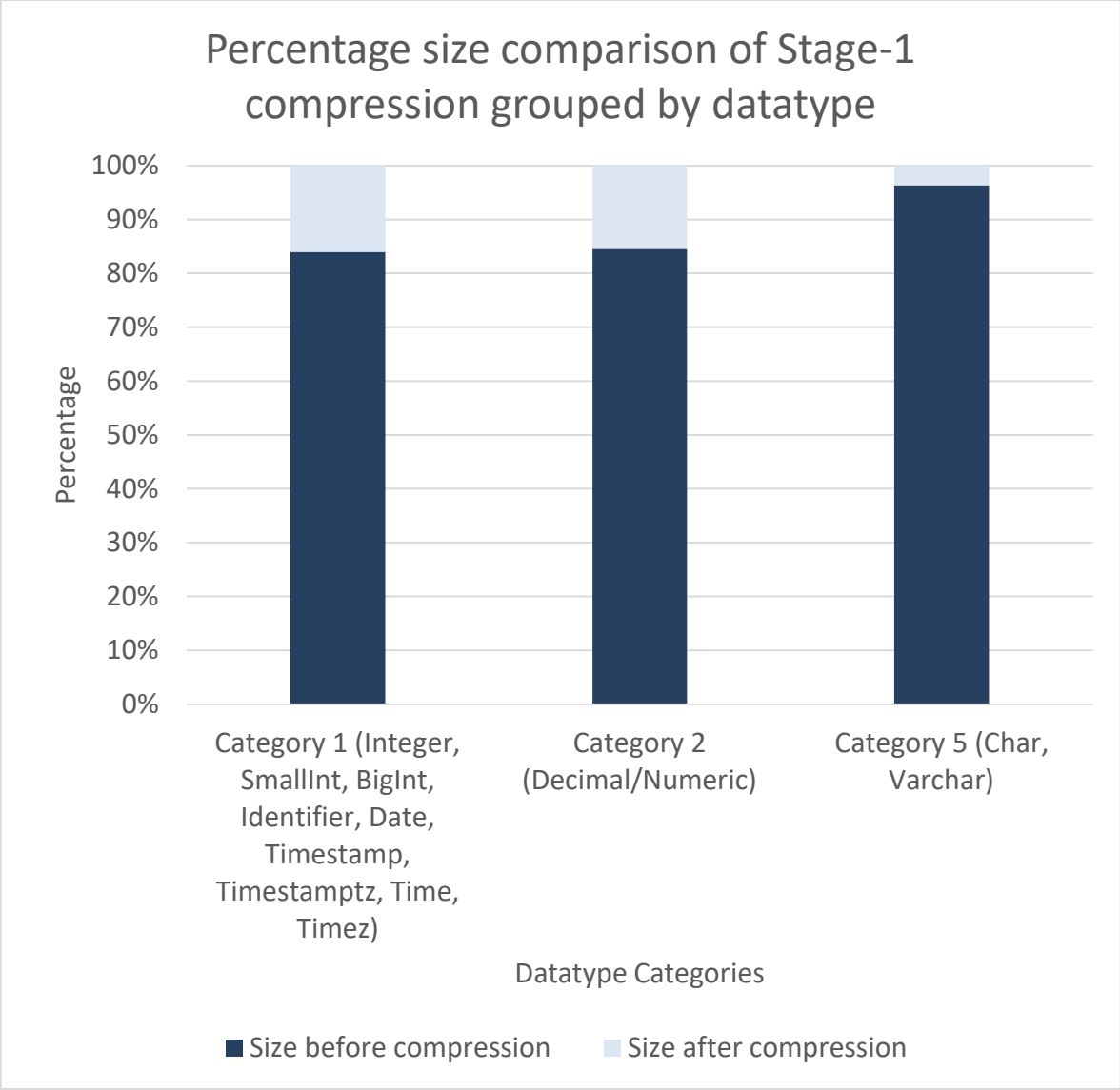

Figure 6.11: Percentage Wise Breakdown of the Space Reduction Achieved for Each of the Three Datatype Categories Available in 1 GB TPC-DS Benchmark Dataset. We can see that compression efficiency is highest for char and varchar types and almost similar for integer, smallint, bigint, date, timestamp, timestamptz, time, timez and numeric types. Refer to Appendix D for more details regarding the data collected.

6.3.6 Compression Results vs Other Compression Algorithms for 1 GB TPC-DS Benchmark Dataset (Dataset – 1)

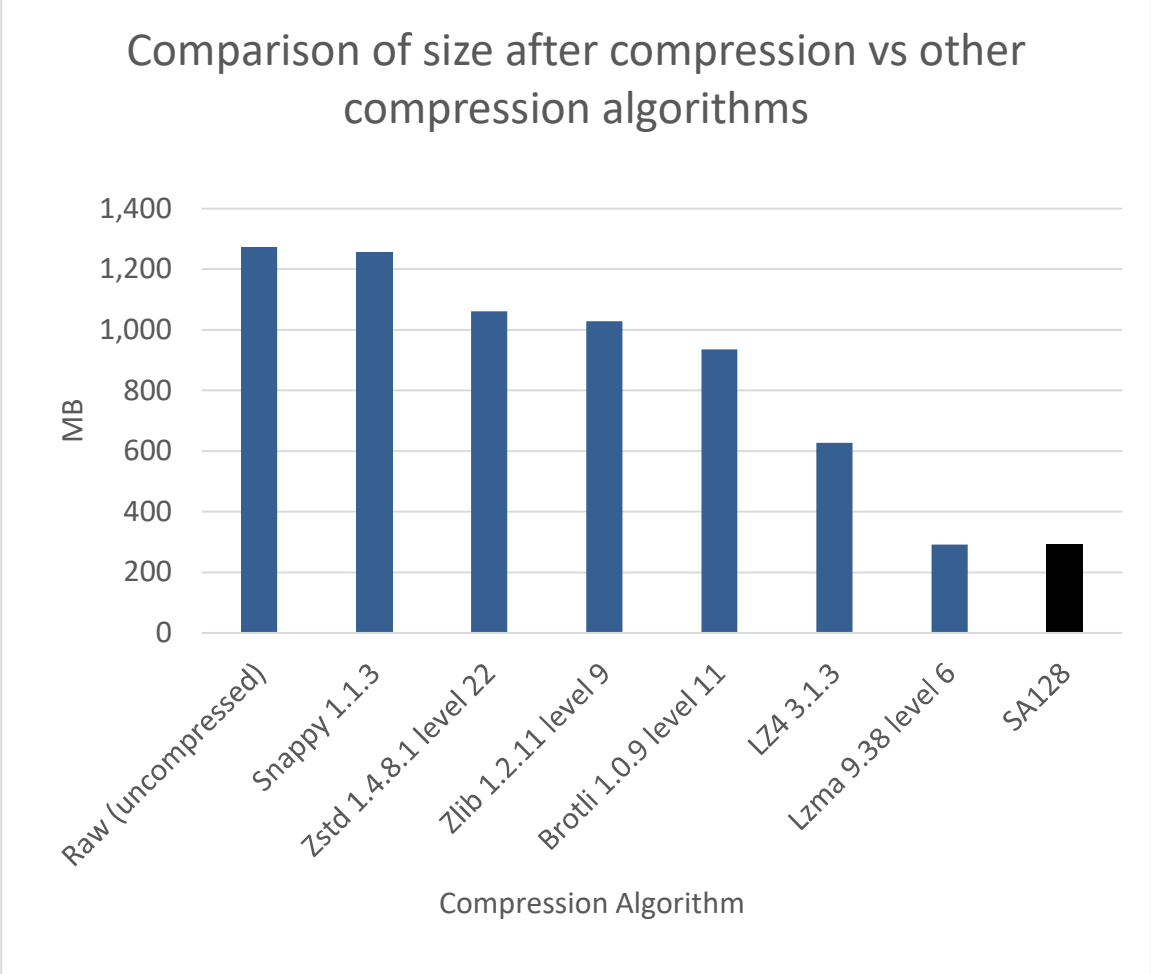

Figure 6.12: Comparative Analysis of Space Savings Achieved Using SA128 Against Other State of the Art Compression Algorithms on 1 GB TPC-DS Benchmark Dataset. For our 1 GB TPC-DS benchmark dataset, our SA128 compression algorithm was able to achieve 60.41% better compression than Zstandard (ZSTD), 75.64% better compression than Snappy, 57.84% better compression than Zlib, 26.36% better compression than LZ4 and 50.52% better compression than Brotli. Only Lzma compressed better by a margin of 0.06% which is comparable to the results achieved using SA128 compression. Refer to Appendix E for more details regarding the data collected.

6.3.7 Performance Results vs Other Compression Algorithms for 1 GB TPC-DS Benchmark Dataset (Dataset – 1)

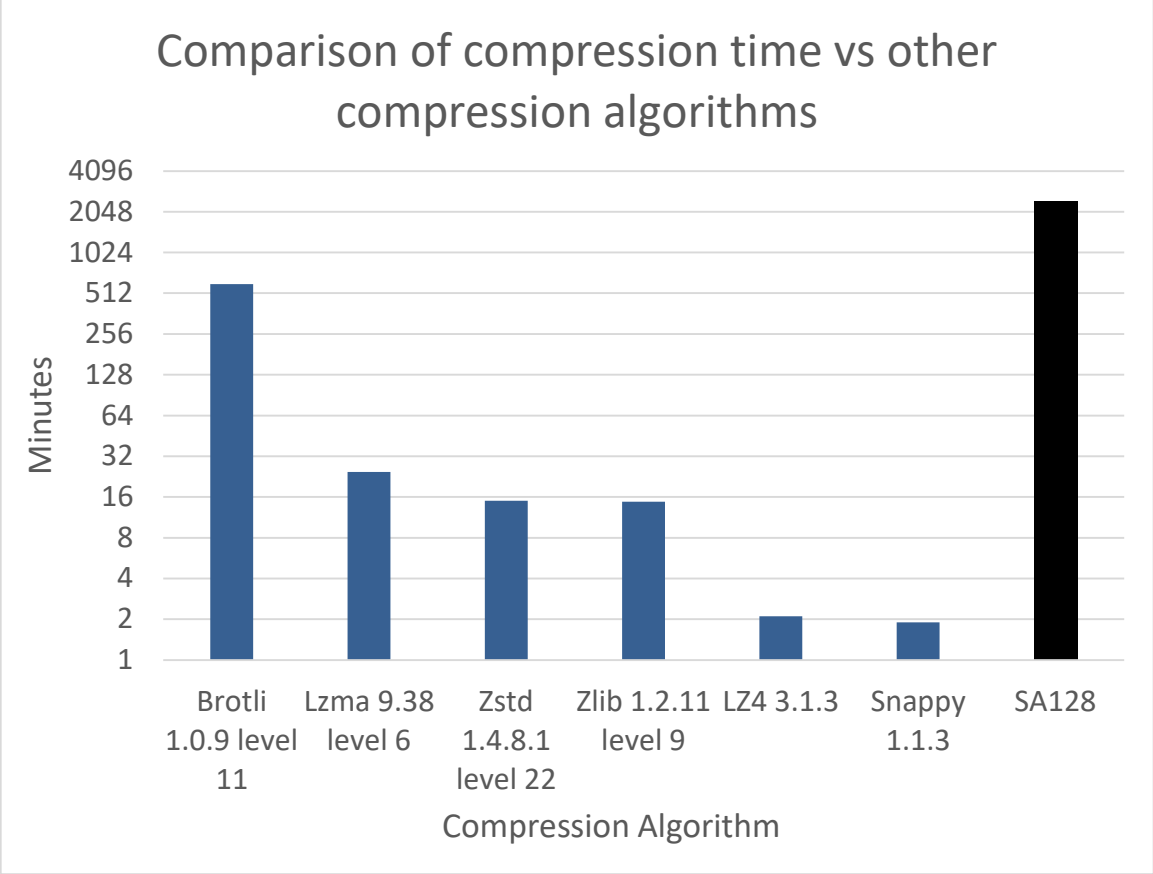

Figure 6.13: Comparative Analysis of Compression Times Using SA128 Against Other State of the Art Compression Algorithms on 1 GB TPC-DS Benchmark Dataset. The data shows us that SA128 has a slow compression time compared to all other algorithms. The best performing algorithm Snappy 1.1.3 has a compression speed of almost 99.92% faster than SA128 and the second worst performing algorithms Brotli 1.0.9 level 11 has a compression speed of almost 75.84% better than SA128. Because of this reason, although using SA128 gives good compression, it's poor performance makes it unsuitable for real-time databases where query execution times are important. Therefore, the primary use cases for SA128 are in the area of data archival and non-real time databases where storage optimization is of higher priority than query execution times. Refer to Appendix J for more details regarding the data collected.

6.3.8 Results on 1 GB Generated Dataset (Dataset – 2)

In this section, we discuss the compression results achieved on our 1 GB Generated Dataset which contains 4 different tables.

6.3.9 Table-wise Compression Results

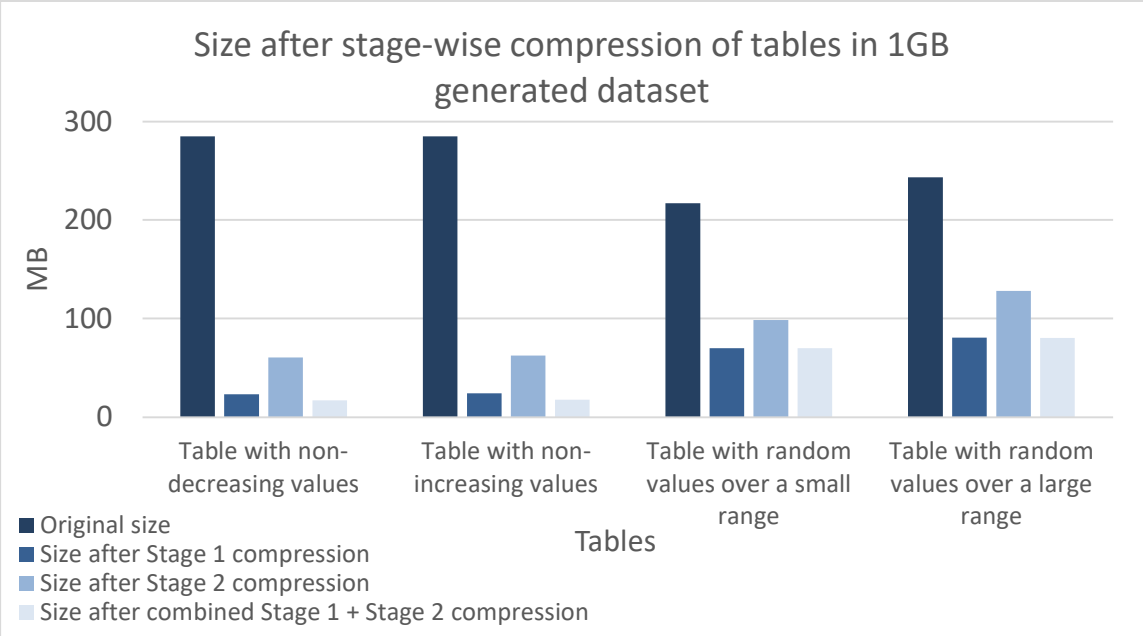

Figure 6.14: Comparative Analysis of Space Reduction Achieved on All 4 Tables from Our 1 GB Generated Dataset using stage-1 SA128 compression, and combined stage-1 + stage-2 SA128 compression. The above graph shows that we achieved the highest compression ratio for all the 4 large tables by combining stage-1 and stage-2 compression in SA128 compression compared to compressing our tables using only stage-1 compression. The space reduction achieved are 94.08% for tables with non-decreasing values, 93.83% for tables with non-increasing values, 67.86% for tables with random values of a small range, 66.97% for tables with random values over a large range. For both the tables with random values, we have a comparable performance between stage-1 SA128 compression and combined stage-1 + stage-2 SA128 compression. For the non-increasing and non-decreasing tables, compression achieved using stage-1 + stage-2 SA128 compression is clearly much better. Refer to appendix F and G for more details regarding the data collected.

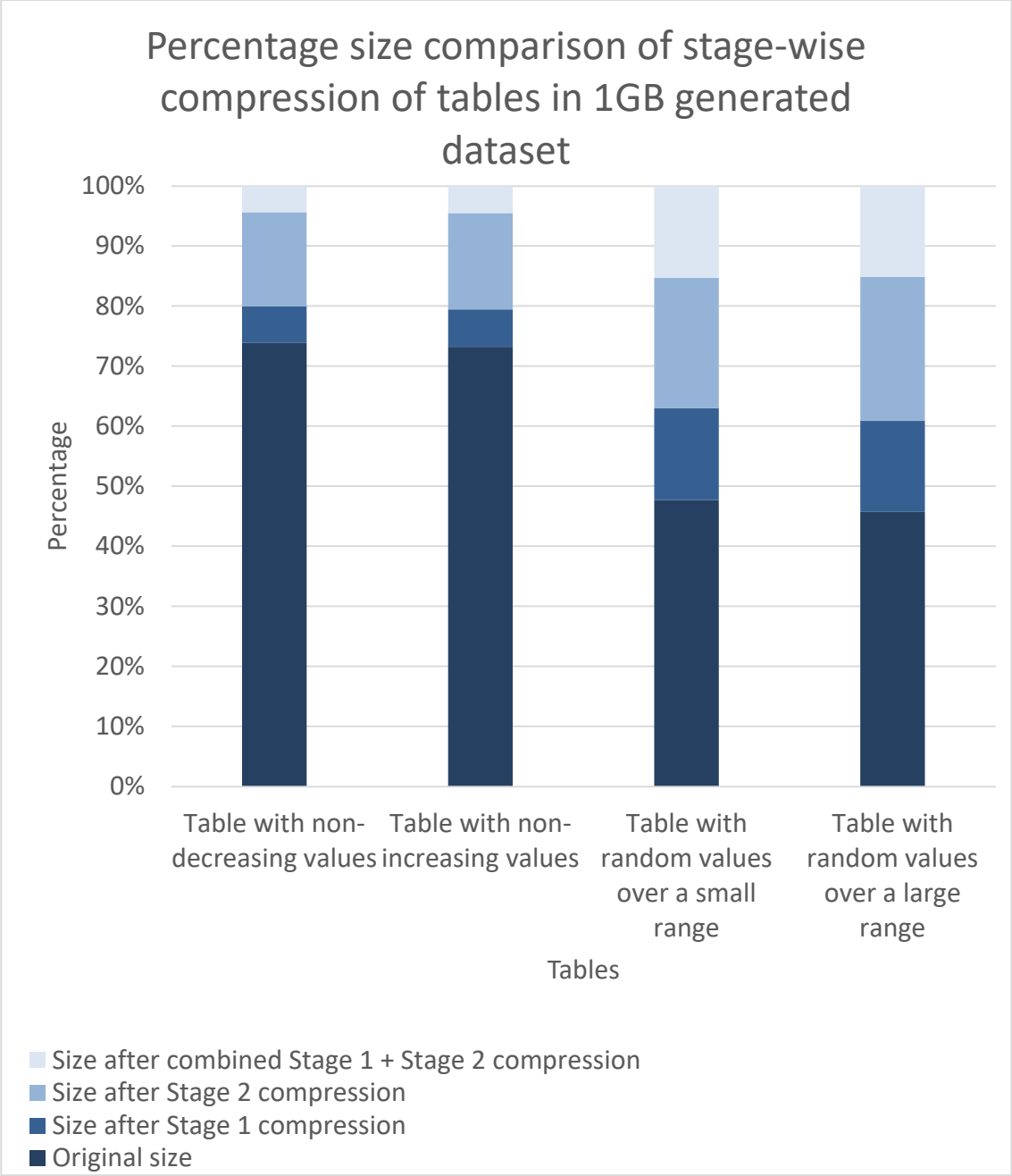

Figure 6.15: Percentage Wise Breakdown of the Space Reduction Achieved for Each of the 4 Tables Available in 1 GB Generated Dataset. Refer to appendix F and G for more details regarding the data collected.

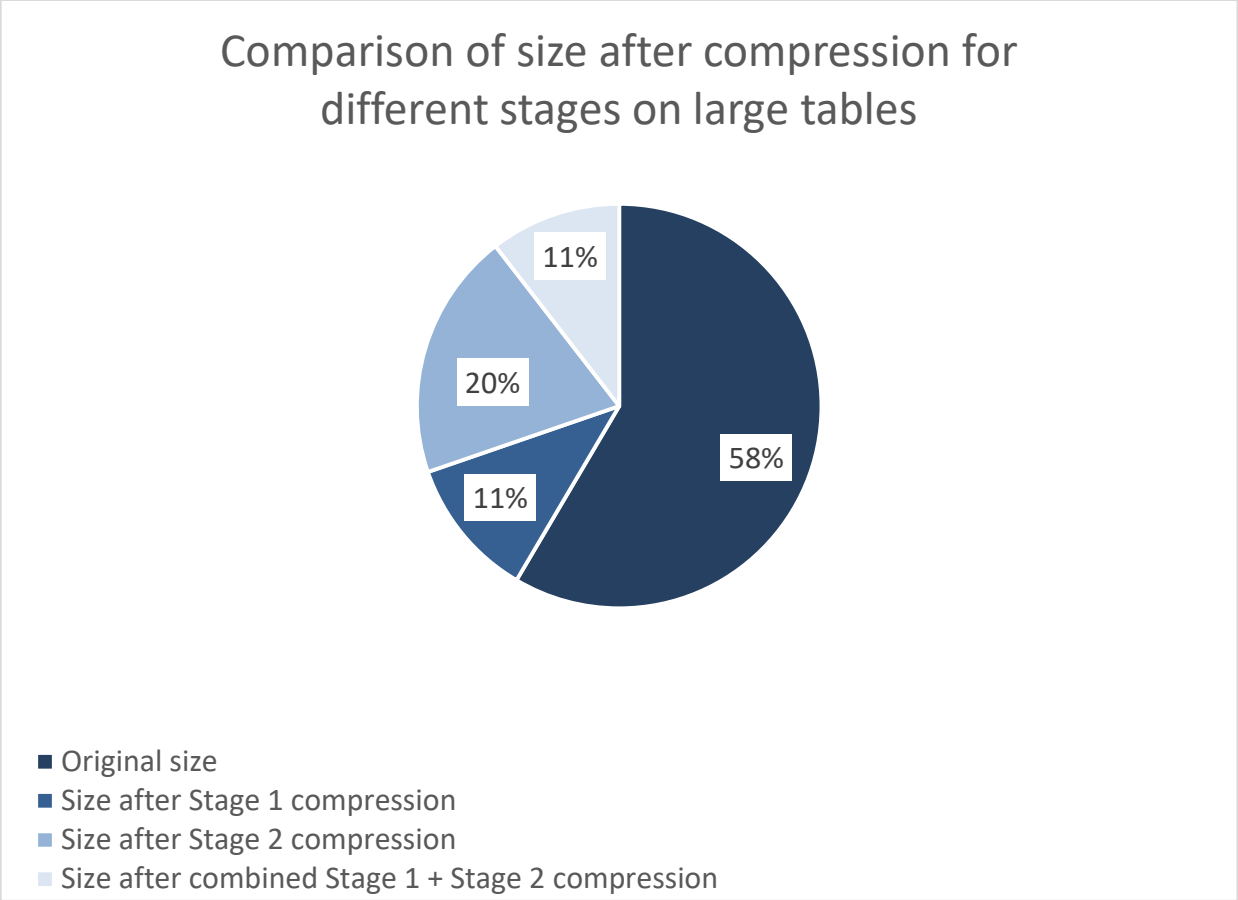

Figure 6.16: Combined Space Reduction Achieved for All 4 Tables from Our 1 GB Generated Dataset Using SA128 Compression for stage-1 SA128 compression and combined stage-1 + stage-2 SA128 compression. The above graph shows that for all the 4 tables, the SA128 algorithm achieved space savings up to 82.19% using both stage-1 and stage-2 compression and 80.82% using only stage-1 compression. Refer to appendix F and G for more details regarding the data collected.

6.3.10 Datatype Specific Compression Results

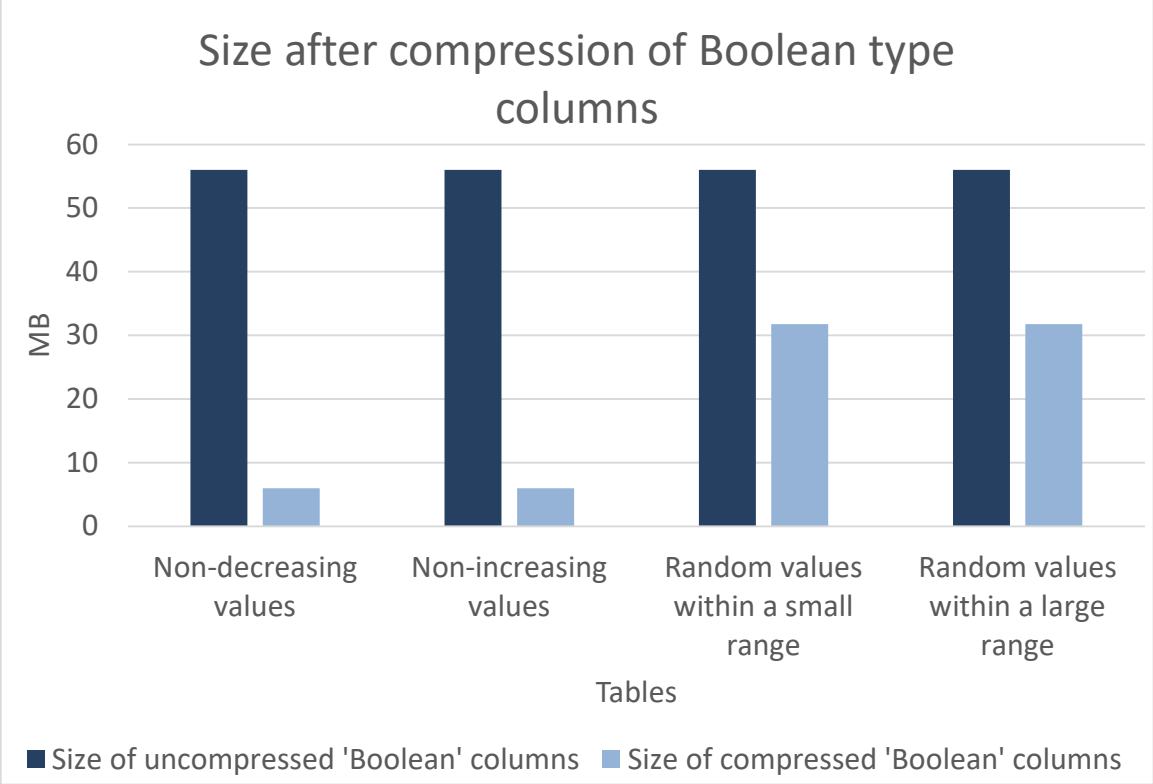

Figure 6.17: Space Reduction Achieved for Boolean Columns in 1 GB Generated Dataset. We achieve a space reduction of 66.3% for Category-4 (boolean datatypes). The compression is almost similar for the non-decreasing and non-increasing columns. The same is true for both the columns with random values. The data also suggests that for random values, compression is less compared to non-decreasing and non-increasing columns. Refer to Appendix H for more details regarding the data collected.

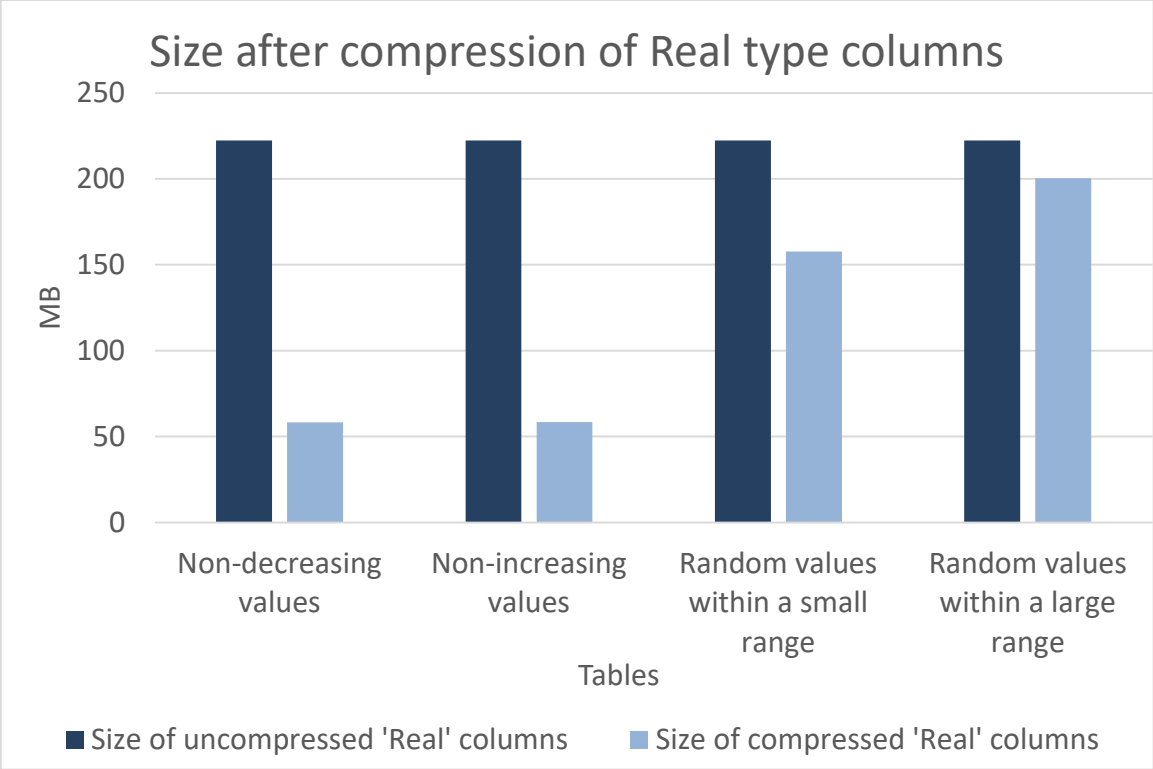

Figure 6.18: Space Reduction Achieved for Real Columns in 1 GB Generated Dataset. We achieve a space reduction of 46.58% for Category-3 Real datatypes. The compression is almost similar for the non-decreasing and non-increasing columns. However, the same is not true for both the columns with random values. The column with random values over a large range compresses lesser than the column with random values over a small range. The data also suggests that for random values, compression is less compared to non-decreasing and non-increasing columns. Refer to Appendix H for more details regarding the data collected.

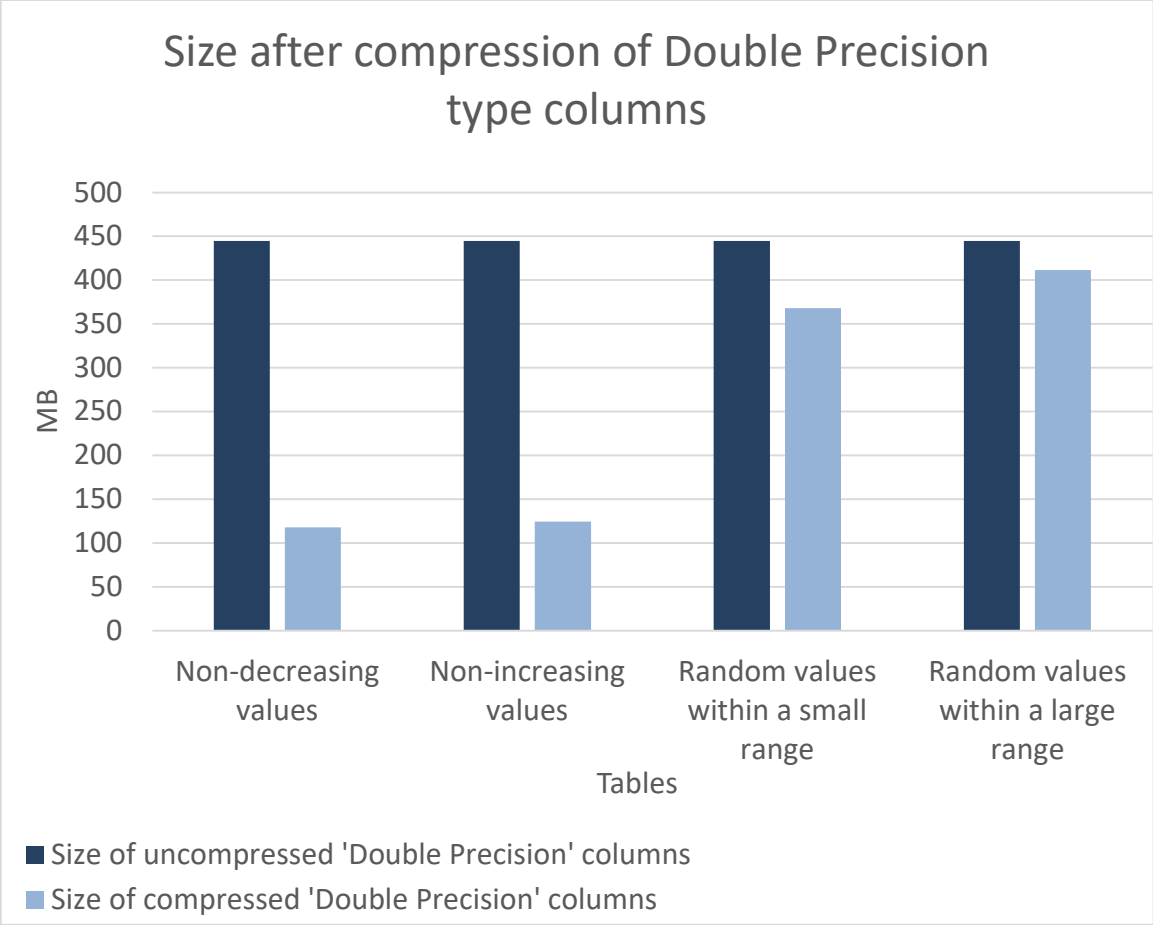

Figure 6.19: Space Reduction Achieved for Double Precision Columns in 1 GB Generated Dataset. We achieve a space reduction of 42.55% for Category-3 Double Precision datatypes. The compression is almost similar for the non-decreasing and non-increasing columns. However, the same is not true for both the columns with random values. The column with random values over a large range compresses lesser than the column with random values over a small range. The data also suggests that for random values, compression is less compared to non-decreasing and non-increasing columns. Refer to Appendix H for more details regarding the data collected.

6.3.11 Compression Results vs other Compression Algorithms for 1 GB Generated Dataset (Dataset – 2)

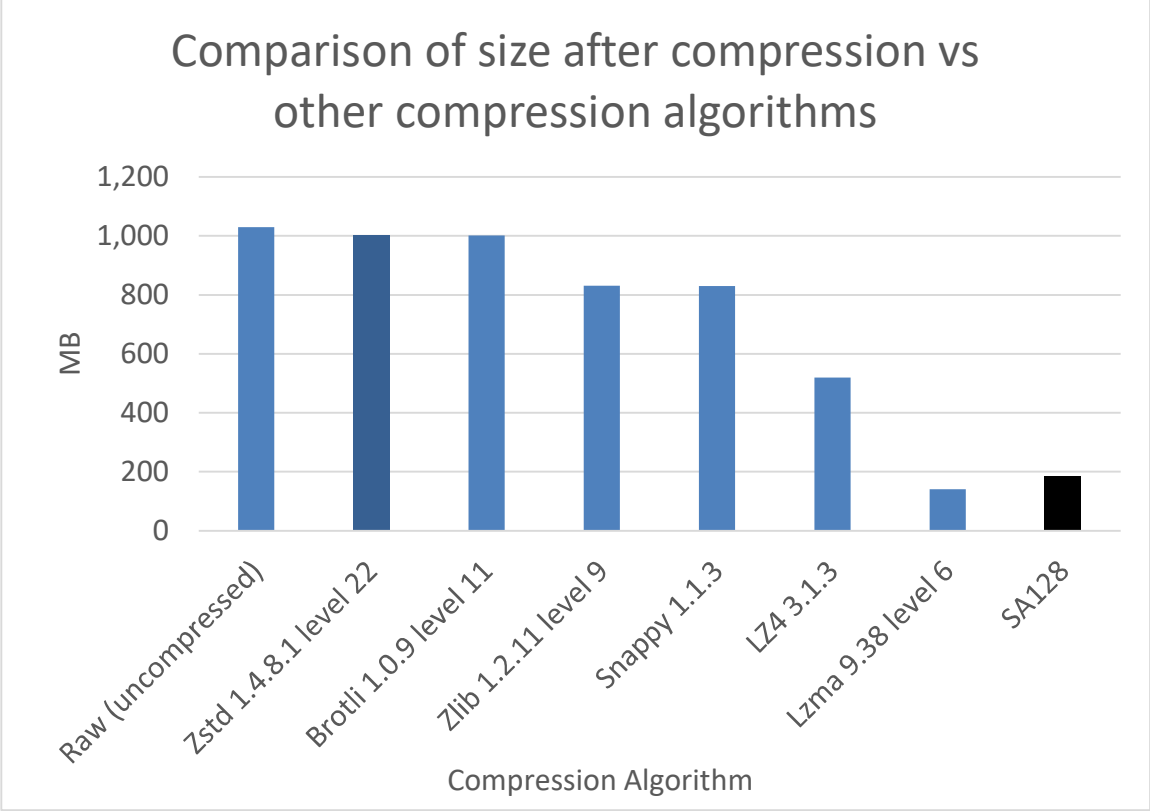

Figure 6.20: Comparative Analysis of Space Saving Achieved Using SA128 Against Other State of the Art Compression Algorithms on 1 GB Generated Dataset. For our 1 GB generated dataset, our SA128 compression algorithm was able to achieve 79.38% better compression than Zstandard (ZSTD), 62.63% better compression than Snappy, 62.74% better compression than Zlib, 32.59% better compression than LZ4 and 79.25% better compression than Brotli. Only Lzma compressed better by a margin of 4.25%. Refer to Appendix I for more details regarding the data collected.

6.3.12 Performance Results vs other Compression Algorithms for 1 GB Generated Dataset (Dataset – 2)

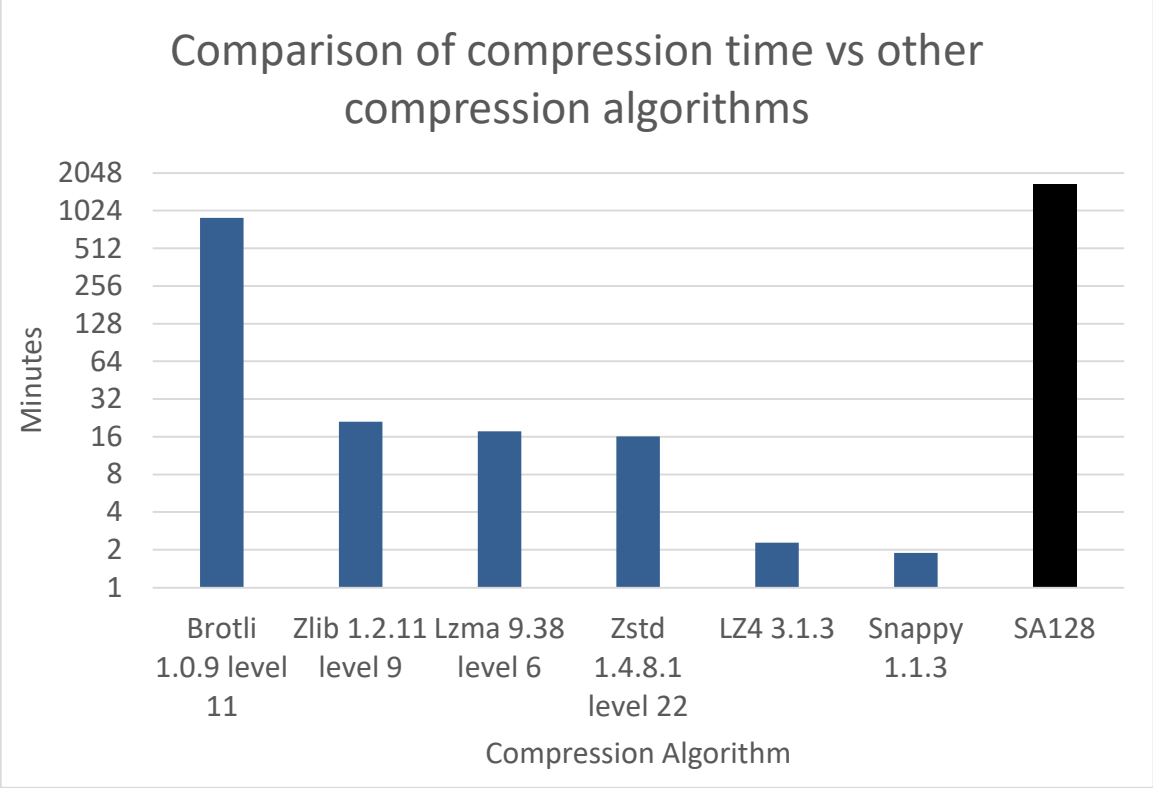

Figure 6.21: Comparative Analysis of Compression Times Achieved Using SA128 Against Other State of the Art Compression Algorithms on 1 GB Generated Dataset. The data shows us that SA128 has a slow compression time compared to all other algorithms. The best performing algorithm Snappy 1.1.3 has a compression speed of almost 99.88% faster than SA128 and the second worst performing algorithms Brotli 1.0.9 level 11 has a compression speed of almost 46.17% better than SA128. These results are slightly better compared to the performance results on 1 GB TPC-DS Dataset. However, although using SA128 gives good compression, it's poor performance makes it unsuitable for real-time databases where query execution times are important. Therefore, the primary use cases for SA128 are in the area of data archival and non-real time databases where storage optimization is of higher priority than query execution times. Refer to Appendix K for more details regarding the data collected.

Chapter 7

LIMITATIONS AND ASSUMPTIONS

A large fraction of popular databases such as AWS Redshift, HadoopDB, etc are derived from PostgreSQL. Therefore, in our implementation, we have assumed PostgreSQL as our foundation database so that it can easily be extended to databases build on top of PostgreSQL.

Although the SA128 compression and decompression algorithm is intended for compression in columnar databases, in our implementation, we used file storage to store the tables belonging to our dataset as ".dat" files instead of storing them in real columnar databases. Therefore, during result collection, we assumed the storage space a column would take when it is stored in a PostgreSQL table by taking into consideration the datatype specific storage format and requirements, instead of the storage space it takes in its file format. The goal of this thesis was to provide a proof of concept (POC) for a smart compression algorithm which when extended for databases, can lead to identical results.

The SA128 algorithm currently supports 14 popular datatypes such as smallint/int2, integer/int4, bigint/int8, date, timestamp (without timezone), timestamp with timezone/timestamptz, time (without timezone), time with timezone/timez, numeric/decimal, real/float4, double precision/float8, boolean, character/char and character varying/varchar. For datatypes not belonging to this list, the column is compressed only using stage-2 compression (rANS entropy encoding).

The goal of this POC work as part of the thesis was to focus on optimizing the compression ratio for our algorithm and its efficiency compared to other state of the art compression algorithms such as ZSTD, Snappy, LZMA, Zlib, LZ4 and Brotli. The decompression time was not a priority for the current 1.0 version, but it can be significantly improved with parallel processing, implementing a C/C++ port our compressor/decompressor and further optimizations in future releases. Therefore, the

current 1.0 release can be used for compressing non-real time databases where fast query responses and decompression times are not important. For example, in applications such as data archiving where storage optimization is more important than query performance.

SA128 compression is highly effective for compressing large and medium sized tables. For very small tables, sometimes SA128 compression may result in increasing the size of the original data. This is due to the overhead due to the block header during stage-1 compression because the metadata information stored in the block header occupies more space than the space reduction achieved on the data block.

Chapter 8

DISCUSSIONS AND CONCLUSION

In this thesis work, we have shown that if data compression techniques are dynamically adapted based on the characteristics of the data set and the data type of the data set, we can get a large increase in compression ratio. When this technique is combined with an entropy encoding stage such as rANS, we reach very close to the entropy limit for the data being compressed.

For TPC-DS benchmark datasets, our SA128 algorithm achieves space savings up to 77% for large tables, 81% for medium tables and 54% for small tables compared the size occupied by uncompressed tables. Also, for generated datasets, our algorithm achieved up to 82% savings in space. For benchmark TPC-DS datasets, our SA128 was able to achieve 60.41% better compression than Zstandard (ZSTD), 75.64% better compression than Snappy, 57.84% better compression than Zlib, 26.36% better compression than LZ4 and 50.52% better compression than Brotli. Only Lzma compressed better by a margin of 0.06% which is comparable to the results achieved using SA128 compression. These results demonstrate the effectiveness of our compression strategy which was the core part of this thesis study. The results from the above data conclude that the approach followed by SA128 successfully answers the research questions specified in section 3.2.

For Category-1 compression, the sequence of steps selected is due to a combination of all the below reasons.

1. The Delta and Delta of Delta encoding steps are executed on multiple block copies before Run-length encoding because Run-length encoding is equally effective when applied before or after Delta and Delta of Delta encoding. This is because Delta encoding does not change the number of runs in values.

2. Zig-zag encoding is applied after the Delta and Delta of Delta encoding step because additional negative numbers can arise in the block after Delta or Delta of Delta encoding step, which needs to be represented efficiently, especially if the magnitude of the negative values are small.
3. The Frame of Reference and Bit Packing steps are applied towards the end to bring down the range of all the values and break down larger values into smaller values respectively. This is done so that few bits are required to represent each block value. This helps us get a better compression towards the end if Delta Encoding, Delta of Delta Encoding and Run-length encoding did not give us good compression in the previous steps.

The current version 1.0 of SA128 does not prioritize performance and only optimizes storage. Performance of SA128 is 75.84% and 46.17% worse compared to Snappy for the TPC-DS Benchmark Dataset and Generated Dataset respectively (where Snappy has the second worst performance). Therefore, it is not ideal to use SA128 for real-time database applications and are favourable in applications such as data archival, etc where storage optimization is prioritized. The performance results in the later sections 6.3.7 and 6.3.12 are highly dependent on the system used to testing the results. The results can be significantly improved if tested on a more powerful server machine with superior configurations compared to the above configuration. Future versions of SA128 would work on improving performance of the algorithm which would make it useful in Big Data applications, data warehousing, business intelligence, etc.

Chapter 9

FUTURE WORK

Some of the future enhancements and research areas to extend our SA128 are given below:

1. Extending support for other database management systems other than PostgreSQL such as MySQL, NoSQL, etc.
2. Supporting more complex data types in PostgreSQL and databases supported in future.
3. Building SA128 as a pluggable library for PostgreSQL database.
4. Improving compression and decompression time for both stages of compression and improving query performance so that real-time database applications can be supported using SA128 compression.
5. Implementing a C/C++ port of our Python prototype to improve performance.
6. Implementing better alternatives of LZ compression techniques in category-5 compression from stage 1 such as LZMA, LZ4, etc.

REFERENCES

Ziv J., Lempel A., "A Universal Algorithm for Sequential Data Compression", IEEE Transactions on Information Theory, Vol. 23, No. 3 (1977), pp. 337-343.

Pelkonen, T., Franklin, S., Teller, J., Cavallaro, P., Huang, Q., Meza, J. & Veeraraghavan, K. (2015). Gorilla: A fast, scalable, in-memory time series database. Proceedings of the VLDB Endowment, 8, 1816--1827.

Daniel Lemire and Leonid Boytsov. 2015. Decoding billions of integers per second through vectorization. Software: Practice and Experience 45, 1 (2015), 1--29.

Duda, J., "Asymmetric numeral systems: entropy coding combining speed of Huffman coding with compression rate of arithmetic coding." *arXiv: Information Theory* (2013): n. pag.

Goldstein J, Ramakrishnan R, Shaft U. Compressing relations and indexes. Proceedings of the Fourteenth International Conference on Data Engineering, ICDE '98, IEEE Computer Society: Washington, DC, USA, 1998; 370–379.

Delbru R, Campinas S, Tummarello G. Searching web data: An entity retrieval and high-performance indexing model. Web Semantics Jan 2012; 10:33–58, doi:10.1016/j.websem.2011.04.004.

HUFFMAN, D. A. 1952. A method for the construction of minimum-redundancy codes. In Proceedings of the Institute of Electrical and Radio Engineers 40, 9 (Sept.), pp. 1098-1101.

RISSANEN, J., AND LANGDON, G. G. 1979. Arithmetic coding. IBM J. Res. Dev. 23, 2 (Mar.), 149-162.

Transaction Processing Performance Council (TPC) 2020, accessed 1 December 2020, <http://tpc.org/tpc_documents_current_versions/pdf/tpc-ds_v2.13.0.pdf>

Mike Freedman 2019, accessed 1 December 2020, <https://blog.timescale.com/blog/building-columnar-compression-in-a-row-oriented-database/>

Joshua, Lockerman, Ajay Kulkarni 2020, accessed 1 December 2020, <https://blog.timescale.com/blog/time-series-compression-algorithms-explained/>.

Daniel Lemire 2012, accessed 1 December 2020, <https://lemire.me/blog/2012/02/08/effective-compression-using-frame-of-reference-and-delta-coding>.

Roman Cheplyaka 2017, accessed 1 December 2020, <https://ro-che.info/articles/2017-08-20-understanding-ans>.

M. Fürstenau, zigzag-encoding.README, 2015, GitHub Repository, accessed 1 December 2020, <https://gist.github.com/mfuerstenau/ba870a29e16536fdbaba>.

History of Lossless Data Compression Algorithms 2019, Engineering and Technology History Wiki, accessed 1 December 2020, <https://ethw.org/History_of_Lossless_Data_Compression_Algorithms>.

PostgreSQL 13 Documentation 2021, The PostgreSQL Global Development Group, accessed 1 March 2021, <https://www.postgresql.org/docs/current/datatype.html>.

Fedor Glazov, Python-rANSCoder, 2020, GitHub Repository, accessed 1 December 2020, <https://github.com/FGlazov/Python-rANSCoder>.

Wesam Manassra, LZ77-Compressor, 2020, GitHub Repository, accessed 1 January 2021, <https://github.com/manassra/LZ77-Compressor>.

Sergey Dryabzhinsky and Anton Shuk, zstd 1.4.8.1, 2020, Facebook, accessed 1 January 2021, Python Library, <https://pypi.org/project/zstd/>.

Snappy, release 1.1.3, 2015, Google, GitHub Repository, accessed 1 February 2021, <https://github.com/google/snappy/releases/tag/1.1.3>.

Igor Pavlov, lzma 9.38 beta, 2015, accessed 1 February 2021, <http://sevenzip.sourceforge.net/sdk.html>.

Mark Adler, Zlib, release 1.2.11, 2017, GitHub Repository, accessed 1 February 2021, <https://github.com/madler/zlib/releases/tag/v1.2.11>.

Jonathan Underwood, Lz4 3.1.3, 2021, Python Library, accessed 1 February 2021, <https://pypi.org/project/lz4/>.

Brotli, release 1.0.9, 2020, Google, GitHub Repository, accessed 1 February 2021, <https://github.com/google/brotli/releases/tag/v1.0.9>.

Mike Freedman. “Building columnar compression in row-oriented database”, TimescaleDB, 31 October 2019, <https://blog.timescale.com/blog/building-columnar-compression-in-a-row-oriented-database/>.

APPENDIX A

DETAILED COMPRESSION RESULTS FOR SA128 COMPRESSION (ONLY USING STAGE-1) ON 1 GB TPC-DS BENCHMARK DATASET

Table name	Original Size (bytes)	Size after compression (bytes)	Compression Ratio	Space savings
call_center	1897	1736	1.0927	8.49%
catalog_page	1643510	1047870	1.5684	36.24%
catalog_returns	21522438	8132433	2.6465	62.21%
catalog_sales	297351932	82877425	3.5879	72.13%
customer	13309372	5815536	2.2886	56.3%
customer_address	5552165	1039001	5.3438	81.29%
customer_demographics	82580896	3933954	20.9918	95.24%
date_dim	10390487	846637	12.2727	91.85%
household_demographics	158853	7493	21.2002	95.28%
income_band	348	70	4.9714	70.89%
inventory	248165139	27670350	8.9686	88.85%
item.dat	5069899	2559284	1.981	49.52%
promotion	37533	18206	2.0616	51.49%
reason	1374	822	1.6715	40.17%
ship_mode	1133	796	1.4234	29.74%
store	3167	2052	1.5434	35.21%
store_returns	32997519	12680449	2.6022	61.57%
store_sales	391325813	120195822	3.2557	69.28%
time_dim	5194180	580174	8.9528	88.83%
warehouse	590	604	0.9768	-2.37%
web_page	5836	1303	4.4789	77.67%
web_returns	9877999	3938056	2.5083	60.13%
web_sales	147597058	40260141	3.6661	72.72%
web_site	8801	4713	1.8674	46.45%

APPENDIX B

DETAILED COMPRESSION RESULTS FOR SA128 COMPRESSION (ONLY USING STAGE 2) ON 1 GB TPC-DS BENCHMARK DATASET

Table name	Original Size (bytes)	Size after compression (bytes)	Compression Ratio	Space savings
call_center	1897	1212	1.5652	36.11%
catalog_page	1643510	1058704	1.5524	35.58%
catalog_returns	21522438	9424636	2.2836	56.21%
catalog_sales	297351932	131715804	2.2575	55.7%
customer	13309372	8445928	1.5758	36.54%
customer_address	5552165	3488464	1.5916	37.17%
customer_demographics	82580896	47133576	1.7521	42.92%
date_dim	10390487	5356692	1.9397	48.45%
household_demographics	158853	65888	2.411	58.52%
income_band	348	108	3.2222	68.97%
inventory	248165139	100258592	2.4753	59.6%
item.dat	5069899	3264840	1.5529	35.6%
promotion	37533	22352	1.6792	40.45%
reason	1374	668	2.0569	51.38%
ship_mode	1133	632	1.7927	44.22%
store	3167	2024	1.5647	36.09%
store_returns	32997519	14486984	2.2777	56.1%
store_sales	391325813	173315548	2.2579	55.71%
time_dim	5194180	2863212	1.8141	44.88%
warehouse	590	360	1.6389	38.98%
web_page	5836	3268	1.7858	44%
web_returns	9877999	4339784	2.2761	56.07%
web_sales	147597058	65353716	2.2584	55.72%
web_site	8801	5656	1.556	35.73%

APPENDIX C

DETAILED COMPRESSION RESULTS FOR SA128 COMPRESSION (STAGE 1 AND STAGE 2 COMBINED) ON 1 GB TPC-DS BENCHMARK DATASET

Table name	Original Size (bytes)	Size after compression (bytes)	Compression Ratio	Space savings
call_center	1897	1340	1.4157	29.36%
catalog_page	1643510	945848	1.7376	42.45%
catalog_returns	21522438	7656440	2.811	64.43%
catalog_sales	297351932	79284800	3.7504	73.34%
customer	13309372	5635404	2.3617	57.66%
customer_address	5552165	974716	5.6962	82.44%
customer_demographics	82580896	2358860	35.0088	97.14%
date_dim	10390487	665548	15.6119	93.59%
household_demographics	158853	4360	36.4342	97.26%
income_band	348	44	7.9091	87.36%
inventory	248165139	24194668	10.257	90.25%
item.dat	5069899	2394908	2.1169	52.76%
promotion	37533	16944	2.2151	54.86%
reason	1374	728	1.8874	47.02%
ship_mode	1133	700	1.6186	38.22%
store	3167	1708	1.8542	46.07%
store_returns	32997519	11983596	2.7536	63.68%
store_sales	391325813	112758020	3.4705	71.19%
time_dim	5194180	431212	12.0455	91.7%
warehouse	590	444	1.3288	24.75%
web_page	5836	1104	5.2862	81.08%
web_returns	9877999	3754996	2.6306	61.99%
web_sales	147597058	38647948	3.819	73.82%
web_site	8801	4288	2.0525	51.28%

APPENDIX D

DETAILED COMPRESSION RESULTS FOR SA128 COMPRESSION (STAGE-1 AND STAGE-2) ON 1 GB TPC-DS BENCHMARK DATASET GROUPED BY DATATYPE CATEGORIES

Datatype Category	**Size before compression (in bits)**	**Size after compression (in bits)**
Category 1 (Integer, SmallInt, BigInt, Identifier, Date, Timestamp, Timestamptz, Time, Timez)	4451721056	850169913
Category 2 (Decimal/Numeric)	8436976136	1543558620
Category 5 (Char, Varchar)	798131824	30237363

APPENDIX E

DETAILED COMPRESSION RESULTS FOR SA128 COMPRESSION (STAGE-1 AND STAGE-2) VS OTHER ALGORITHMS ON 1 GB TPC-DS BENCHMARK DATASET

Table name	SA128 (bytes)	Zstd 1.4.8.1 level 22 (bytes)	Snappy 1.1.3 (bytes)	Lzma 9.38 level 6 (bytes)	Zlib 1.2.11 level 9 (bytes)	LZ4 3.1.3 (bytes)	Brotli 1.0.9 level 11 (bytes)
call_center	1,340	1,519	1,867	766	1,480	1,040	1,222
catalog_page	9,45,848	15,19,931	15,92,491	3,30,041	15,15,411	7,15,441	11,88,076
catalog_returns	76,56,440	1,42,48,764	2,17,95,826	81,33,186	1,40,45,197	1,65,85,151	1,33,35,417
catalog_sales	7,92,84,800	17,03,85,185	26,46,33,075	8,00,06,037	16,74,11,674	16,18,00,478	16,72,81,432
customer	56,35,404	1,17,25,513	1,22,84,111	39,63,821	1,16,62,441	85,96,448	1,12,94,418
customer_address	9,74,716	54,80,528	54,34,131	8,73,294	54,42,895	20,89,348	48,69,051
customer_demographics	23,58,860	9,97,32,219	8,60,00,318	11,76,988	9,53,47,797	1,41,67,563	8,86,20,228
date_dim	6,65,548	77,93,539	87,90,442	6,49,023	75,28,782	27,42,312	70,53,095
household_demographics	4,360	2,23,653	1,73,252	6,660	2,09,203	46,277	1,87,604
income_band	44	528	388	120	461	227	428

inventory	2,41,94,668	35,38,70,139	27,16,55,139	3,05,81,908	33,98,38,509	8,64,52,593	29,51,40,459
item	23,94,908	40,80,160	50,00,481	10,32,298	39,86,625	24,02,933	33,85,637
promotion	16,944	32,172	32,511	10,029	31,325	18,001	27,912
reason	728	1,501	1,181	465	1,282	750	1,417
ship_mode	700	1,213	1,016	582	1,072	813	1,166
store	1,708	2,473	3,004	1,079	2,458	1,541	2,088
store_returns	1,19,83,596	2,40,72,687	3,35,98,253	1,27,09,555	2,34,89,630	2,62,66,128	2,21,94,622
store_sales	11,27,58,020	26,59,21,359	38,55,37,856	10,71,57,408	25,84,54,411	21,60,72,982	24,63,45,827
time_dim	4,31,212	54,74,089	46,93,605	2,10,685	48,51,695	13,23,461	53,04,895
warehouse	444	555	564	276	556	336	473
web_page	1,104	5,504	5,377	1,121	5,298	2,174	5,207
web_returns	37,54,996	62,21,941	86,43,912	34,57,199	61,10,132	69,70,778	58,83,588
web_sales	3,86,47,948	8,98,55,881	14,46,91,793	4,05,89,684	8,80,34,756	8,10,51,919	6,27,21,493
web_site	4,288	7,145	8,650	2,614	6,988	4,024	5,705

APPENDIX F

DETAILED COMPRESSION RESULTS FOR SA128 COMPRESSION (ONLY USING STAGE-1) ON 1 GB GENERATED DATASET

Table type	**Original Size (bytes)**	**Size after compression (bytes)**	**Compression Ratio**	**Space savings**
Non-decreasing values	284777810	23244551	12.2514	91.84%
Non-increasing values	284777810	24084695	11.824	91.54%
Random values within a small range	217000000	69730642	3.112	67.87%
Random values within a large range	243459776	80601985	3.0205	66.89%

APPENDIX G

DETAILED COMPRESSION RESULTS FOR SA128 COMPRESSION (STAGE-1 AND STAGE-2) ON 1 GB GENERATED DATASET

Table type	**Original Size (bytes)**	**Size after compression (bytes)**	**Compression Ratio**	**Space savings**
Non-decreasing values	284777810	16854876	16.8959	94.08%
Non-increasing values	284777810	17559804	16.2176	93.83%
Random values within a small range	217000000	69747320	3.1112	67.86%
Random values within a large range	243459776	80423180	3.0272	66.97%

APPENDIX H

DETAILED COMPRESSION RESULTS FOR SA128 COMPRESSION (STAGE-1 AND STAGE-2) ON GENERATED DATASET GROUPED BY DATATYPE CATEGORIES

Table type	Size of uncompressed 'Boolean' column (Category 4)	Size of compressed 'Boolean' column (Category 4)	Size of uncompressed 'Real' column (Category 3)	Size of compressed 'Real' column (Category 3)	Size of uncompressed 'Double Precision' column (Category 3)	Size of compressed 'Double Precision' column (Category 3)
Non-decreasing values	56000000	5960994	222249984	58246558	444499968	117906375
Non-increasing values	56000000	5960994	222249984	58544488	444499968	124228609
Random values within a small range	56000000	31777234	222249984	157719231	444499968	367882976
Random values within a large range	56000000	31782972	222249984	200336756	444499968	411304616

APPENDIX I

DETAILED COMPRESSION RESULTS FOR SA128 COMPRESSION (STAGE-1 AND STAGE-2) VS OTHER ALGORITHMS ON 1 GB GENERATED DATASET

Dataset type	**SA128 (bytes)**	**Zstd 1.4.8.1 level 22 (bytes)**	**Snappy 1.1.3 (bytes)**	**Lzma 9.38 level 6 (bytes)**	**Zlib 1.2.11 level 9 (bytes)**	**LZ4 3.1.3 (bytes)**	**Brotli 1.0.9 level 11 (bytes)**
Non-decreasing values	1,68,54,876	22,91,66,496	21,57,08,567	48,43,244	18,70,91,654	10,24,86,671	26,47,94,923
Non-increasing values	1,75,59,804	22,91,66,496	21,57,08,567	45,29,440	18,70,91,654	10,38,96,451	26,47,94,923
Random values within a small range	6,97,47,320	26,84,58,645	19,23,47,382	6,01,14,930	22,02,97,953	14,72,29,902	22,38,93,416
Random values within a large range	8,04,23,180	27,54,45,674	20,60,21,863	7,12,30,911	23,63,55,739	16,67,29,656	24,74,00,357

APPENDIX J

DETAILED PERFORMANCE RESULTS FOR SA128 COMPRESSION (STAGE-1 AND STAGE-2) VS OTHER ALGORITHMS ON 1 GB TPC-DS BENCHMARK DATASET

Table name	SA128 (seconds)	Zstd 1.4.8.1 level 22 (seconds)	Snappy 1.1.3 (seconds)	Lzma 9.38 level 6 (seconds)	Zlib 1.2.11 level 9 (seconds)	LZ4 3.1.3 (seconds)	Brotli 1.0.9 level 11 (seconds)
call_center	0.4366811	0.1303319	1.5773426	0.3205073	0.2933708	0.3563261	0.3451584
catalog_page	231.4868507	1.673366	2.2356708	1.5689873	1.4247972	0.2166766	33.4822501
catalog_returns	5148.512151	13.848849	1.6678461	30.5955206	10.0134453	1.2510897	234.0672823
catalog_sales	38377.70037	188.816018	18.6293666	351.6022775	97.5715193	19.6599702	3471.438509
customer	5069.595101	14.5198804	2.3108879	17.1598471	10.9767005	1.1579375	242.592673
customer_address	473.2280742	3.5697162	0.563466	5.9612266	4.7294366	0.8303148	118.8600405
customer_demographics	3454.887581	62.55600466	9.431628	64.7205317	98.2416144	8.7474576	3333.355909
date_dim	466.3007508	8.9662253	0.6365887	11.405986	5.2624393	0.7468312	135.7156169
household_demographics	3.6830113	0.4879069	0.372183	0.1897928	0.6602827	0.1743653	12.2070271
income_band	0.5323762	0.1399542	0.0979287	0.0723273	0.1609516	0.09083	0.1356843
inventory	17743.64505	285.8395503	48.3806576	255.5463801	393.5011736	40.5011046	18381.53144
item	2685.321681	3.6743306	1.9909686	5.9389027	3.3598995	1.6369267	59.6010026
promotion	16.8266457	0.1954242	0.2430946	0.2520135	0.1832551	0.1423423	0.8495315
reason	1.3037819	0.1550379	0.1705809	0.298334	0.1245954	0.1088859	0.1784709
ship_mode	1.331422	0.1231088	0.1685167	0.1990435	0.3961256	0.1172685	0.1293318
store	1.7963789	0.1488782	0.1200909	0.2325414	0.1113067	0.1258229	0.1349827
store_returns	8559.911133	19.2585725	2.7019705	49.5209619	16.8946312	2.3405282	364.0769071
store_sales	42843.13556	217.0176965	22.0806776	509.9929592	168.6811037	25.2211857	7659.880805
time_dim	470.5416569	5.9074665	1.881421	2.4466295	5.6542149	1.9246727	171.2270268
warehouse	1.6022269	0.3900925	0.0928006	0.0445481	0.1332628	0.1059779	0.0591814

web_page	1.754422	0.174559	0.157161	0.084332	0.138463	0.154303	0.210741
web_returns	2805.774448	6.791739	0.849223	8.922033	5.589458	0.644420	120.1577151
web_sales	19956.51049	66.1117904	9.640339	150.0581109	63.3075715	7.640709	1534.557929
web_site	6.523709	1.535133	0.1453549	0.1336634	0.1934167	0.1589683	0.2396275

APPENDIX K

DETAILED PERFORMANCE RESULTS FOR SA128 COMPRESSION (STAGE-1 AND STAGE-2) VS OTHER ALGORITHMS ON 1 GB GENERATED DATASET

Dataset type	SA128 (seconds)	Zstd 1.4.8.1 level 22 (seconds)	Snappy 1.1.3 (seconds)	Lzma 9.38 level 6 (seconds)	Zlib 1.2.11 level 9 (seconds)	LZ4 3.1.3 (seconds)	Brotli 1.0.9 level 11 (seconds)
Non-decreasing values	14494.91977	280.675516	34.2860658	271.1570873	370.5804404	27.6235839	14112.28481
Non-increasing values	14049.51347	267.0388133	31.6263264	253.5042029	280.5093933	29.2293526	13870.83912
Random values within a small range	36903.05286	220.7413967	34.5630973	276.5787353	272.0547708	27.7811676	13473.63411
Random values within a large range	34138.85045	195.1213051	36.1044758	256.8656053	342.9404333	28.8372896	12153.94776