TECHNISCHE UNIVERSITÄT MÜNCHEN

SCHOOL OF COMPUTATION, INFORMATION AND TECHNOLOGY - INFORMATICS

Master's Thesis in Data Engineering and Analytics

Exploiting Column Correlations for Compression

Thomas Glas

TECHNISCHE UNIVERSITÄT
MÜNCHEN

SCHOOL OF COMPUTATION, INFORMATION AND
TECHNOLOGY - INFORMATICS

Master's Thesis in Data Engineering and Analytics

Exploiting Column Correlations for Compression

Ausnutzen von Spaltenkorrelationen zur Datenkompression

Author:	Thomas Glas
Supervisor:	Prof. Dr Thomas Neumann
Advisor:	Prof. Dr. Peter Boncz, Azim Afroozeh
Submission Date:	15.12.2023

I confirm that this master's thesis is my own work and I have documented all sources and material used.

Munich, 15.12.2023 Thomas Glas

Acknowledgments

This thesis project was completed at Centrum Wiskunde & Informatica (CWI) in Amsterdam. I want to give a special thank you to Peter Boncz, Azim Afroozeh, and the CWI Database Architectures Group for hosting me and guiding me throughout this project.

Abstract

Open file formats typically use a set of lightweight compression (LWC) schemes to compress columns, which exploit data patterns within a column for compression. However, typical LWC schemes compress each column individually and do not consider column correlations that may be helpful for compression. Real-world datasets exhibit many such correlations and we researched how they can be exploited for LWC. In this thesis, we propose six new multi-column LWC schemes that exploit different types of correlations between columns. To detect the correlations, we use sampling-based algorithms to estimate the compression ratio achieved by each of our multi-column LWC schemes for a pair of columns. During compression, we use both multi-column and single-column LWC schemes in combination and choose the most effective scheme to compress each column. We evaluated the effectiveness of our multi-column LWC schemes on the Public BI benchmark, containing real-world datasets, and achieved 1.2x higher compression ratios than only using single-column LWC schemes.

Contents

Acknowledgments **v**

Abstract **vii**

1 Introduction **1**
1.1 Research Questions 2
1.2 Outline 3

2 Background **5**
2.1 Lightweight Compression Schemes 5
2.1.1 One Value Encoding 5
2.1.2 Frequency Encoding 6
2.1.3 Run-Length Encoding (RLE) 6
2.1.4 Bit-packing 8
2.1.5 Frame of Reference (FOR) 8
2.1.6 Pseudodecimal Encoding (PDE) 9
2.1.7 Dictionary Encoding 10
2.1.8 Fast Static Symbol Table (FSST) 10
2.2 BtrBlocks 11
2.3 Public BI Benchmark 14

3 Related Work **17**
3.1 Exploiting Column Correlations 17
3.1.1 BHUNT 17
3.1.2 CORDS 18
3.1.3 DeCoRel 18
3.2 Column Correlations for Compression 19
3.2.1 Whitebox Compression 19
3.2.2 CorBit: Leveraging Correlations for Compressing Bitmap Indexes 20

4 Design and Implementation **23**
4.1 Single-Column LWC Schemes 23

4.2 Multi-Column LWC Schemes 25
4.2.1 Equality 26
4.2.2 Numerical 28
4.2.3 1-to-1 Dictionary 30
4.2.4 1-to-N Dictionary 32
4.2.5 Dictionary-FOR 35
4.2.6 Dictionary Sharing 36
4.3 Compression Framework 38
4.3.1 Finding Correlated Columns 39
4.3.2 Choosing Correlated Compression Schemes 41
4.3.3 Compression 44
4.3.4 Decompression 46
4.3.5 Multi-Row Group Compression 47
4.4 Compressed column format 48
4.4.1 Single-Column Scheme Compressed Format 49
4.4.2 Compressed Equality Scheme Format 50
4.4.3 Compressed Numerical Scheme Format 51
4.4.4 Compressed 1-to-1 Dictionary Scheme Format 51
4.4.5 Compressed 1-to-N Dictionary Scheme Format 52
4.4.6 Compressed DFOR Scheme Format 52
4.4.7 Compressed Dictionary-Sharing Scheme Format 55
4.4.8 Compressed Dictionary Format 55
4.4.9 Compressed Exceptions Format 55
4.4.10 Compressed Nullmap Format 56

5 Results and Discussion 57
5.1 Baseline 57
5.2 Individual Multi-Column Schemes 57
5.3 Combining All Schemes 61
5.3.1 Reversing Bad Multi-Column Schemes 63
5.3.2 Sample run size 65
5.3.3 Multi-Row Group: Sharing Correlations 65
5.3.4 Compression Ratio Improvement per Table 66

6 Conclusion 69
6.1 Research Questions 69
6.2 Future Work 70

List of Figures 71

List of Tables **75**

Bibliography **77**

1 Introduction

With the increasing demand for efficient data processing and storage platforms, the need to optimize data storage and retrieval processes has become more relevant. The pursuit of efficient storage solutions has led to the development and adoption of columnar file formats. These formats, such as Apache Parquet, Apache ORC, and BtrBlocks organize data based on columns rather than rows, and store values of a column consecutively on disk [1] [2] [3].

Columnar storage provides benefits in terms of query performance and better compression ratios compared to traditional row-oriented formats. The improved compression in columnar storage not only reduces storage requirements but also increases the query performance of data processing systems by increasing I/O performance and reducing data transfer times [4] [5]. Compression for columnar file formats is typically done by applying a set of lightweight compression (LWC) schemes in individual columns. LWC schemes take advantage of the fact that all values of a column share the same data type, and that neighboring values are often similar to each other. Examples of LWC schemes include run-length encoding (RLE) and Dictionary encoding [4]. In RLE, values that are repeated consecutively are encoded as a pair of values and run-lengths. Dictionary encoding is useful for columns with a small number of unique values that occur frequently. Each value in the column is replaced with a smaller code, and a dictionary maps each code to its corresponding value. Since each LWC scheme only exploits a specific data pattern, a set of LWC schemes are commonly used together to cover as many common patterns as possible.

Given a group of columns to compress, LWC schemes are applied to individual columns, and a compression framework needs to decide which LWC scheme is most suitable for a column. This is usually done by inspecting a sample of the column for certain patterns [3] [5]. Since LWC schemes only consider every column individually, they do not make use of patterns that may exist between different columns. We see opportunities for LWC schemes that can exploit correlations between columns. We introduce six new multi-column LWC schemes that exploit correlations we encountered in real-world datasets. Our *Equality Scheme* exploits column pairs that are identical or near-identical. The *Numerical Scheme* exploits numerical linear correlations between pairs of integer columns. The *1-to-1 Dictionary* exploits correlations in which a 1-to-1

mapping exists between values of two columns. The *1-to-N Dictionary* exploits correlations between a categorical column and a column containing members of the categories. The *Dictionary-FOR Scheme* exploits correlations in which the values of a target column can be grouped in distinct and separate ranges, which can be mapped to a category derived by another column. The *Dictionary-Sharing Scheme* exploits correlations in which two columns have a large overlap of unique values.

We implement a compression framework that combines these multi-column LWC schemes with conventional single-column LWC schemes. For every multi-column scheme, we implemented methods to detect correlations using samples and column statistics. The addition of the multi-column schemes allows the framework to exploit any correlations it detects, and fall back to single-column schemes if none are detected for a column. Our implementation is open-sourced and is available in our GitHub repository **https://github.com/cwida/C3**.

1.1 Research Questions

This thesis will investigate the following research questions:

1. How can we exploit column correlations to improve compression? Different kinds of column correlations may require different approaches to exploit them for LWC. Which methods can we use to utilize correlation information to compress columnar data?

2. How can column correlations be efficiently detected? The search space for finding correlated pairs of columns is quadratic, thus a method of efficiently detecting correlations is necessary to keep the overhead as low as possible.

3. How can multi-column LWC schemes that exploit column correlations be used in combination with typical single-column LWC schemes? LWC frameworks are designed to compress one column at a time and are thus not compatible with the requirements of multi-column LWC schemes. How can we design an LWC framework suitable for both single-column and multi-column LWC schemes combined?

4. How much compression benefit does exploiting column correlations provide in real-world datasets compared to existing LWC schemes? Multi-column LWC schemes need to show meaning compression benefit to justify the added complexity, how much benefit can we get in real-world datasets?

5. If compressing multiple row group groups, can we amortize the overhead of finding correlations over multiple row groups by reusing correlations found in one row group with other row groups? Assuming that column correlations hold over entire columns, then correlations found in one row group should also be present in other row groups. How could we use this to reduce the overhead of finding correlations?

1.2 Outline

The rest of the thesis is organized as follows. In Chapter 2 we present background information on LWC schemes, the Public BI benchmark, and the BtrBlocks file format. Chapter 3 gives a brief overview of related works. Chapter 4 goes into detail on the design and implementation of our new compression schemes and compression framework. Chapter 5 presents and discusses the results on the Public BI benchmark. The final conclusions are drawn in Chapter 6.

2 Background

In the following chapter, we present existing lightweight comrpession schemes that are used in our implementation. We further introduce the BtrBlocks file format, which we use as a basis for implementing and testing our compression schemes. Finally, we introduce our chosen benchmark, the Public BI benchmark, and give an overview of its characteristics.

2.1 Lightweight Compression Schemes

Compression schemes used in database systems can be separated into two categories, lightweight and general-purpose (heavyweight) compression compression. General-purpose compression algorithms such as Zstd [6] are designed to work with any type of data by exploiting data patterns on the byte-level. This makes them usable for many use cases, including compression in database systems. In contrast, lightweight (LWC) schemes exploit patterns on data-type level, and are applied to blocks of data containing only one data type, e.g. integers, doubles, and strings [4]. This makes them ideal for columnar file formats, which group data column-wise so data chunks are made up of elements of the same data type. LWC schemes offer faster compression and decompression speeds than general-purpose schemes [7]. One LWC scheme can only exploit one specific pattern, so typically a group of LWC schemes are used together to cover as many common data patterns as possible. To decide which LWC scheme to apply to a column, a compression framework needs to estimate which scheme is the most suitable, typically by inspecting a sample of values [3].

In the following, we introduce single-column LWC schemes used in this thesis.

2.1.1 One Value Encoding

One-Value Encoding or **Constant Encoding** is a LWC scheme and is used for columns in which only a single unique value occurs [3]. This compression scheme only needs to store the unique value only once, leading to large compression ratios. This scheme can be used for arbitrary data types. Given a column

containing n elements, one value encoding compresses it to $\frac{1}{n}$-th size. Figure 2.1 shows an example of one-value encoding.

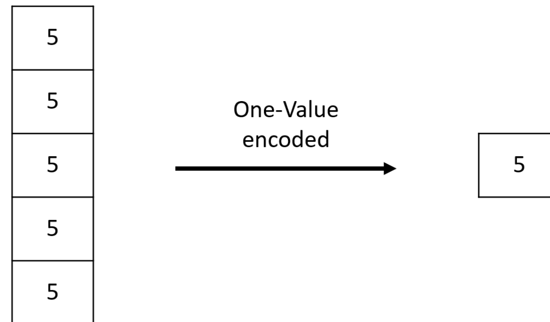

Figure 2.1: An integer column is compressed using one-value encoding.

2.1.2 Frequency Encoding

Frequency Encoding is a LWC scheme applied to columns in which one value occurs exponentially more often than other values [3]. The scheme stores the most common value and keeps an additional bitmap to mark which entries in the column corresponding to this value. All entries which do not correspond to the most common value are stored as uncompressed exceptions. This scheme can be used for arbitrary data types. The compression ratio of this scheme is inversely proportional to the number of exceptions. Given a data type of b bytes and a column with n elements, frequency encoding reduces the size from $b * n$ to $b + \lceil \frac{n}{8} \rceil + e * n$ bytes, with e being the number of exceptions. Figure 2.2 shows an example of frequency encoding.

2.1.3 Run-Length Encoding (RLE)

Run-Length Encoding can be applied to any data type and exploits runs of consecutive repeated values [4][3]. RLE replaces each run of repeated values in the column with a pair of values and counts, encoding how often each value occurs in succession. The total number of value-count pairs is equal to the total number of runs in the column. Given a data type of b bytes and a column with k runs, RLE encodes the column into $k * (b + 4)$ bytes, assuming the use of 32-bit (4 bytes) integers to encode the counts. Figure 2.3 shows an example of RLE.

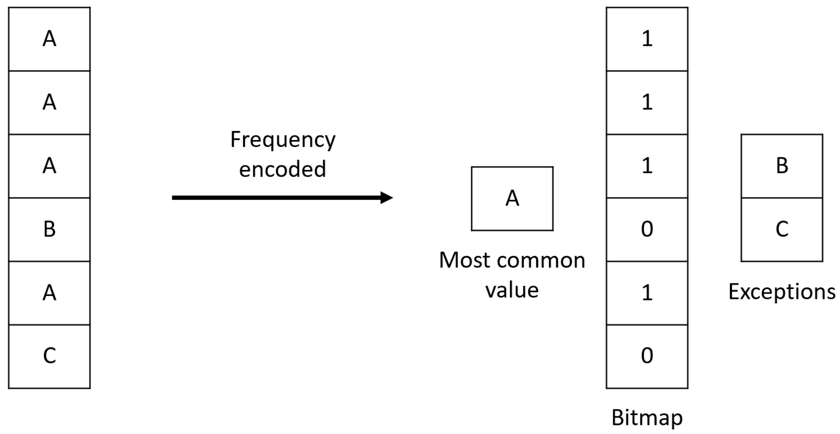

Figure 2.2: A string column is compressed using Frequency Encoding.

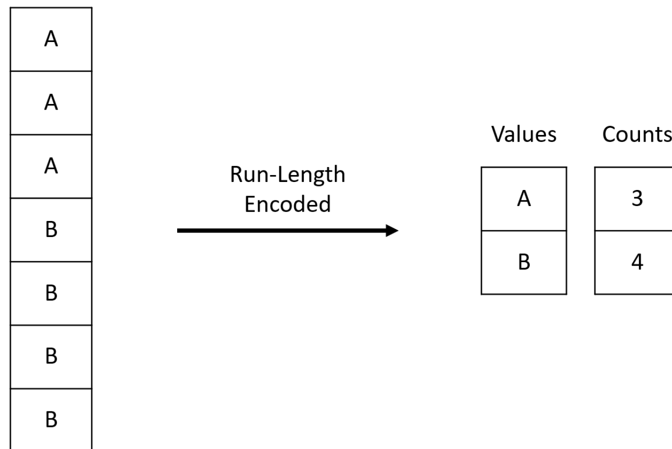

Figure 2.3: A string column is compressed using Run-Length Encoding.

2.1.4 Bit-packing

Bit-packing is a technique to compress integers by omitting leading zero bits [3]. For example, signed 32-bit integers can encode values between a minimum value of -2,147,483,648 (-2^{31}) and a maximum value of 2,147,483,647 ($2^{31} - 1$), but in practice, the values of integers in most columns only use a small fraction of this range. As a consequence, these integers are encoded with many leading zero bits which can be omitted to reduce the total number of bits used to encode the value. Bit-packing uses the minimum number of bits to encode an integer column and stores the number of bits used to decode individual values during decompression. Given a column of integers and a maximum value k, bit-packing uses $\lfloor \log_2 k \rfloor + 1$ bits to store all integers in the column. The smaller the range of integers, the higher the compression ratio achieved by bit packing. To handle outliers and take advantage of local data attributes, bit-packing often compresses small consecutive ranges of values individually, e.g. 32 integers at a time. Figure 2.4 shows an example of bit-packing.

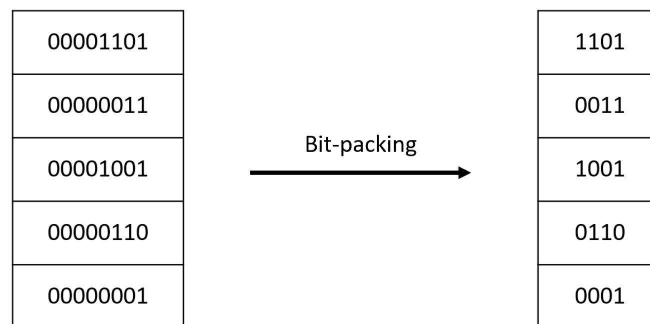

Figure 2.4: An integer column is compressed using bit-packing.

2.1.5 Frame of Reference (FOR)

Frame of Reference is another LWC scheme for integers [8]. The scheme finds the minimum integer value in a column and stores it as the reference value. It then subtracts the reference from all integer values in the column, with the goal of decreasing the values of the integers for improved bit-packing. After subtracting the reference, the resulting values are bit-packed. Frame of reference encoding is most effective when integer values are distributed in a narrow range close to a large reference value. The compression ratio achieved by the final

bit-backing step depends on the size of integers after subtracting the frame of reference value. Figure 2.5 shows an example of FOR encoding.

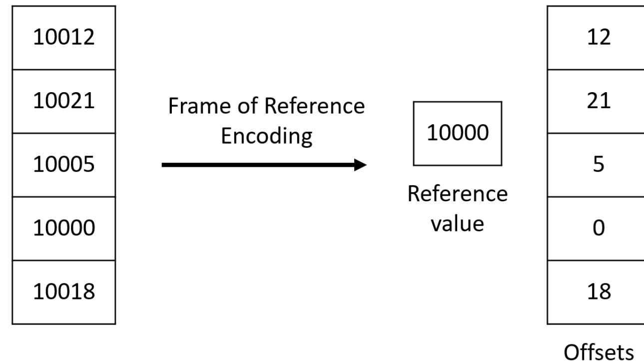

Figure 2.5: An integer column is compressed using Frame of Reference encoding.

Patched Frame of Reference (PFOR) is a variation introduced to make FOR more resistant to outliers [5]. If after subtracting the reference, a resulting value is still larger than a predetermined threshold, it is treated as an outlier and is stored separately. This method further reduces the size of the maximum integer value in the column, leading to higher compression ratios from bit-packing. During decompression, all values are first decompressed regardless of exceptions, after which a second loop over the data patches all exception values [7].

2.1.6 Pseudodecimal Encoding (PDE)

Pseudodecimal Encoding is a recently introduced compression scheme for double data types [3]. As doubles are typically stored in IEEE 754 representation (1-bit sign, 11-bit exponent, 52 bits mantissa), the approaches of other numerical LWC schemes such as bit-packing or FOR are not effective, since the mantissa bits can vary greatly even for numerically close values. PDE makes use of the fact that doubles are often used to encode values for which fixed-point encoding would be sufficient (e.g. prices such as $0.99). Even though some decimal values such as 0.99 can not be represented accurately in IEEE 754 representation (it is stored physically as 0.98999...), the physical value can be efficiently compressed by representation with a pair of integers, the significant digits 99 and the exponent 2. Due to the characteristics of the rounding errors occurring in floating point arithmetic, $99 * 10^{-}2$ results in the bit-wise exact same value as the IEEE 754 floating point representation of 0.99. In this way, a column of doubles can be transformed into two arrays of integer values, which can be further compressed

using integer compression schemes. True floating point numbers that can not be represented in this way are stored separately as exceptions and are not further compressed. PDE is not effective if columns contain many exceptions, as these are not compressed and patching exception values will increase decompression time significantly. Figure 2.6 shows an example of pseudo-decimal encoding.

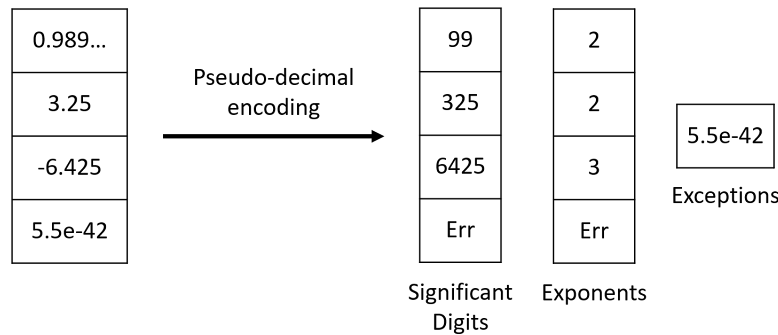

Figure 2.6: A float column is compressed using Pseudo-decimal encoding.

2.1.7 Dictionary Encoding

Dictionary Encoding is a LWC scheme for all data types [4]. It is most effective for columns with only a small number of unique values that occur multiple times in the column. Dictionary encoding creates a separate dictionary containing all the unique values and provides a unique integer key to access each value in the dictionary. The values of the original column are replaced with the dictionary keys of the respective value, resulting in an encoded integer column. This integer column is further compressed with bit-packing. Given a data type of b bytes and a column with n elements and m unique elements, dictionary encoding needs $n * (\lfloor \log_2 m \rfloor + 1)$ bytes to encode the column of integer keys using bit-packing. $m * b$ bytes are additionally needed to store the dictionary itself. Figure 2.7 shows an example of dictionary encoding.

2.1.8 Fast Static Symbol Table (FSST)

FSST is a newly proposed LWC scheme for strings [9]. FSST takes advantage of the fact that strings often share common substrings. Given a group of strings, FSST finds the 256 most common shared substrings of up to 8 characters long and stores these in a fixed-size symbol table. Each substring in the table is assigned a 1-byte symbol to represent the substring. String columns are compressed by

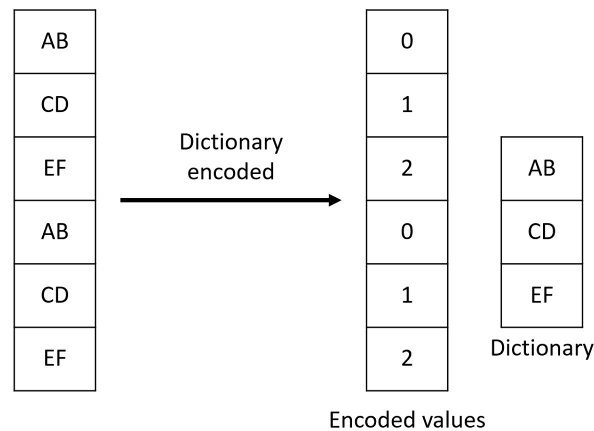

Figure 2.7: A string column is compressed using Dictionary encoding.

replacing all occurrences of these substrings with their corresponding symbols. To decompress the encoded strings, the symbol table is used to replace the symbols with the original substrings. Figure 2.8 shows an example of FSST encoding.

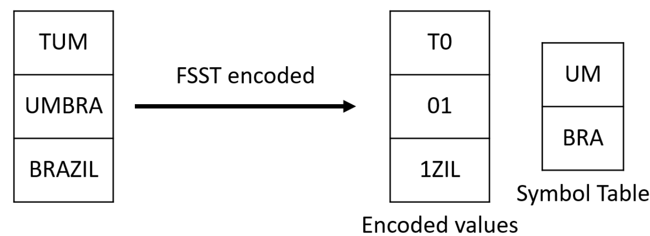

Figure 2.8: A string column is compressed using FSST encoding. "UM" is replaced with the symbol "0", "BRA" is replaced with the symbol "1".

2.2 BtrBlocks

BtrBlocks is a columnar file format for storing relational data, the authors provide an open-source C++ implementation [3]. BtrBlocks compresses data using a set of LWC schemes. Data is first ingested as CSV files and is transformed into an in-memory columnar binary representation. In binary representation, the data

of each column is stored in separate files and each column has a bitmap that marks its null values.

To compress the data, BtrBlocks reads in columns in binary representation and applies LWC schemes to individual columns. During compression, columns are split into row groups with up to 65536 rows, and each row group is compressed and stored individually. BtrBlocks uses a sampling-based algorithm to decide which LWC scheme to apply to a column. The data types that BtrBlocks supports are 32-bit integers, doubles, and strings. BtrBlocks performs a single pass over each column and generates statistics of the data, such as the number of unique values and minimum and maximum values for numerical columns. These statistics are used to further filter out compression schemes that are estimated to be unsuitable for a column. For all schemes that are left, BtrBlocks computes an estimation of the compression ratio that the scheme achieves on a column by using one of two methods. For integer and double columns, BtrBlocks takes a sample of the data and applies the compression schemes to the sample, computing the achieved compression ratio. The samples are generated by randomly picking 10 non-overlapping runs of 64 tuples from the column. For string columns, BtrBlocks relies only on the generated statistics to estimate a compression ratio for the compression schemes. BtrBlocks chooses the scheme with the best compression ratio to apply to the column. If no compression scheme achieves a compression ratio greater than 1, then the column stays uncompressed. If a compression scheme is applied to a column, but the compressed size is larger than the uncompressed size, then the column stays uncompressed.

To improve compression ratios, BtrBlocks uses recursive compression. Whenever the output of compression schemes includes arrays of integers, doubles, or strings, BtrBlocks recursively compresses these data arrays again using the most suitable LWC scheme. For example, applying RLE to an integer column produces two new integer arrays containing the values and the counts of the RLE runs. These two arrays would further be compressed by BtrBlocks. BtrBlocks calls this cascading, and the cascading depth is a parameter that can be set to limit the depth of recursive compression and is set to 2 by default.

Table 2.1 shows which LWC schemes BtrBlocks supports and for which data type they are applicable. BtrBlocks handles null values by keeping Roaring bitmap for all columns. The compression schemes used by BtrBlocks have vectorized decompression algorithms for higher decompression throughput. Tests on the Public BI benchmark show that while combinations of compression algorithms such as parquet + zstd can achieve higher compression ratios (7.06 vs 8.24), BtrBlocks achieved 3.8x faster compression speeds.

Table 2.1: Compression schemes and data types supported by BtrBlocks. Many schemes support cascading compression, where the output can be recursively compressed further.

Scheme	Data Types	Cascading compression
RLE	All	Cascading integer compression for RLE values Cascading integer compression for RLE lengths
One Value	All	-
Dictionary	All	Cascading integer compression for codes Cascading FSST compression for string dictionary
Frequency	All	Cascading compression for exceptions
PFOR	Integer	-
Bit-Packing	Integer	-
FSST	String	-
Pseudodecimal	Double	Cascading integer compression for significant digits and exponents. Cascading double compression for exceptions
Roaring	Bitmap	-

2.3 Public BI Benchmark

The Public BI benchmark [10] is a user-generated benchmark containing real data from public workbooks in Tableau Public. The benchmark contains tabular data in CSV files and was created to be more representative of real-world data than synthetically generated benchmarks such as TPC-H.

The benchmark contains a total of 47 datasets, although multiple datasets have the same or very similar schemas, and some tables within a dataset also have overlapping data. To prevent over-representation of datasets with a large number of tables, we only use the first table of each dataset. We also remove datasets that have very same or very similar schemas, such as IGlocations1 and IGlocations2. The datasets AirlineSentiment and IUBLibrary are left out due to their low number of tuples. From here on, we refer to this subset of the Public BI benchmark. Table 2.2 provides an overview of the tables and column data types in the subset. Since BtrBlocks only supports 32-bit integers, doubles, and floats, data types such as date, time, timestamp, and 64-bit integers are left out and grouped under "other". The subset of the Public BI benchmark has a total of 2223 columns across 36 tables. In total, the data types of the columns are 45% integers, 21% decimal, and 31% string, while 3% of the columns are unsupported by BtrBlocks, including booleans, 64-bit integers, dates, and timestamps.

Table 2.2: Tables of the Public BI benchmark subset, with statistics on column count and data type.

Table	Columns	Integer	Decimal	String	Other
Arade_1	11	2	4	4	1
Bimbo_1	12	10	2	0	0
CityMaxCapita_1	12	10	2	0	0
CMSProvider_1	26	5	7	14	0
CommonGovernment_1	56	9	10	37	0
Corporations_1	27	6	0	21	0
Eixo_1	80	12	0	59	9
Euro2016_1	11	2	2	6	1
Food_1	6	3	1	2	0
Generico_1	43	14	2	26	1
HashTags_1	101	41	10	48	2
Hatred_1	31	8	6	17	0
IGlocations1_1	18	15	0	3	0
Medicare1_1	26	10	6	10	0
MedPayment1_1	28	5	7	16	0
MLB_1	48	28	14	6	0
Motos_1	44	14	2	27	1
MulheresMil_1	81	12	0	60	9
NYC_1	54	4	2	42	6
PanCreactomy1_1	29	5	7	17	0
Physicians_1	28	5	7	16	0
Provider_1	28	7	6	15	0
RealEstate1_1	28	5	4	17	2
Redfin1_1	44	6	30	6	2
Rentabilidad_1	141	18	64	57	2
Romance_1	12	3	2	6	1
SalariesFrance_1	57	6	30	21	0
TableroSistemaPenal_1	27	4	4	19	0
Taxpayer_1	28	7	6	15	0
Telco_1	181	1	174	1	5
TrainsUK1_2	28	4	10	13	1
TrainsUK2_1	37	13	3	13	8
Uberlandia_1	81	15	0	57	9
USCensus_1	519	512	0	6	1
Wins_1	257	196	38	20	3
YaleLanguages_1	30	8	0	20	2
Total	2223	1022	466	735	66

3 Related Work

The following sections give an overview of relevant existing literature on finding and exploiting column correlations in database systems.

3.1 Exploiting Column Correlations

Multiple papers explore opportunities to exploit column correlations in database systems. Even though they do not exploit correlations for compression, the following related work deals with similar challenges such as defining correlation metrics and detecting correlated columns within the large search space. In the following, we give an overview of these papers and discuss their approaches to these challenges.

3.1.1 BHUNT

BHUNT is a proposed scheme for finding algebraic constraints between pairs of numerical columns in database tables [11]. The constraints are used to speed up query processing by expressing them as query predicates and injecting them into incoming queries. BHUNT also supports fuzzy constraints, by storing records that do not conform to the constraint in exception tables. BHUNT modifies queries by incorporating constraints in the form of predicates. This helps query optimizers filter out data partitions that do not fulfill the predicates, speeding up query processing time. In the case of fuzzy constraints, the unmodified query is additionally run against the exception table, and the results are combined with the modified query.

BHUNT searches for constraints between two columns by first generating a set of "key-like" columns and looking for "foreign-key-like" matching columns. Queries are likely to contain predicates referring to such column pairs. To reduce the search space, BHUNT applies a set of pruning rules to these candidate column pairs, with the goal of keeping column pairs likely to generate useful constraints, in the context of speeding up query performance. The constraints are defined based on data types, row count, the number of distinct values and null values, and the existence of column indexes amongst others. To compute

the algebraic constraint between the column pairs, BHUNT uses segmentation and clustering techniques on random samples to create "bump intervals" for the target column, in which most of the values fall into. Values that do not fall into the intervals are considered exceptions and are stored in a separate exception table. The number of samples is used to control the size of the size of bump intervals, and consequently the number of exceptions and the size of the exception table. The most useful constraints are stored and are used during query optimization.

3.1.2 CORDS

CORDS is a tool that builds upon ideas used in BHUNT, also searching for correlations between pairs of columns for use in query optimization [12]. CORDS outputs groups of columns for which it recommends maintaining joint statistics. These statistics can be used by query optimizers to improve selectivity estimates, which typically assume statistical independence of columns. CORDS also extends the functionality of BHUNT by being able to detect correlations between numerical and categorical attributes.

CORDS finds correlations and soft functional dependencies by first generating candidate column pairs, for which a pairing rule exists between values of both columns. For columns of different tables, the pairing rule can be a join predicate. As in BHUNT, for each "key-like" column, CORDS uses samples to search every other column to find matching "foreign-key-like" columns, which contain all the values found in the source column. To reduce the search space, CORDS applies a set of pruning rules based on data types, the number of distinct values, any declared primary key and foreign key relationships, and predicates that were present in past queries. To detect functional dependencies between a column pair, CORDS analyzes the number of distinct values in random samples and estimates the strength of the dependency. To detect correlations between columns, CORDS uses a sampling-based chi-square test to determine whether two columns are correlated. CORDS recommends column groups for which to maintain join statistics based on the computed strength of correlations and functional dependencies. The joint statistics can include information about distinct values, frequencies, and quantiles, and can used by query optimizers to improve selectivity estimates.

3.1.3 DeCoRel

DeCoRel is a method introduced to detect groups of correlated columns for usage in data analytics and database operations [13]. The detected groups of columns

may overlap and consist of columns with mixed data types. To create groups of correlated columns, DeCoRel models pair-wise correlations in a graph in which nodes represent columns, and weighted edges represent correlations between columns. The correlation metric used by DeCoRel is a pair-wise asymmetric metric that combines statistical properties of Shannon entropy for discrete or categorical data with cumulative entropy for real-valued data. An algorithm sorts columns into groups, in which each column is correlated with most of the other columns, given a minimum threshold for the correlation metric. Since the number of column groups may be very large, but the output should also be suitable for manual data analysis, similar groups are merged to create fewer and larger groups. The authors do not give specific use cases for the detected correlated column groups, but state that they are useful for many applications in database systems.

3.2 Column Correlations for Compression

There is very little existing research on exploiting correlations for compression. Two papers we have foudn are *Whitebox Compression* and *CorBit*. Whitebox Compression superficially explores using column correlations in the context of a "Whitebox" compression framework. CorBit exploits column correlations to compress bitmap indexes.

3.2.1 Whitebox Compression

Whitebox Compression is a conceptual model introduced by Ghita et al. for data compression in columnar database systems [14] [15]. Whitebox compression defines physical columns as the physical bytes stored on disk and logical columns as defined by the database schema. Whitebox compression creates mappings between physical and logical columns, consisting of composite functions that contain information on how the logical columns can be recreated by the physical columns. One idea behind Whitebox compression is that this mapping information can be made available to databases' query engines, which can exploit it during query optimization and execution, e.g. for predicate push-downs into the physical columns. Another advantage of Whitebox compression is the compression opportunities created by the functional mappings. Since physical and logical columns are decoupled, multiple logical columns can be represented by one physical column to avoid storing redundant data. On the other hand, a single logical column can also be split (e.g. splitting string prefixes) into multiple physical columns, which can introduce more effective compression opportunities

for the physical columns.

To find a suitable mapping between physical and logical columns, the authors propose a recursive exhaustive algorithm to build an expression tree. Using data samples, a cost model estimates the compression ratio achieved by recursively applying various transformative operators to the data. The algorithm aims to maximize the total compression ratio based on the estimated compression ratio of the cost model. The output of the algorithm is a tree structure with leaf nodes representing physical columns, root nodes representing logical columns, and inner nodes representing logical transformation operators. Whitebox compression further tries to reduce the number of physical columns by finding dictionary mappings between correlated string columns. If a string column can be efficiently reconstructed by another string column, a dictionary mapping, and a number of exceptions, then this physical column is removed and replaced by the dictionary and exceptions. All remaining physical columns are subsequently compressed using conventional LWC schemes.

Although Whitebox Compression exploits column correlations for compression, the work is in the early stages of research and is implemented only as a proof-of-concept in Python. The proof-of-concept applies transformations to the logical columns, but for the final task of compressing the columns, the data is ingested into a Vectorwise database and compressed there. The authors evaluate their proof-of-concept Whitebox compression implementation on the Public BI benchmark and report a 1.92X increased compression ratio over conventional "black-box" compression [14], when only considering the columns for which Whitebox compression was applied. When considering all columns, the improved compression ratio is 1.43X [15]. Unfortunately, an evaluation of how much the column correlations contributed to the total compression improvement is missing. In their evaluation of the benchmark, the authors state that 28% of used operators are related to column correlations, but this metric cannot accurately be used to derive the contribution of these operators towards the achieved compression ratio.

3.2.2 CorBit: Leveraging Correlations for Compressing Bitmap Indexes

Lyu et al. propose a new method of leveraging column correlations to improve compression of bitmap indexes [16]. Bitmap indexes are associated with individual columns and typically each unique value of the column is assigned a bitmap. Each bitmap stores a 0 or 1 bit for each tuple, signifying whether the tuple corresponds to the unique value or not. Common techniques to compress bitmap

indexes include Roaring and Tree-Encoded Bitmaps. The authors propose a new compression scheme for bitmaps called *CorBit*, which leverages correlations between columns to improve bitmap compression even further.

The idea behind CorBit is to improve bitmap compressibility by increasing sparseness within bitmaps. Given two bitmaps A and B, the difference X between them can be computed using XOR. If A and B are very similar (small Hamming distance), X will be a very sparse bitmap. To leverage this for compression, bitmap B is discarded and only its diff-encoded form X is stored. A and X are passed to compression schemes such as Roaring or RLE compression. The sparseness of X will allow these compression schemes to achieve higher compression ratios than if applied to the original bitmap B. To reconstruct bitmap B, bitmaps A and X are first decompressed, then XOR-ed, which results in the original bitmap B.

To find pairs of correlated columns for which this compression method is suitable, the authors designed a new metric called *Total Reduced Popcnt*, which measures by how much the number of 1-bits of diff-encoded bitmaps can be reduced. The metric is highly correlated with the amount of saved space achieved.

The process of compressing bitmaps with CorBit works as follows. Each pair of columns in a table is assigned a contingency table to store the frequency distribution of the values. During a single pass of the table, all contingency tables are updated with the frequency counts. In the next step, these frequency counts are used to calculate the Total Reduced Popcnt metric for each pair of columns. A graph is constructed to model the relationships between all columns. Graph nodes represent columns and weighted edges represent the Total Reduced Popcnt metric of column pairs. A greedy algorithm picks the highest-weighted edges, in an approximation of the optimal dependency graph. Given the final edges, all bitmaps of the source columns are compressed directly by Roaring compression. The bitmaps of the target columns are first diff-encoded with the bitmaps of the source column and are then compressed by Roaring. Evaluated on three datasets, the authors report between 2.4% and 9.1% smaller storage footprint using CorBit, compared to compressing all bitmaps directly with Roaring. The increased latency overhead during decompression of CorBit on the datasets was between around -1% and 12.6%. The added overhead for finding pairs of correlated columns during compression was not evaluated. These columns are expected to be more suitable for CorBit compression and we expect the improvement in storage footprint to be significantly less when considering all columns.

While CorBit shares some similar approaches, our work aims to exploit correlations to compress columns themselves, instead of bitmap indexes.

4 Design and Implementation

Our compression framework is implemented in C++ and is integrated into the BtrBlocks file format, which has a compression framework for single-column LWC schemes. The general steps taken to compress a table are as follows: a table is horizontally partitioned into row groups of 65536 tuples. In one pass over all columns in the row group, statistics of the columns are collected. For every column, a sample of the values and the statistics are used in combination to calculate an estimated compression ratio achieved by every applicable compression scheme on the column. In the case of multi-column compression schemes, this process involves computing the estimated compression ratio for all pairs of columns. A greedy algorithm picks which compression schemes to use based on the estimated compression ratios. The best schemes are used to compress each column, and all relevant metadata is added to a header of the compressed data. Figure 4.1 provides an overview of the compression workflow.

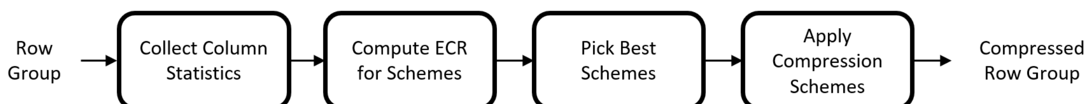

Figure 4.1: Overview of the compression workflow. Columns of a row group are analyzed to collect statistics, which are used in conjunction with samples to compute estimated compression ratios (ECR) for all compression schemes. The best schemes are selected and applied to compress the columns.

The following sections go into more detail on the design and implementation of the individual compression schemes and the surrounding compression framework.

4.1 Single-Column LWC Schemes

For our single-column compression schemes, we use the implementions in BtrBlocks, with some modifications to limit cascading compression. We modified

the schemes so that they more accurately represent regular LWC schemes used by most file formats. This lets us evaluate how much benefit our multi-column compression schemes can provide compared to a conventional compression framework with regular LWC schemes. Compared to the original BtrBlocks schemes as listed in 2.1, the schemes are modified to have cascading behavior as listed in table 4.1. Instead of computing estimated compression ratios by compressing samples of a column, we instead compress the entire column. This effectively gives us 100% accurate estimated compression ratios or single-column schemes and allows us to better evaluate the benefit provided by our multi-column compression schemes.

Table 4.1: The single-column compression schemes of BtrBlocks, modified to represent regular LWC schemes with limited cascading compression.

Scheme	Data Types	Cascading compression
RLE	All	RLE values and RLE lengths produced by encoding are further compressed with bit-packing.
One Value	All	-
Dictionary	All	Integer codes are further compressed with bit-packing. Dictionaries are not further compressed.
Frequency	All	Exceptions are not further compressed.
PFOR	Integer	-
Bit-Packing	Integer	-
FSST	String	-
Pseudodecimal	Double	Significant digits and exponents are compressed with bit-packing. Exceptions are not further compressed.
Roaring	Bitmap	-

Single-column dictionary compression. A few of our multi-column compression schemes will need to dictionary encode columns without compressing the codes. To support this we added our own implementation of a single-column dictionary encoding scheme. This scheme only dictionary-encodes a column and returns the dictionary, but does not compress the encoded integer values. This scheme is also able to take an existing dictionary as input and use it to

dictionary-encode a column.

4.2 Multi-Column LWC Schemes

We developed six multi-column compression schemes that work on pairs of columns to exploit correlations. We present our compression schemes, including how they compress and decompress columns, as well as how we compute their estimated compression ratios. All multi-column compression schemes operate on a source column and target column, where the aim is to reduce the number of bytes needed to encode the target column, by using information from the source column. Since at the time of writing the BtrBlocks file format supports only three data types: 32-bit integers, doubles, and strings, our compression schemes are also limited to these data types. In our C++ implementation, a column of integers a represented by a vector of int32_t, doubles by a vector of doubles, and strings are represented by an array of int32_t offsets followed by a block of bytes containing the string characters, as shown in figure 4.2.

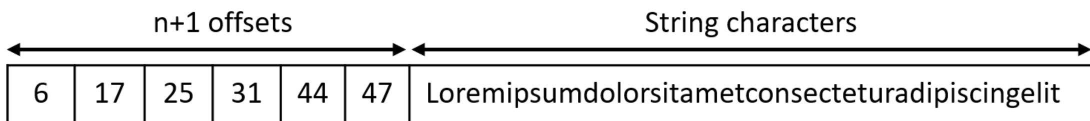

Figure 4.2: An example of an encoded string column containing 6 strings. Given n strings, n+1 integer offsets are used to find the start and end of each string. All string characters are stored in a single chunk after the offsets.

Estimating bit-packing compression ratio. To accurately estimate the size of bit-packed integers, we take into account that our bit-packing scheme compresses integers in blocks of 128 values. Each block is divided into mini-blocks containing 32 integers each. For every mini-block, the number of bits required to bit-pack all integers in the block is computed, based on the largest value in the mini-block. Each mini-block is compressed with its own number of bits, and 4 8-bit unsigned integers containing the number of bits used to encode each mini-block are stored at the beginning of the block. Compared to a naive implementation in which the maximum value of the entire column is used to determine the number of bits used to encode each value, this mini-block-based bit-packing achieves better compression ratios if the maximum values of mini-blocks require fewer bits to encode than the global maximum. To estimate the compression ratio of this

bit-packing implementation, we use the total tuple count of a column, an array of samples, and an array containing the indexes of the samples in the column. Using the total tuple count, we compute the number of blocks and mini-blocks that will be generated. Every block needs 4 bytes to store the number of bits used to encode each of its 4 mini-blocks. To estimate the size of mini-blocks, we use the indexes of each sample to determine which mini-block it belongs to and collect the maximum value for each mini-block. In case there are mini-blocks to which no samples were matched, we set its maximum value equal to the average maximum value of all other mini-blocks. Using these maximum values, we compute the number of bits needed to encode the values in each mini-block and derive the total compressed sizes of every mini-block.

4.2.1 Equality

The *Equality* compression scheme is a multi-column compression scheme that exploits columns that are identical or nearly identical. The columns must have the same data type. In the case of two identical columns, only the source column needs to be stored, and the target column can be reconstructed by simply copying the source column. If two columns are mostly identical, with few values differing from each other, then the target values that deviate from the source values are stored separately as exceptions. Exceptions are stored as two arrays of the same length, one containing the indexes at which the exceptions occur, and one containing the corresponding values of the exceptions. To keep track of null values in the target column, the original nullmap of the target column is kept. The compressed target column consists only of exceptions and the nullmap, whose sizes added together make up the compressed size of the target column. To reconstruct the target column in this case, the source column is copied and all exceptions are patched by replacing the values at the exception indexes with the corresponding exception values. In the Equality scheme, the source column is left unchanged and can be compressed with the best single-column compression scheme for the column. Figure 4.3 shows an example of two columns compressed with the Equality scheme.

The exception values are stored in the data type of the columns, whereas the exception indexes are stored as integers. Since patching exception values can slow down the decompression of the target column, we set the upper limit for the number of exceptions allowed to 10% of the total tuple count. The limit is a parameter that can be configured.

Estimating the compression ratio. We compute the estimated compression ratio the Equality scheme, in order to compare it to the estimated compression ratios achieved by single-column schemes. Since in the Equality scheme, the

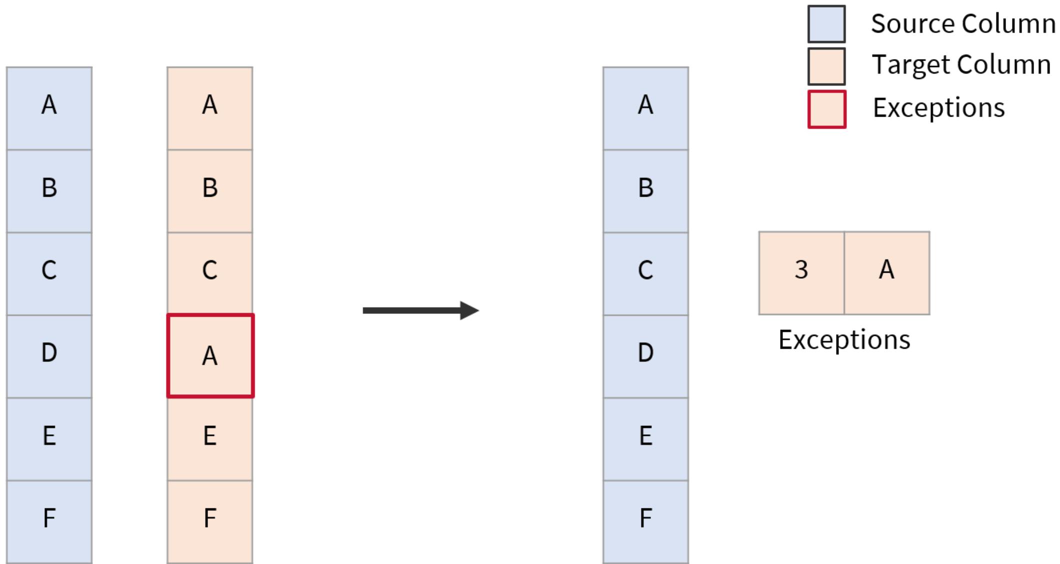

Figure 4.3: A pair of columns encoded with the Equality compression scheme.

source column is compressed with the best single-column scheme, we only need to estimate the compression ratio achieved for the target column. To compute this, we need to estimate the number and size of exceptions, whose sum equals the compressed size of the target column. We take a sample of n values from each column and compare their values for equality. The exception size in bytes is initialized with 0, and for each pair of values that are not equal, we add 4 bytes (size of integer) for the exception index and the size of the exception value (depending on data type and string length). The estimated compression ratio $ECR_Equality$ for the target column is computed by dividing the total size of the target samples by the exception size, as seen in equation 4.11. Exceptions are left uncompressed.

$$target_samples_size_bytes = \sum_{i=1}^{n} sizeof(sample_i) \tag{4.1}$$

$$exceptions_size = \sum_{e \in exceptions} sizeof(integer) + sizeof(e) \tag{4.2}$$

$$ECR_{Equality} = \frac{target_samples_size_bytes}{exceptions_size} \tag{4.3}$$

Compressing with the Equality scheme. To compress two columns with the Equality scheme, all values of the columns are compared to find exceptions. These are stored in the exception value array, and the index at which they occur is stored in the exception index array. The target column is discarded, and only the exception arrays and the nullmap are kept. The exception arrays are left uncompressed and the nullmap is compressed with Roaring. The source column is compressed with the best single-column compression scheme.

4.2.2 Numerical

Our *Numerical* compression scheme exploits linear numerical correlations between two integer columns. If the values of a target column can be computed by applying a linear function to the values of a source column, then the target values do not need to be stored at all, since they can be reconstructed by the source values and the linear function. Our scheme also exploits soft linear correlations for column pairs, so that columns that are not perfectly correlated can still be encoded with this scheme. In these cases the source values can approximately determine the values of the target column, but with small deltas. We compute these deltas and store them in place of the original target values. If the deltas are smaller than the original target values, this will benefit compression techniques such as bit-packing. To take advantage of this, we shift all offsets into the range $[0, \infty)$ and apply bit-packing. The shift is necessary since bit-packing is not effective on negative values. Figure 4.4 shows an example of two columns compressed with the Equality scheme.

Estimating the compression ratio. We compute the estimated compression ratio of the Numerical scheme, in order to compare it to the estimated compression ratios achieved by single-column schemes. Since in the Numerical scheme, the source column is compressed with the best single-column scheme, we only need to estimate the compression ratio achieved for the target column. We only compute the estimated compression ratio for column pairs with a Pearson correlation coefficient with an absolute value greater than 0.7. We use a group of n source samples s_i and target samples t_i to compute it as follows:

$$corr_coef = \frac{\sum_{i=1}^{n}(s_i - \bar{s})(t_i - \bar{t})}{\sqrt{\sum_{i=1}^{n}(s_i - \bar{s})^2 \sum_{i=1}^{n}(t_i - \bar{t})^2}} \tag{4.4}$$

$\bar{s}$ and $\bar{t}$ denote the mean of the source and target samples. After filtering out column pairs with a low correlation coefficient, the slope α and intercept β are

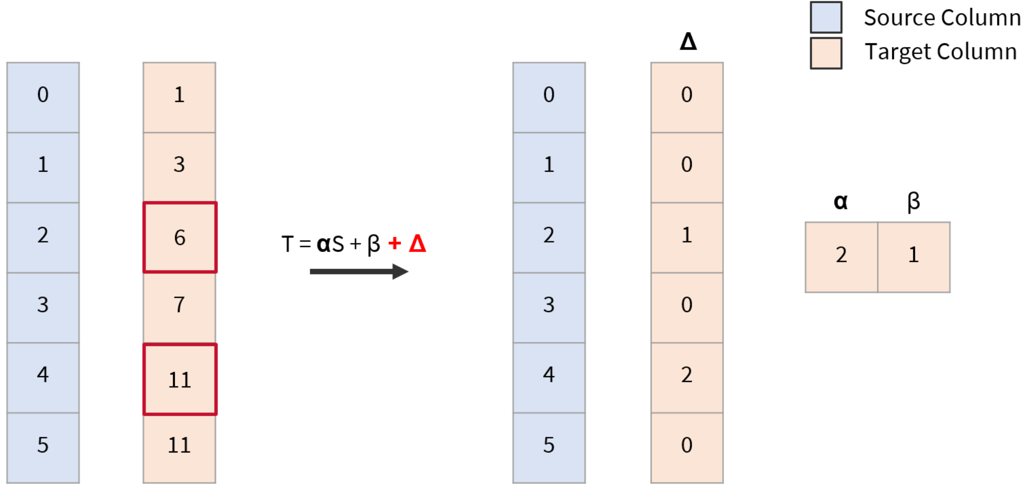

Figure 4.4: A pair of columns encoded with the Numerical compression scheme.

computed based on the same sample as follows:

$$\beta = \frac{\sum_{i=1}^{n}(s_i - \bar{s})(t_i - \bar{t})}{\sum_{i=1}^{n}(s_i - \bar{s})^2} \tag{4.5}$$

$$\alpha = \bar{t} - \beta\bar{s} \tag{4.6}$$

To avoid duplicate computations, we reuse factors from computing the correlation coefficient to compute β. To estimate the compressed size of the target column using the Numerical scheme, the target deltas d_i are computed for the samples:

$$d_i = t_i - \lfloor \alpha s_i + \beta \rfloor \tag{4.7}$$

We compute the estimated size of bit-packing the delta values and use this as the estimated size of the target column. The estimated compression ratio is the original size of the column divided by the estimated compressed size:

$$ECR_{Numerical} = \frac{target_col_size_bytes}{bitpacked_deltas} \tag{4.8}$$

Compressing with the Numerical scheme. To compress a column with the Numerical scheme, we reuse the slope a and intercept b computed on the samples. The target deltas d_i are computed as in equation 4.7. The target deltas replace the

original target values and are compressed further with bit-packing. The slope and intercept are stored in the metadata of the compressed target column. The source column is compressed with the best single-column compression scheme. To decompress the target column, the source column is first decompressed and the offsets stored in the target column are decompressed. Using the slope and intercept stored in the target column's metadata, the target values t_i are then reconstructed as follows:

$$t_i = \lfloor \alpha * s_i + \beta \rfloor + d_i \tag{4.9}$$

4.2.3 1-to-1 Dictionary

The *1-to-1 Dictionary* scheme is designed to compress a target column, whose values can directly be determined by a source column through a dictionary mapping. The scheme can be applied to two columns of arbitrary mixed data types. If the values of a source column always map to the same corresponding values of a target column and these value-pairs occur often, then this mapping can be stored once and the target column does not need to be stored. To soften the criteria needed to apply this scheme, we allow exceptions to this mapping which are stored as separate exception indexes and exception values.

Compressing the 1-to-1 Dictionary scheme. The compression scheme is applied as follows: the source column and target column are both separately dictionary-encoded, whereby each unique value is replaced with an integer code. In a single pass over the source and target column, each unique mapping between source and target codes is stored and their occurrence frequency is counted. For each unique source code, we determine the most frequently mapped target code and store it in a dictionary mapping. In a second pass over the source and target column, all target codes that do not correspond to the code in the dictionary mapping are identified as exceptions. The index and the target code of the exception are stored in separate integer arrays. Since we dictionary encoded the target column, both the mapping dictionary and exception values are reduced to the integer codes. This reduces the sizes of the mapping dictionary and exception values (for double and string target columns). In the case of string columns, the integer codes also allow a more accurate estimation of the mapping dictionary size and exception size using samples by fixing the size of mappings and exceptions. To keep track of null values in the target column, the original nullmap of the target column is kept. The codes for the source and target column are bit-packed, and the exceptions and the mapping dictionary are left uncompressed. Figure 4.5 gives an example of two columns encoded with the 1-to-1 Dictionary scheme.

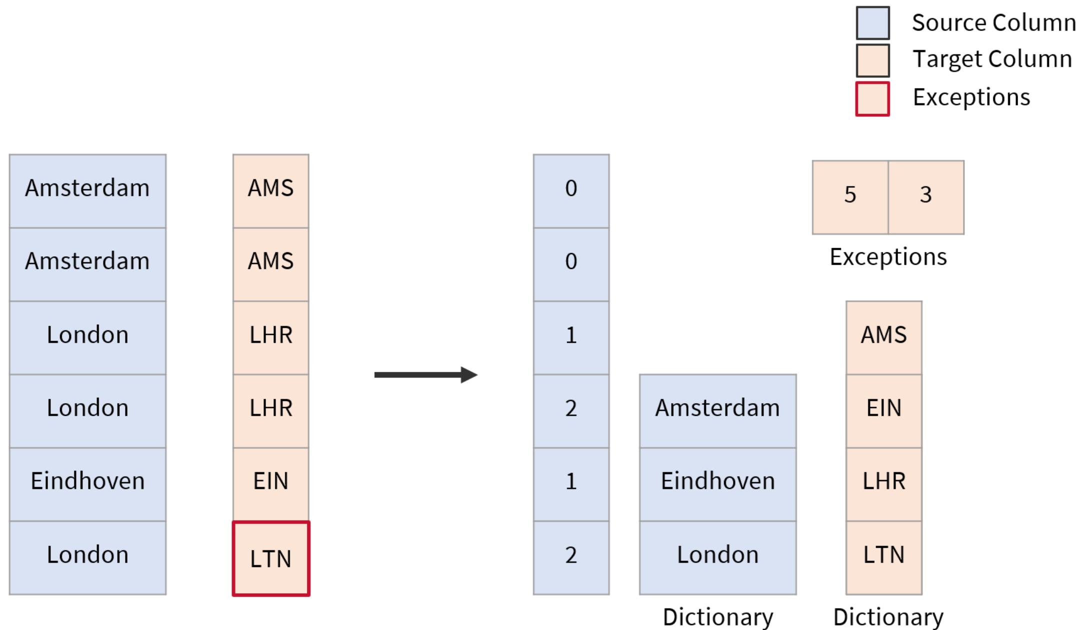

Figure 4.5: A pair of columns encoded with the 1-to-1 Dictionary compression scheme.

The number of entries in the source dictionary and the target dictionary equals the number of unique values in the respective columns. The number of entries in the source-target mapping dictionary also equals the number of unique source values.

Decompressing the 1-to-1 Dictionary scheme. To decompress a 1-to-1 Dictionary-encoded column, the compressed source column must first be bit-unpacked to its integer dictionary codes. The dictionary-encoded target column is recreated by applying the mapping dictionary. All exceptions are patched in the target column, after which the original target values are retrieved by replacing the dictionary codes with the original values using the target dictionary. To fully decompress the source column, the source dictionary is used to replace the source codes with their original values. Since patching exception values can slow down the decompression of the target column, we limit the maximum number of allowed exceptions to 10% of the total tuple count.

Estimating the compression ratio. We compute the estimated compression ratio of the 1-to-1 Dictionary scheme, in order to compare it to the estimated compression ratios achieved by single-column schemes. Since this scheme requires the source column to be dictionary encoded, we need to include both the source and target column in the estimated compression ratio. We compute

the estimated compression ratio with a combination of column statistics and samples. The statistics were generated by a single pass over the columns and include the total number of tuples, the number of unique values, and the total size of unique values. We know the size of the source dictionary and target dictionary since they are equal to the total size of unique values from the respective column statistics. The size of the source-target mapping dictionary is equal to the number of unique source values, multiplied by 4 (size of integer codes). We need the samples only to estimate the number of exceptions. To achieve this, we create a dictionary with all unique mappings between source and target samples, including a counter for how often they occur. For each source sample, we determine the most frequent target sample, and the frequencies of all mappings are summed up and equal the exception count. We extrapolate the number of exceptions we found in the samples to the total number of tuples in the column. For each exception, we store an integer index and an integer code, so the total exception size is the exception count multiplied by 8 bytes. We estimate the compressed size of the source column using the number and size of unique values, and by estimating the bit-packed size of the codes. We leave exceptions and dictionaries uncompressed.

$$\begin{aligned} estimated_compressed_target_size = {} & exception_count * 2 * sizeof(integer) + \\ & target_total_unique_values_size + \\ & source_unique_values_count * sizeof(integer) \end{aligned} \tag{4.10}$$

$$ECR_{1-to-1-Dict} = \frac{source_column_size + target_column_size}{estim_compressed_source_size + estim_compressed_target_size} \tag{4.11}$$

4.2.4 1-to-N Dictionary

The *1-to-N Dictionary* scheme is designed to exploit a correlated column pair in which a 1-to-N mapping exists between unique values of the source and target column. This scheme can be applied to columns of arbitrary mixed data types. For example, a 1-to-N mapping would hold for a source column containing countries and a target column containing cities within the country. For each unique source value, all corresponding N unique target values are gathered and assigned a code starting from 0 to N. Instead of applying dictionary encoding to the target column, which would result in target codes with values between $[0, target_unique_count)$, target codes are instead limited to $[0, N)$, which can

improve compression ratios achieved by schemes such as bit-packing, if $N \ll target_unique_count$.

Compressing the 1-to-N Dictionary scheme. To compress two columns with this scheme, the source column is first dictionary encoded. In a single pass over the source and target column, a mapping dictionary is built. The mapping dictionary is filled by going through source codes in ascending order, and for each source code, all unique target values are added to the mapping dictionary. We build a dictionary containing offsets for each source code. The offsets point to the segment of target values in the mapping dictionary which belong to the source code. Finally, the target column is replaced with integer codes, so that the offset of the source code added to the target code equals the index to access the correct value in the mapping dictionary. Figure 4.6 shows an example of a 1-to-N Dictionary encoding. To keep track of null values in the target column, the original nullmap of the target column is kept. The source and target codes are bit-packed and the dictionaries are kept uncompressed.

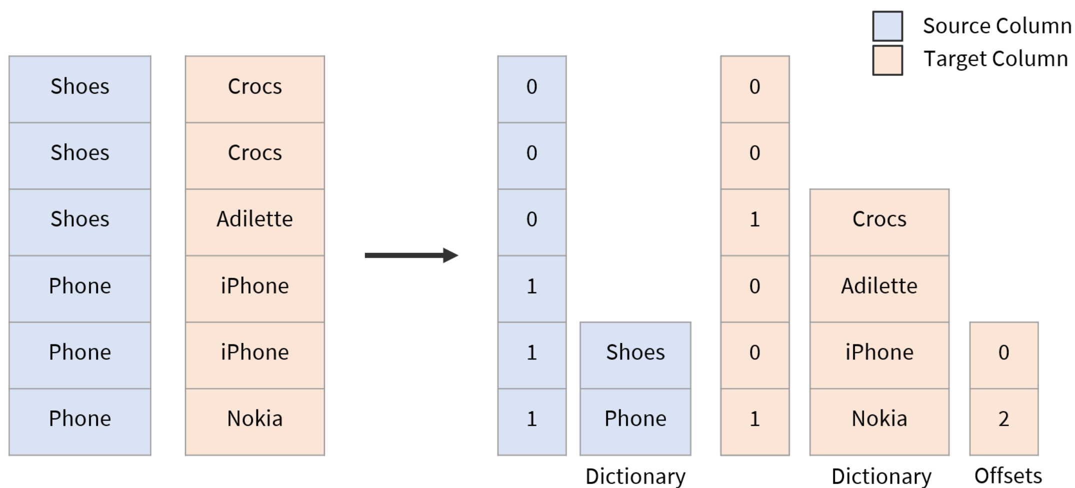

Figure 4.6: A pair of columns encoded with the 1-to-N Dictionary compression scheme.

Decompressing the 1-to-N Dictionary scheme. To decompress the target column, the source column and target column are first bit-unpacked, so that they are in dictionary-encoded form. Each target value is reconstructed by using the integer code in the source column as an index to retrieve the offset from the offset dictionary. This offset is added to the target integer code, which results in the final index used to access the correct target value in the mapping dictionary. To fully decompress the source column, the source dictionary is used to replace

the integer codes with the original values.

Estimating the compression ratio. We compute the estimated compression ratio of the 1-to-N Dictionary scheme, in order to compare it to the estimated compression ratios achieved by single-column schemes. Since this scheme requires the source column to be dictionary encoded, we need to include both the source and target column in the estimated compression ratio. We use pre-collected column statistics are used to estimate the size of the compressed source column, the source dictionary, and the offset dictionary. Samples are needed to compute and estimate the sizes of the target column and target dictionary. The size of the dictionary to dictionary-encode the source is equal to the total size of unique values in the column, which are part of the source column statistics. We compute the estimated size of the source codes by using samples to estimate the bit-packed. The size of the offset dictionary is equal to the total number of unique source values multiplied by the size of integers. To estimate the size of the target column and target dictionary, we need a set of samples. Based on the samples, a mapping is created between each unique source sample, and a vector of all unique target values it maps to. The number of elements in the target dictionary is equal to the number of unique mappings between source and target samples. In the case of an integer or double target column, we can directly compute the size of the target dictionary, by multiplying the number of entries with the size of the data type. In the case of a string target column, we use the statistics to compute the average string length in the column, and use this value multiplied by the number of entries to estimate the target dictionary size. To estimate the size of the target codes, we compute them for a number of samples and use to to estimate the size of the codes after bit-packing. The dictionaries are not further compressed. The final estimated compression ratio of the scheme is computed as in equation 4.15.

$$offset_size = source_unique_values_count * sizeof(integer) \tag{4.12}$$

$$target_dict_size = total_mapping_count * avg_size_data_type \tag{4.13}$$

$$\begin{aligned} estimated_compressed_target_size = offset_size + \\ target_dict_size + \\ estimated_bitpacked_codes_size \end{aligned} \tag{4.14}$$

$$ECR_{1-to-N-Dict} = \frac{source_column_size + target_column_size}{estim_compressed_source_size + estim_compressed_target_size} \quad (4.15)$$

4.2.5 Dictionary-FOR

The *Dictionary-Frame-of-Reference* (DFOR) scheme exploits correlations between a source column of arbitrary data type and an integer target column, in which the target values mapped to unique source values fall into distinct and separate ranges. In this case, every unique source value can be assigned a reference value equal to the minimum target value it maps to. All target values mapped to the source value can be adjusted by subtracting the corresponding reference value. If the original target column is not very suitable to be compressed with a conventional integer compression schemes, this scheme may produce smaller integer values for the target column which will benefit from bit-packing. Figure 4.7 shows an example of DFOR encoded columns.

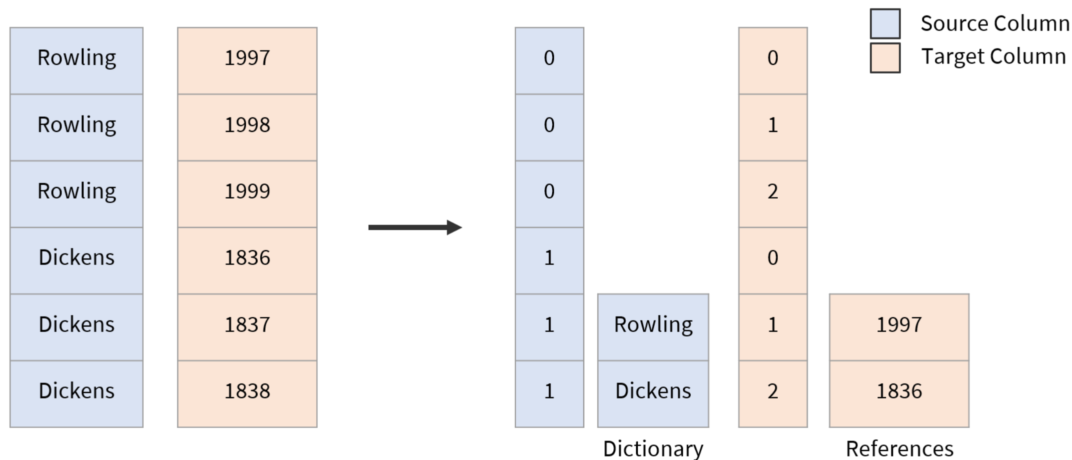

Figure 4.7: A pair of columns encoded with the DFOR compression scheme.

Compressing the DFOR scheme. To compress a pair of columns with the DFOR scheme, the source column is first dictionary-encoded. In a first pass over the source and target column, a dictionary is built that maps each unique source value to the minimum corresponding target value. During a second pass of the source and target column, the minimum target values corresponding to the source values are subtracted from the target values. The source and target columns are further compressed with bit-packing.

Decompressing the DFOR scheme. To decompress the DFOR scheme, the source column is bit-unpacked to its dictionary-encoded form. The values in the source column are used to access the correct frame-of-reference values, which are added to the target values to reconstruct the original target values. To fully decompress the source column, the source dictionary is used to replace the codes with the original source values.

Estimating the compression ratio. To estimate the compression ratio achieved by the DFOR scheme, pre-collected column statistics and samples are used in conjunction. Since this scheme requires the source column to be dictionary encoded, we need to compute the estimated compression ratio of the source and target column combined. For the size of the source column, the estimated compression ratio can be calculated by using samples to compute the estimated bit-packed size. To estimate the bit-packed size of the target column, we need to estimate how large the encoded integer values will be. We apply the DFOR encoding to a sample and use it to estimate the bit-packed size. The size of the frame-of-references equals the number of unique source values, multiplied by the size of integers. The total estimated compression ratio is computed as in equation 4.18.

$$references_size = source_unique_values_count * sizeof(integer) \tag{4.16}$$

$$\begin{split} estimated_compressed_target_size = references_size + \\ estimated_bitpacked_codes_size \end{split} \tag{4.17}$$

$$ECR_{DFOR} = \frac{source_column_size + target_column_size}{estim_compressed_source_size + estim_compressed_target_size} \tag{4.18}$$

4.2.6 Dictionary Sharing

The *Dictionary Sharing* scheme uses a shared dictionary to dictionary encode two columns, and can be applied to two columns of equal data type. If the source column and target column have an overlapping set of unique values, then the combined dictionary will be smaller than the total size of two individual dictionaries. The amount of bytes saved is equal to the total size of shared unique values between the columns. To compress a pair of columns with the Dictionary

Sharing scheme, we need to compute the union of both sets of unique values. For both columns, we use the maps from the column statistics which already map each unique value to a dictionary code. For each unique target value, we check whether it is contained in source column's map of unique values. If it is not present, the target value is added to the map and given an integer code equal to the new size of the map. This combined dictionary is used to dictionary encode both the source and target column, as seen in Figure 4.8. The dictionary is stored only with the source column, but is used to decode both the source and target column during decompression.

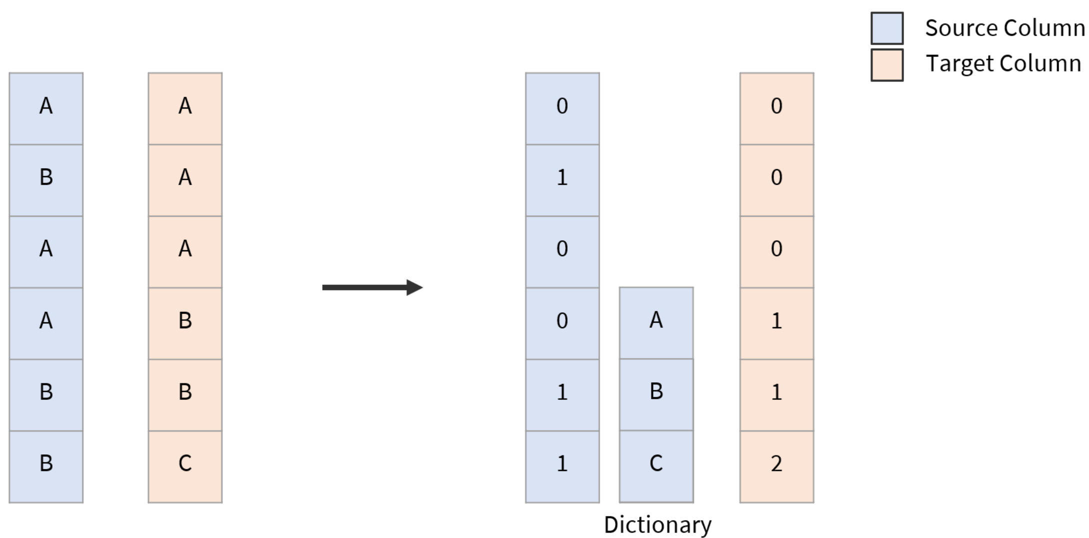

Figure 4.8: A pair of columns encoded with the Dictionary-Sharing compression scheme.

Estimating the compression ratio. To estimate the compression ratio achieved by applying the Dictionary Sharing scheme to a pair of columns, we use the pre-collected column statistics which include a set of all unique values of both columns. Since this scheme required the source column to be dictionary encoded, the estimated compression ratio of the scheme needs to be calculated on both the source and target columns combined. To compute the size in bytes of the combined dictionary, we take the total size of unique values of the source column, and for each unique target value not contained in the source column, add the size of the target value. The estimated size of the source and target codes can be computed using samples to estimate the bitpacked size. The final estimated compression ratio is computed as in equation 4.19.

$$ECR_{DShare} = \frac{source_column_size + target_column_size}{estim_compressed_source_size + estim_compressed_target_size} \quad (4.19)$$

4.3 Compression Framework

After horizontally partitioning a table into row groups, each row group is compressed individually. In the first step, the compression framework iterates over each column in the row group. For each column, it generates statistics and uses the statistics as well as a set of samples to compute an estimated compression ratio for each applicable single-column compression scheme. The column statistics are used in combination with samples to compute the estimated compression ratios of both single-column and multi-column compression schemes. The statistics collected for each column differ between numerical columns (integer and double) and string columns.

Statistics for integer and double columns include:

- Map containing all unique values. Each unique value is mapped to an occurrence counter and a unique integer id.
- Minimum value.
- Maximum value.
- Total number of tuples.
- Number of null values.
- Number of unique values.
- Average length of value runs.

Statistics for string columns include:

- Map containing all unique strings. Each unique string is mapped to a unique integer id.
- Total size in bytes of the column, including both offsets and strings.
- Total string size in bytes of the column, starting from the end of offsets to the end of the column.

- Total unique strings size in bytes. The sum of the length of all unique strings without offsets.
- Total number of tuples.
- Number of null values.
- Number of unique values.

4.3.1 Finding Correlated Columns

After generating statistics and computing the estimated compression ratios for single-column compression schemes, we further move on to finding correlations to exploit with multi-column compression schemes. Given n columns and m multi-column compression schemes, the total search space to find correlated column pairs for compression is $\mathcal{O}(n^2 * m)$. Our goal is to find multi-column compression schemes that achieve higher compression ratios than applying single-column schemes to individual columns. To do this we iterate over all column pairs and compute the estimated compression ratio of all applicable multi-column compression schemes. Compression schemes are applicable if the data type requirements are met, as listed in table 4.2. The methods used to compute the estimated compression ratios for each scheme are described in section 4.2.

Table 4.2: Data type requirements for multi-column compression schemes.

Scheme	Data Types
Equality	Any matching pair of data types
Numerical	Any source column data type, integer target column type
1-to-1 Dictionary	Any two data types
1-to-N Dictionary	Any two data types
DFOR	Any source column data type, integer target column type
Dictionary-Sharing	Any two data types

Pruning Rules. To reduce the quadratic search space as much as possible, we skip compression schemes for column pairs based on pruning rules defined for each scheme as listed in table 4.3. These pruning rules are set to filter out as many unpromising column pairs and schemes as possible using column statistics while trying to avoid pruning column pairs and schemes which have

the potential to achieve high compression ratios. The efficacy of these pruning rules is discussed in chapter 5. As a final tool to reduce the search space for tables with a large number of columns n, we introduce a window size parameter $w < n$, which adds a restriction to the maximum allowed distance between two columns based on column indexes in the table. This limits the search space to $\mathcal{O}(w^2 * m)$. In our implementation, we set $w = 100$.

Table 4.3: Column-pair pruning rules for multi-column compression schemes using column statistics.

Scheme	Pruning Rules
Equality	Skip column pair, if the difference between the number of unique values is larger than the allowed threshold for exceptions.
Numerical	Skip column pair, if the difference between the number of unique values is larger than 0.3% of total tuple count.
1-to-1 Dictionary	Skip column pair, if the number of unique values of either source or target column exceeds 15% of total tuple count. Skip column pair, if the difference between the number of unique values is larger than the allowed threshold for exceptions.
1-to-N Dictionary	Skip column pair, if the number of unique values of either source or target column exceeds 15% of total tuple count.
DFOR	Skip column pair, if the number of unique values of source column exceeds 10% of total tuple count.
Dictionary-Sharing	Skip column pair, if the number of unique values of either source or target column exceeds 25% of total tuple count.

Sampling. Since the search space for multi-column schemes is quadratic in the number of columns, the number of estimated compression ratios that are computed is very high. Given a sample size of s, the algorithms to compute the estimated compression ratios for the Equality and Numerical scheme have a time complexity of $\mathcal{O}(s)$, and $\mathcal{O}(s \log s)$ for all other multi-column schemes. We try to keep the sample size for computing the estimated compression ratios as small as possible. We add two parameters to control the sample size: the number

of sample runs and the size of each sample run. The total sample size is the product of these parameters. A run of samples refers to a range of consecutive values sampled from the column, compression schemes that take advantage of patterns between consecutive values need runs of samples to effectively estimate compression ratios. Given parameters *sample_runs* and *run_size*, a column is divided into as many equal-sized windows as the number of *sample_runs*, and *run_size* consecutive values are sampled with a random offset so that the entire run is contained within the window. This ensures that the sample runs are non-overlapping.

4.3.2 Choosing Correlated Compression Schemes

Building a correlation graph. While iterating over all pairs of columns, we compute the estimated compression ratio for every applicable multi-column scheme. To decide whether the estimated compression ratios are good enough to consider using the scheme, we compare the expected compression ratio of the multi-column scheme with the expected compression ratios of the best single-column schemes for the two columns. Only if the multi-column scheme is estimated to compress the columns better than the single-column schemes by a minimum margin, do we keep the multi-column compression scheme for further consideration. For schemes with exceptions, the estimated number of exceptions also cannot exceed the maximum exception count threshold, which is set as a parameter. If these conditions are fulfilled, the multi-column compression scheme is added to a correlation graph which contains all multi-column schemes that are estimated to improve the compression ratio over single-column schemes. The directed graph consists of n nodes representing each column of the table, while weighted edges represent multi-column compression schemes between two columns, pointing from the source to the target column. The edge weights represent the estimated number of bytes that are saved by using the multi-scheme compression scheme instead of single-column compression schemes. Figure 4.9 shows an example of a correlation graph.

Picking the final schemes. The correlation graph may contain a high number of multi-column compression schemes that are not compatible with each other for compression. We implement a greedy algorithm to pick a subset of the edges to maximize the number of bytes saved compared to compressing with single-column compression schemes. This subset of edges forms the final correlation graph which contains the final compression schemes we will use to compress the columns. The algorithm picks the highest-weighted edges that conform to the following restrictions until none are left:

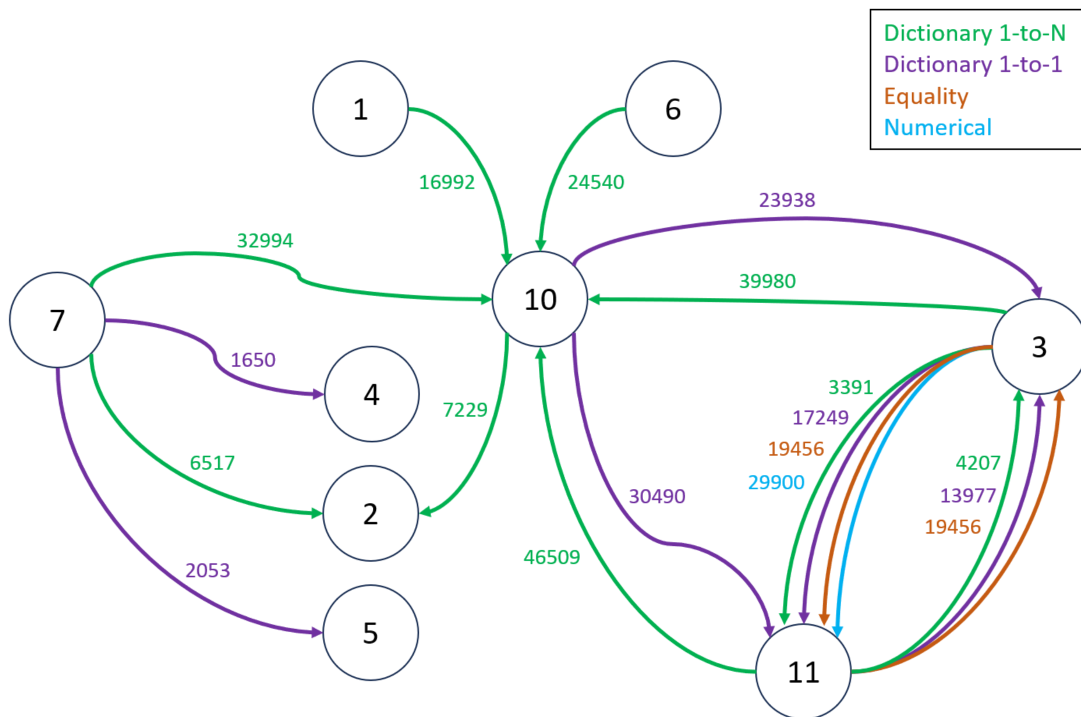

Figure 4.9: An example of a correlation graph. Nodes represent columns and weights on the edges represent the estimated amount bytes saved by using the multi-column scheme over single-column schemes. For example, the 1-to-N Dictionary scheme can be applied between the source column 7 and target column 10, and would save an estimated 32994 bytes, compared to using the best single-column schemes on both columns.

- The final correlation graph may only contain paths of length 1. This prevents chained correlations and ensures that at most only two columns need to be accessed to compress and decompress any column.
- Every node in the final correlation graph may at most only have one input edge.
- Every node may only have one outgoing edge representing a Dictionary Sharing scheme. This is to simplify the compression process for the Dictionary Sharing scheme, as one source column sharing a dictionary with multiple target columns would add complexity and decrease the efficiency of the scheme.

Figure 4.10 shows an example of the final edges chosen by the greedy algorithm.

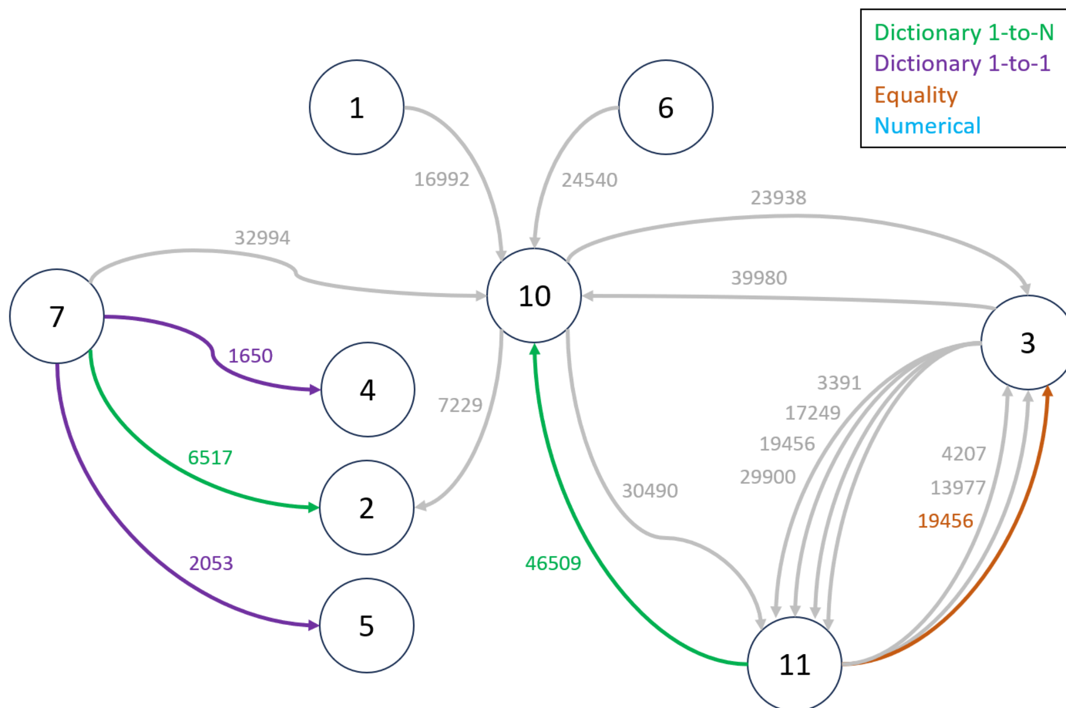

Figure 4.10: The final edges chosen by the greedy algorithm, applied to the correlation graph shown in Figure 4.9. The algorithm picks edges greedily in an attempt to maximize the number of bytes saved by using multi-column schemes. The order of edges picked is $(11 \rightarrow 10)$, $(11 \rightarrow 3)$, $(7 \rightarrow 2)$, $(7 \rightarrow 5)$, $(7 \rightarrow 4)$.

4.3.3 Compression

Compression of a row group happens in two stages. First, all multi-column compression schemes are applied, after which all remaining uncompressed columns are compressed with single-column compression schemes. Since multiple multi-column compression schemes can share the same source column, and different compression schemes have different requirements on how the source column should be encoded, all compression schemes using the same source column need to be taken into consideration during compression.

Order of applying compression schemes. We need to split the multi-column compression schemes into three categories based on how they require the source column to be encoded:

- **Non-Dictionary-Encoding schemes: Equality and Numerical.** To apply these compression schemes, the source column needs to be in its original state with no encodings applied.
- **Regular-Dictionary-Encoding schemes: Dictionary 1-to-1, Dictionary 1-to-N, and DFOR.** To apply these schemes, the source column first needs to be dictionary encoded.
- **Dictionary Sharing scheme.** To apply this scheme, the source column also needs to be dictionary encoded, but the dictionary needs to be extended to include unique values of the target column.

To avoid encoding and compressing the source column multiple times for each multi-column scheme using the source column, we define a strict order for all compression operations:

1. **Apply all non-dictionary-encoding schemes.** All multi-column schemes that need the source column with its original values are applied. Specifically, this step includes encoding and compressing the target column. After applying each scheme, use the compression graph to check if there are any more multi-column compression schemes that have not yet been applied and share the same source column. If yes, the source column is not compressed, if no, the source column is compressed with the best single-column compression scheme according to the estimated compression ratios.
2. **Apply all dictionary Sharing schemes.** All Dictionary Sharing compression schemes are applied. Specifically, dictionary-encoding the source and target columns, as well as compressing the encoded target column. After applying each scheme, use the compression graph to check if there are any more multi-column compression schemes that have not yet been applied

and share the same source column. If yes, the encoded source column is not compressed, if no, the source column is compressed with an integer compression scheme.

3. **Apply all regular-dictionary-encoding schemes.** All regular-dictionary-encoding compression schemes are applied. At this point, the source column may already be dictionary encoded by a previously applied compression scheme. If this is the case, dictionary encoding the source column can be skipped and only the target column needs to be encoded and compressed. After applying each scheme, use the compression graph to check if there are any more multi-column compression schemes that have not yet been applied and share the same source column. If yes, the encoded source column is not compressed, if no, the source column is compressed with an integer compression scheme. In the case that we need to compress the source column and it was already dictionary encoded by a previous scheme, we need to retrieve the dictionary originally used to encode it and store it together with the compressed source column.

Reversing Compression Schemes. Since the estimated compression ratios for the multi-column compression schemes are computed using samples, the accuracy of the estimations depends greatly on the sample size and how well the samples represent the characteristics of the columns. We want to avoid cases in which inaccurate estimated compression ratios lead us to choose multi-column schemes which actually compress worse than single-column schemes. We added a mechanism to discard a multi-column scheme during compression and compress the columns with single-column schemes instead. Reversing the multi-column is done as follows:

- **When to trigger scheme reversal.** After applying a multi-column compression scheme, the size of the compressed columns is used to re-evaluate whether the final compressed size is larger than the estimated size achieved by using the best single-column compression schemes on the source and target column. If the size is larger by a minimum margin, the multi-column compression scheme is reversed.

- **Target column.** To reverse the scheme, the compressed target column is discarded and the original target column is kept in place.

- **Source column.** If the current multi-column scheme is the first scheme using the source column and the source column has been dictionary encoded, the dictionary encoding is discarded and the original source column is kept in place. If the source column was dictionary encoded and is used

by a previously applied multi-column scheme and there are no further multi-column schemes using the source column that need to be applied, then the encoded source column is bitpacked to complete the compression of the source column. In all other cases, nothing needs to be done for the source column.

Single-column compression. After all multi-column compression schemes have been applied, all columns that have been left uncompressed are compressed with the best single-column compression scheme, which we derive from the estimated compression ratios previously computed for all single-column schemes.

4.3.4 Decompression

In many cases, systems may only need to access and decompress individual columns of a table, and some systems may even support working directly on compressed data [17]. This section discusses our procedure for efficiently decompressing all columns of a row group, but it can be adapted and applied to decompressing only a subset of columns. When decompressing all columns of a row group, the decompression procedure should consider that multiple multi-column schemes may use the same source column. Multi-column schemes in which the source column is dictionary compressed need first bit-unpack the source codes without dictionary-decoding the codes, in order to reconstruct the target column. In contrast, multi-column schemes without dictionary-encoded source columns require the source column to be fully decompressed, in order to reconstruct the target column. To take this into account and to minimize decompression work, we decompress a row group in four stages:

1. Decompress all columns that are not part of a multi-column compression scheme and are compressed only with single-column schemes. Additionally, all dictionary-encoded columns that are a source column of a multi-column scheme are bit-unpacked to retrieve the codes dictionary codes. The codes are not decoded back to the original values at this stage.

2. Decompress all columns that are target columns of a multi-column dictionary compression scheme. The previous step ensures that by this point the integer codes of the dictionary-encoded source columns are already bit-unpacked.

3. Fully decode all dictionary encoded columns whose integer codes were bit-unpacked in step one.

4. Decompress all target columns that were compressed with multi-column non-dictionary-encoding schemes. The previous steps ensure that by this point the source columns have been fully decompressed to its original values.

4.3.5 Multi-Row Group Compression

Previous sections have discussed the compression of individual row groups, this section discusses optimization opportunities presented when compressing a batch of row groups from the same table.

Since we expect that most column correlations hold over an entire column and are not limited to a single row group, correlations found in one row group should also be applicable to other row groups. If this is the case, then the multi-column schemes found in one row group can also be reused in other row groups as well. To take advantage of this, we introduce two mechanisms to share the multi-column compression schemes found in the first row group of a table with subsequent row groups of the same table. This allows us to amortize the cost of finding correlated columns as described in section 4.3.1 over multiple row groups.

Sharing final schemes. Our first method involves sharing all multi-column compression schemes that were picked as the final schemes from the compression graph. After compressing the first row group, the final schemes are extracted, including the type of the multi-column compression scheme and the source and target column indexes. For all subsequent row groups of the same table, the process of finding correlated columns is altered. Instead of finding beneficial multi-column schemes for all column pairs and schemes, the search space is reduced to include only the column pairs and schemes as given by the final schemes of the first row group. Only for these schemes, the estimated compression ratio is computed using the statistics and samples of the new row group, and it is re-evaluated whether they outperform the single-column compression schemes. By re-evaluating the estimated compression ratios, we allow the schemes to be discarded in case that the correlations do not hold for the current row group. All schemes that are estimated to achieve higher compression ratios than single-column schemes are added to the correlation graph of the new row group. Since these multi-column schemes were all part of the final schemes of the first row group, they are guaranteed to be compatible and can directly be used as the final schemes for the new row group.

Sharing all schemes in the correlation graph. As a less aggressive alternative, a second method involves sharing all multi-column compression schemes that were added to the correlation graph of the first row group, instead of only the

final picked schemes. These are all schemes that were estimated to achieve higher compression ratios than using single-column compression schemes. For all subsequent row groups of the same table, the search space of finding correlations between columns is reduced to only the column pairs and schemes which were added to the correlation graph of the first row group. For all these schemes, the estimated compression ratio is re-computed with the statistics and samples of the new row group, and only those estimated to achieve higher compression ratios than single-column schemes are added to the correlation graph of the new row group. To pick the final schemes used for compression, the greedy scheme picking algorithm described in section 4.3.2 is run on the correlation graph of the new row group.

4.4 Compressed column format

The general format of compressed columns is shown in figure 4.11. The format contains:

- Multi-Column Scheme: an identifier encoding whether a multi-column compression scheme was used to compress the column and which scheme was used.
- Column Data Type: identifier encoding the original data type of the column.
- Source Column-ID: if a multi-column scheme was used to compress the column, this contains the column-ID of the source column of the scheme. The current column is always the target column of the scheme.
- Original Column Size: size in bytes of the original uncompressed column.
- Single-Column Data Offset: offset in bytes to access the data chunk containing the compressed column and metadata used by the single-column compression scheme.
- Data: split into a chunk containing data used by the multi-column compression scheme, and a chunk containing data used by the single-column compression scheme. The multi-column chunk contains data used by the multi-column compression scheme to decompress the target column of the scheme, such as all exceptions and dictionaries. The single-column chunk contains a header with metadata about the single-column compression scheme used, the compressed values itself, the compressed nullmap, and any other data used by the single-column compression scheme.

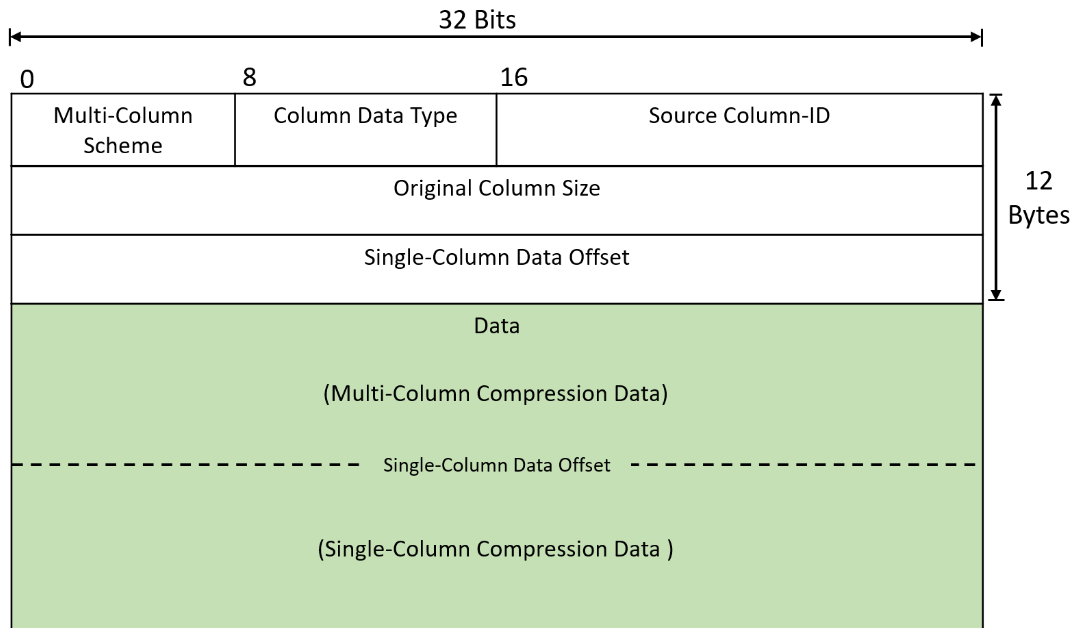

Figure 4.11: Format of a compressed column, including a fixed size header and variable sized data chunk.

The following sections discuss the format of the data stored in the *Multi-Column Compression Data* chunk and *Single-Column Compression Data* chunk shown in figure 4.11.

4.4.1 Single-Column Scheme Compressed Format

The *Single-Column Compression Data* of all values that are compressed with single-column compression schemes is stored in the format as shown in figure 4.12. This includes both compressed column values and other data arrays generated by compression schemes such as exceptions, dictionaries, and RLE values and lengths. Value chunks that are left uncompressed are also stored in this format, in this case, the scheme type is set to "uncompressed". The format contains:

- Scheme Type: identifier encoding which single-column compression scheme was used.
- Nullmap Type: identifier encoding whether the nullmap is filled with only ones or zeros, or the bits were flipped for better Roaring compression.
- Data Type: identifier encoding the data type of the values in the chunk.

- Nullmap Offset: offset in bytes to access the compressed nullmap.
- Tuple Count: number of tuples contained in the chunk.
- Data: the compressed nullmap for the chunk and the compressed values, which contain all data output of the used single-column compression scheme.

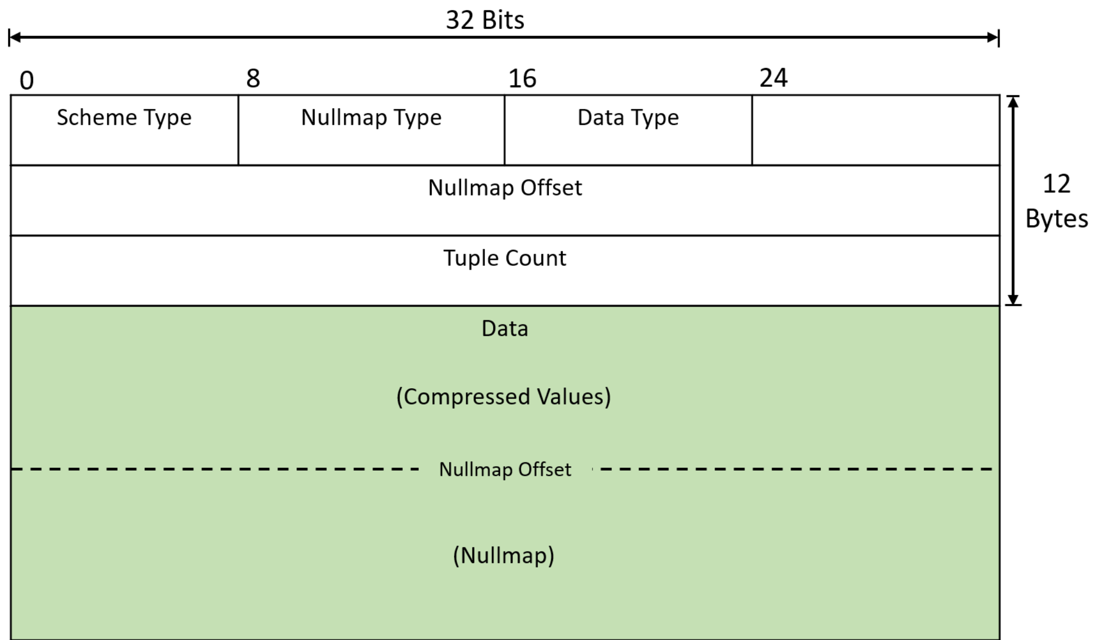

Figure 4.12: Format of a value chunk compressed with single-column compression schemes, including a fixed size header and variable sized data chunk.

4.4.2 Compressed Equality Scheme Format

The *Multi-Column Compression Data* of all column chunks that were compressed with the Equality Compression Scheme is stored in the format as shown in figure 4.13. The format contains:

- Exceptions Offset: offset in bytes to access the exceptions.
- Data: one data chunk containing the nullmap of the target column, followed by a chunk containing all exceptions.

Since the column values are replaced with exceptions, the *Single-Column Compression Data* chunk is empty.

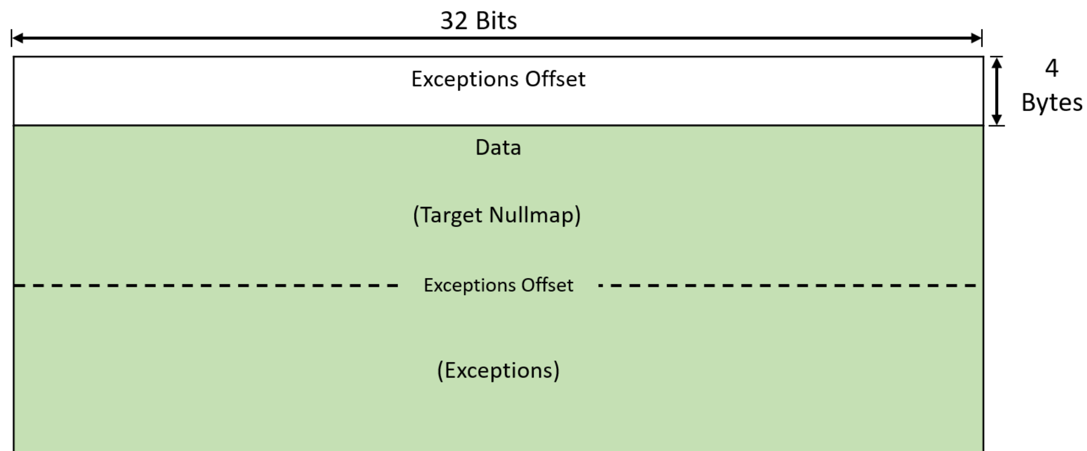

Figure 4.13: Format of a column compressed with Equality compression schemes, including a fixed size header and variable sized data chunk.

4.4.3 Compressed Numerical Scheme Format

The *Multi-Column Compression Data* of all columns that were compressed with the Numerical Compression Scheme is stored in the format as shown in figure 4.14. The format contains:

- Slope: float value containing the slope of the linear correlation.
- Intercept: float value containing the slope of the linear correlation.
- Reference Value: integer value containing the reference value by which the encoded target values are shifted.

The *Single-Column Compression Data* chunk contains the compressed target values generated by the scheme.

4.4.4 Compressed 1-to-1 Dictionary Scheme Format

The *Multi-Column Compression Data* of all columns that were compressed with the 1-to-1 Dictionary Compression Scheme is stored in the format as shown in figure 4.15. The format contains:

- Target Dictionary Offset: offset in bytes to access the dictionary for decoding the dictionary-encoded target column.
- Mapping Dictionary Offset: offset in bytes to access the dictionary used to recreate the target column codes from the source column codes.

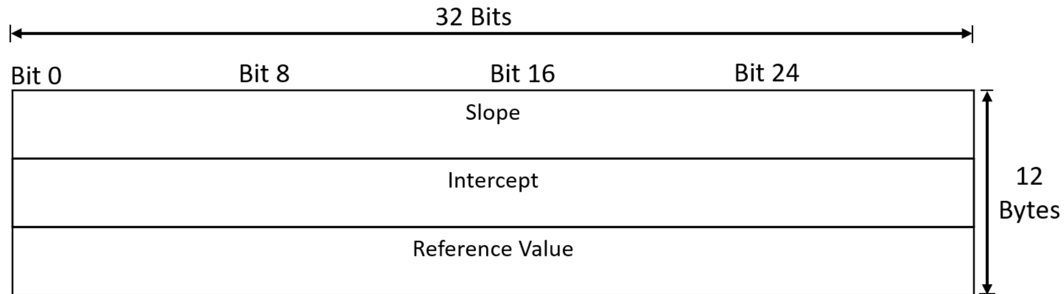

Figure 4.14: Format of a column compressed with Numerical compression schemes.

- Exceptions Offset: offset in bytes to access the exceptions.
- Data: contains data chunks for the nullmap of the target column, the target dictionary, the mapping dictionary, and the exceptions.

Since the column values are replaced with dictionaries and exceptions, the *Single-Column Compression Data* chunk is empty.

4.4.5 Compressed 1-to-N Dictionary Scheme Format

The *Multi-Column Compression Data* of all columns that were compressed with the 1-to-N Dictionary Compression Scheme is stored in the format as shown in figure 4.16. The format contains:

- Mapping Dictionary Offset: offset in bytes to access the dictionary used to retrieve the target column values from the source column codes.
- Data: contains data chunks for the dictionary containing the offset for each source code and the data chunk for the mapping dictionary.

Since the column values are replaced with dictionaries and exceptions, the *Single-Column Compression Data* chunk is empty.

4.4.6 Compressed DFOR Scheme Format

The *Multi-Column Compression Data* of all columns that were compressed with the DFOR Compression Scheme is stored in the format as shown in figure 4.17. The format contains:

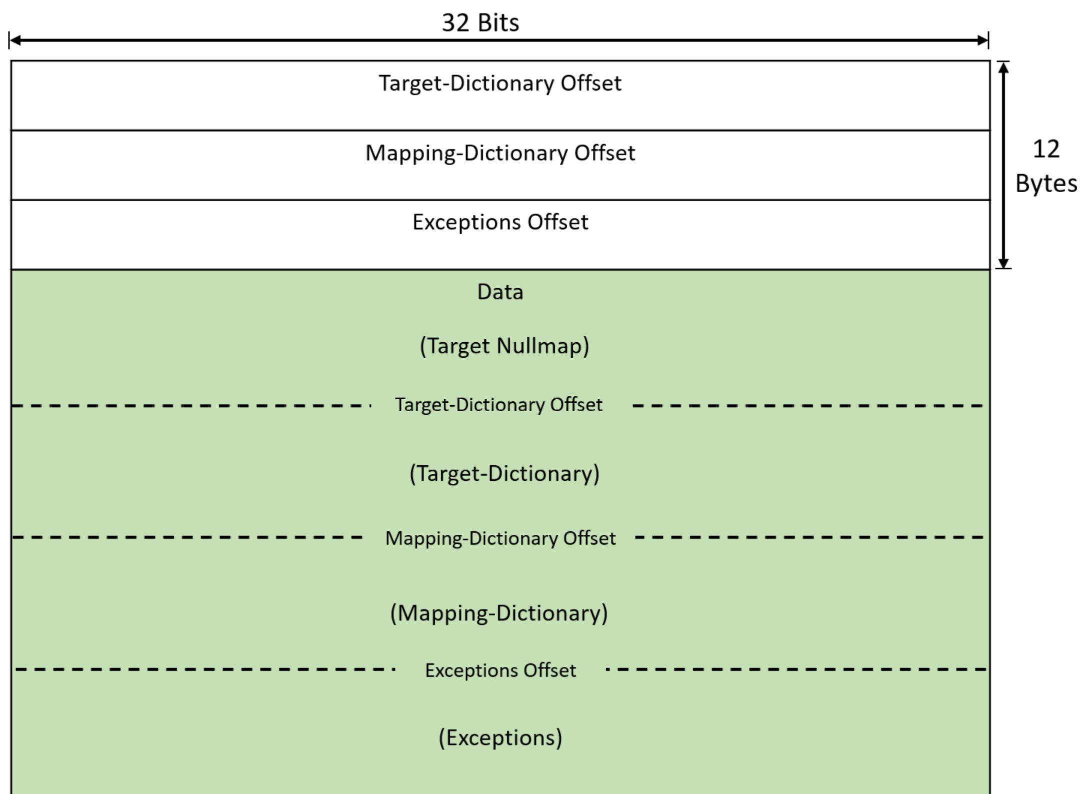

Figure 4.15: Format of a column compressed with 1-to-1 Dictionary compression schemes, including a fixed size header and variable sized data chunk.

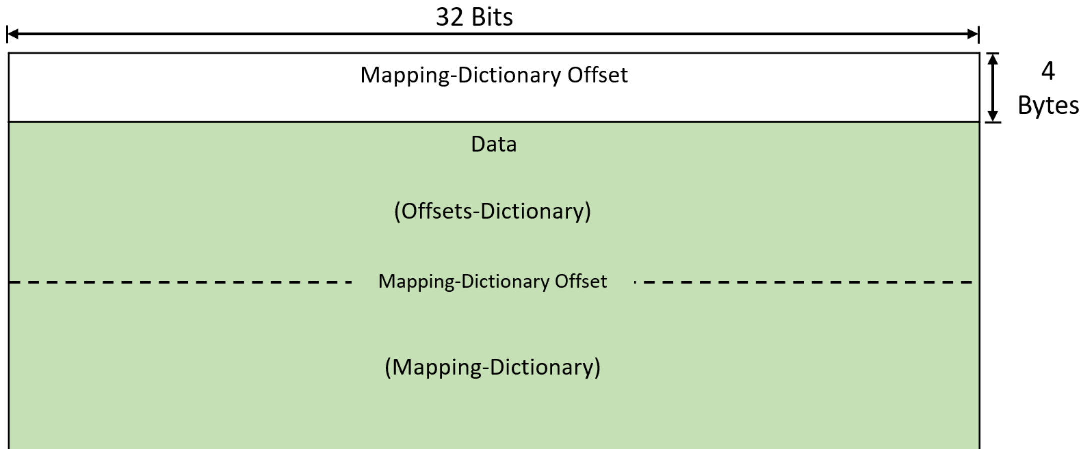

Figure 4.16: Format of a column compressed with 1-to-N Dictionary compression schemes, including a fixed size header and variable sized data chunk.

- Null-Reference Value: integer containing the reference value used for source values that are null.
- Data: contains the data chunk for the dictionary containing the reference value used for each source code.

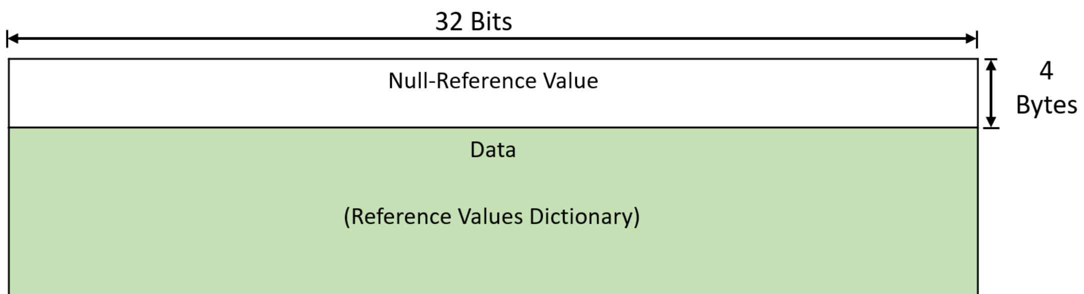

Figure 4.17: Format of a column compressed with DFOR compression schemes, including a fixed size header and variable sized data chunk.

The *Single-Column Compression Data* chunk contains the compressed target values generated by the scheme.

4.4.7 Compressed Dictionary-Sharing Scheme Format

A pair of columns compressed with the Dictionary-Sharing Scheme does not need any *Multi-Column Compression Data*, since the dictionary is stored with the dictionary-encoded source column, and the compressed codes of the target column are stored in the *Single-Column Compression Data* chunk.

4.4.8 Compressed Dictionary Format

Dictionaries of multi-column compression schemes and the *Multi-Column Compression Data* of columns which were dictionary encoded as the source column of a multi-column compression scheme are stored as shown in figure 4.18. The format contains:

- Data Type: identifier encoding the data type of the values in the dictionary.
- Data: contains the data chunk with the dictionary values.

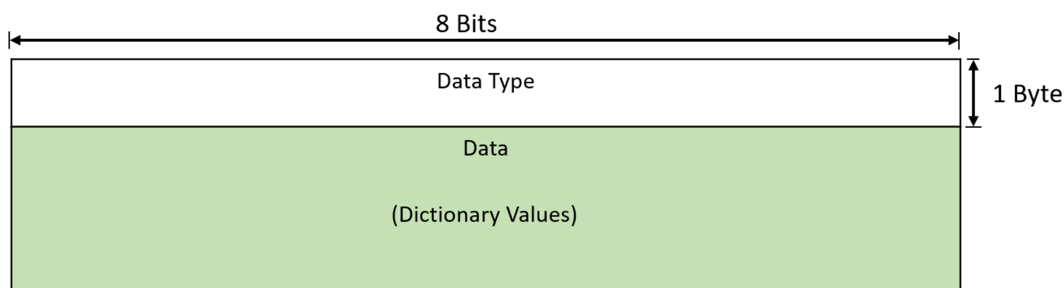

Figure 4.18: Format of a compressed dictionary, including a fixed size header and variable sized data chunk.

4.4.9 Compressed Exceptions Format

All exceptions of multi-column compression schemes are stored in the format as shown in figure 4.19. The format contains:

- Exception Values Offset: offset in bytes to access the exception values.
- Data: contains a data chunk containing the exception indexes and a data chunk containing the exception values.

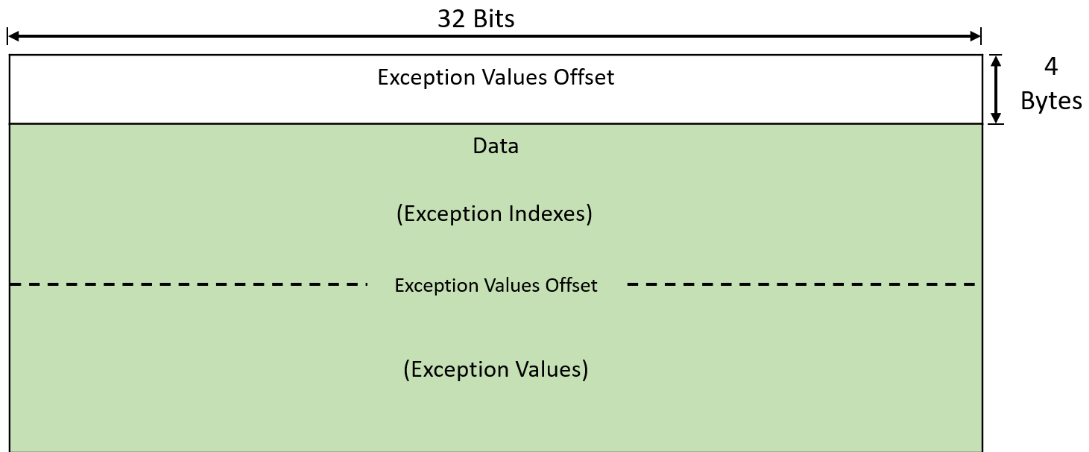

Figure 4.19: Format of compressed exceptions, including a fixed size header and variable sized data chunk.

4.4.10 Compressed Nullmap Format

Some multi-column compression schemes store nullmaps of the target columns separately, since they completely replace the target column chunk with dictionaries and exceptions. In these cases, the *Single-Column Compression Data* chunk is empty and the target nullmap is stored in the *Multi-Column Compression Data* chunk as shown in figure 4.20. The format contains:

- Nullmap Type: identifier encoding whether the nullmap is filled with only ones or zeros, or the bits were flipped for better Roaring compression.
- Data: contain the data chunk containing the Roaring bitmap.

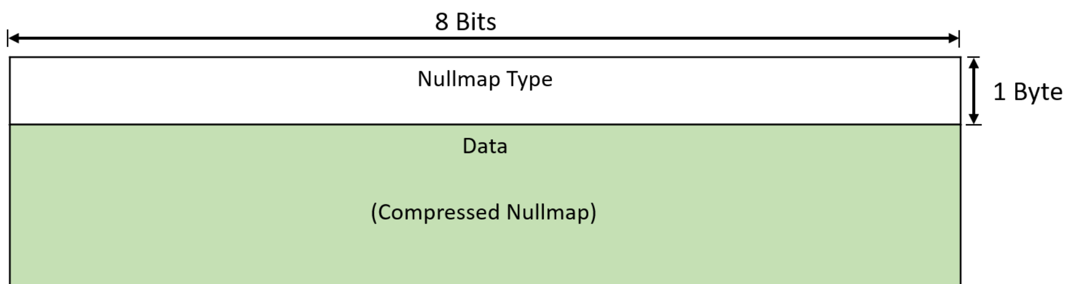

Figure 4.20: Format of a compressed nullmap, including a fixed size header and variable sized data chunk.

5 Results and Discussion

We evaluated the compression ratios achieved by our compression framework on the tables listed in table 2.2.

5.1 Baseline

As a baseline measurement, we disabled all multi-column compression schemes and only used the single-column compression schemes as described in section 4.1. The tests are run on a single row group.

Figure 5.1 shows the compression ratios achieved for each table, with a mean compression ratio of 6.72. These results represent the compression ratios achieved with regular single-column compression schemes.

5.2 Individual Multi-Column Schemes

Next, we evaluate how much compression benefit each individual multi-column scheme can add compared to the baseline. For testing each scheme, we disable all other multi-column schemes. We test each scheme with 8 different sample sizes, representing roughly 0.5%, 1%, 5%, 10%, 25%, 50%, 75%, 100% of total tuples. Since the multi-column schemes do not exploit patterns between consecutive tuples of columns, we do not need the samples to contain runs of tuples. We set the size of sample run size to 1 and use only the sample run count to adjust the sample size. The sample sizes affect the accuracy of the estimated compression ratios of the multi-column schemes, which are used to pick which schemes to compress with. We only use a single row group of each table for these tests.

Figure 5.2 shows how much each multi-column scheme improves the compression ratio of the baseline results on average when used individually. The mechanism to reverse multi-column schemes, if they compress worse than expected, is disabled for these tests. Table 5.1 shows the number of occurrences and average compression ratio improvement for each scheme, when using a sample size of 100%. The table suggests that correlations in the form of dictionary mappings between two columns are the most common type of correlation in the

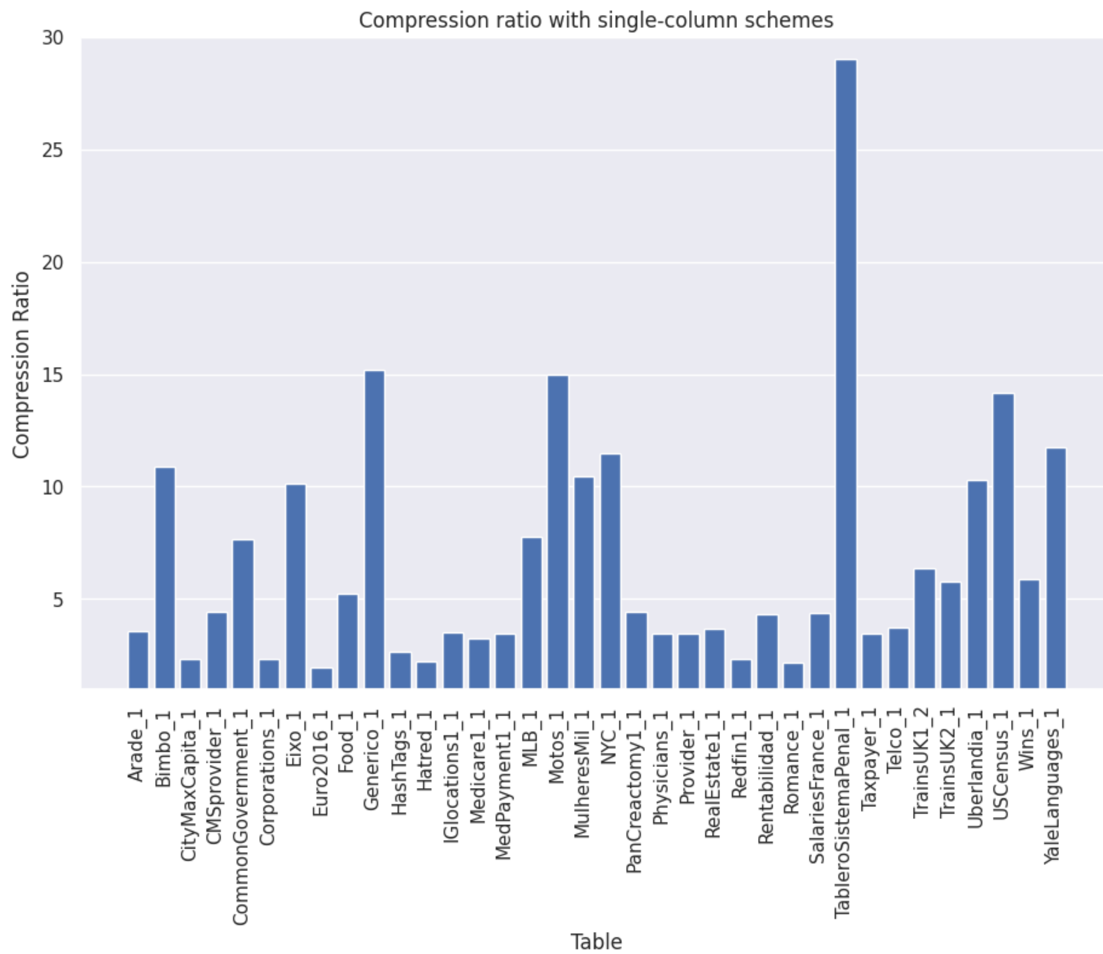

Figure 5.1: Compression ratios for each table using only single-column compression schemes.

dataset, whereas the Equality and Dictionary-Sharing schemes have the highest average compression improvement. Figure 5.3 shows the proportions of multi-column schemes used, and gives an estimate on how much each multi-column scheme contributes to the overall compression ratio improvement. 1-to-1 and 1-to-N Dictionary schemes are by far the most used, and the Numerical scheme is by far the least used.

Table 5.1: How often each multi-column scheme was used and the average compression ratio improvement when scheme is used individually. Using 100% sample size. Dataset contains 2223 columns in total.

	1-to-1 D.	1-to-N D.	Equality	D.-Sharing	DFOR	Numerical
Scheme count	489	363	111	163	137	30
Avg. Compr. Ratio Improv.	3.5x	2.8x	4.8x	4.2x	1.5x	1.4x

1-to-1 Dictionary Scheme. The compression ratio of the 1-to-1 Dictionary Scheme drops off sharply for sample sizes under 5% of total tuples. This can be explained by the difficulty of estimating the number of exceptions using very small sample sizes. Given a small sample, the probability is high that each unique sample of the source column is only encountered once, and consequently the sample contains no exceptions. This probability rises the more unique values the source column has, leading to false positives which seem to have few exceptions, but in reality, have many. These cases can lead to an even worse compression ratio than using single-column schemes, and drag down the average improvement. With sample sizes of 5% and higher the estimated compression ratios improve significantly and the 1-to-1 Dictionary Scheme improves the compression ratio of the baseline by roughly 17%. The average exception ratio is only 0.5%, which is much less than the upper limit we set of 10% (with 100% sample size).

1-to-N Dictionary Scheme. The 1-to-N Dictionary Scheme achieves around 20% improved compression ratios compared to the baseline for large sample sizes. The compression ratio drops off for sample sizes between 0.5% and 10%. The accuracy of the estimated compression ratios sinks since samples are needed to estimate the size of the mapping dictionary, which is dependent on the number of unique mappings between source and target values. Similar to the 1-to-1 Dictionary Scheme, the compression ratio drop-off for small sample sizes can be explained as false positives, in which the real compression ratio is worse than estimated.

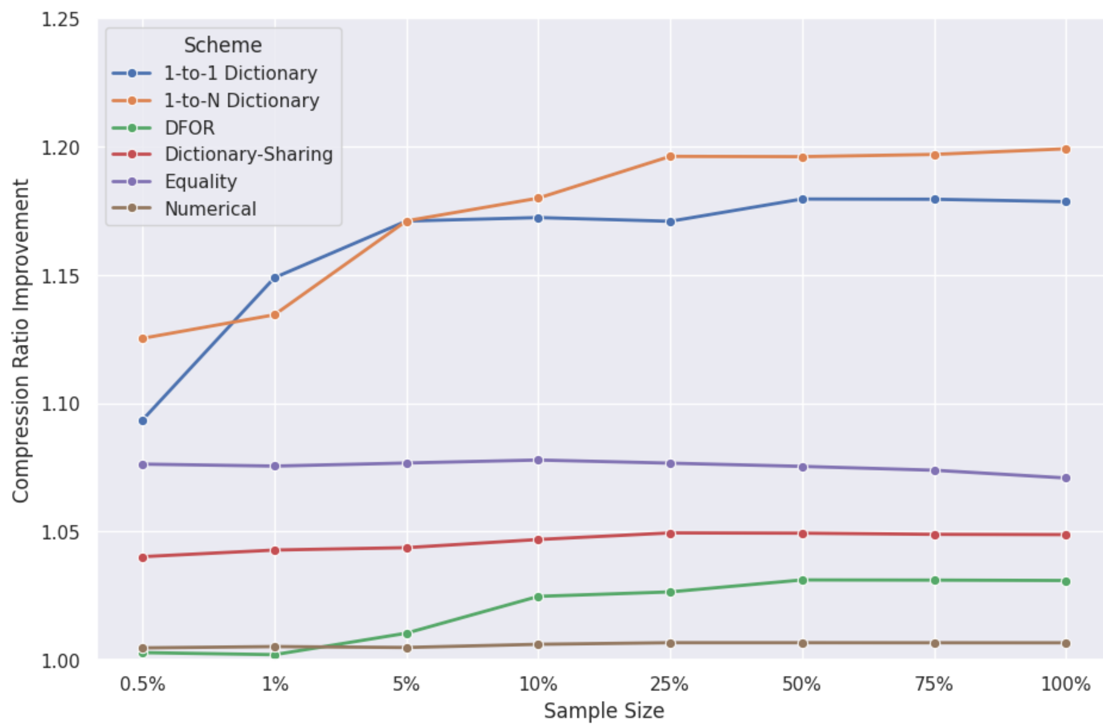

Figure 5.2: Compression ratio improvement of using individual multi-column compression schemes compared to the baseline result, using different sample sizes for computing estimated compression ratios of the multi-column schemes.

Equality Scheme. The Equality Scheme achieves the third-highest compression ratio improvement of roughly $7 - 8\%$. The compression ratio is relatively stable across all sample sizes, even though the estimated compression ratio is fully reliant on samples. This suggests that for column pairs in which a 0.5% sample is equal, there is a very high chance that most other values will be equal as well. The slight drop in compression ratio for the highest sample sizes is unexpected and may be due to the greedy algorithm of the correlation graph running into local minima. The average exception ratio of the Equality scheme is around 0.8%, much less than the upper limit we set of 10% (with 100% sample size).

Dictionary-Sharing Scheme. The Dictionary-Sharing Scheme achieves roughly 5% improved compression ratio. The compression ratio is relatively stable across all sample sizes, with little decrease for small sample sizes. This is because computing the estimated compression ratio does not need samples, but instead relies only on the unique values of the columns, which are provided by the column statistics. However, the sample size is still used as an upper limit on how many unique values to use for computing the estimated compression ratio. This explains why there is a slight compression ratio drop-off for low sample sizes, even though no direct column samples are used.

Dictionary-Frame-of-Reference Scheme. The Dictionary-Frame-of-Reference Scheme achieves roughly 3% improved compression ratio compared to the baseline for large sample sizes. The compression ratio improvement drops to almost 0 for small sample sizes, signifying the difficulty of accurately estimating the compression ratio with few samples. Low sample sizes lead to many false positives, in which correlations are falsely identified and have bad compression ratios.

Numerical. The Numerical Scheme achieves only around 0.5% improved compression ratios compared to the baseline. Table 5.1 shows that although the scheme improves compression ratios by an average of 40%, only 30 instances of Numerical correlation were found between integer columns in the dataset, which explains the overall little improvement.

5.3 Combining All Schemes

We test using all multi-column compression schemes combined. Figure 5.4 shows that the improved compression ratio lies between $23 - 26\%$, depending on the sample size. Surprisingly, the highest compression ratio is not achieved by the largest sample size, but by a sample size of 50%. Analyzing the output shows that this improvement only comes from two tables, and is not due to more

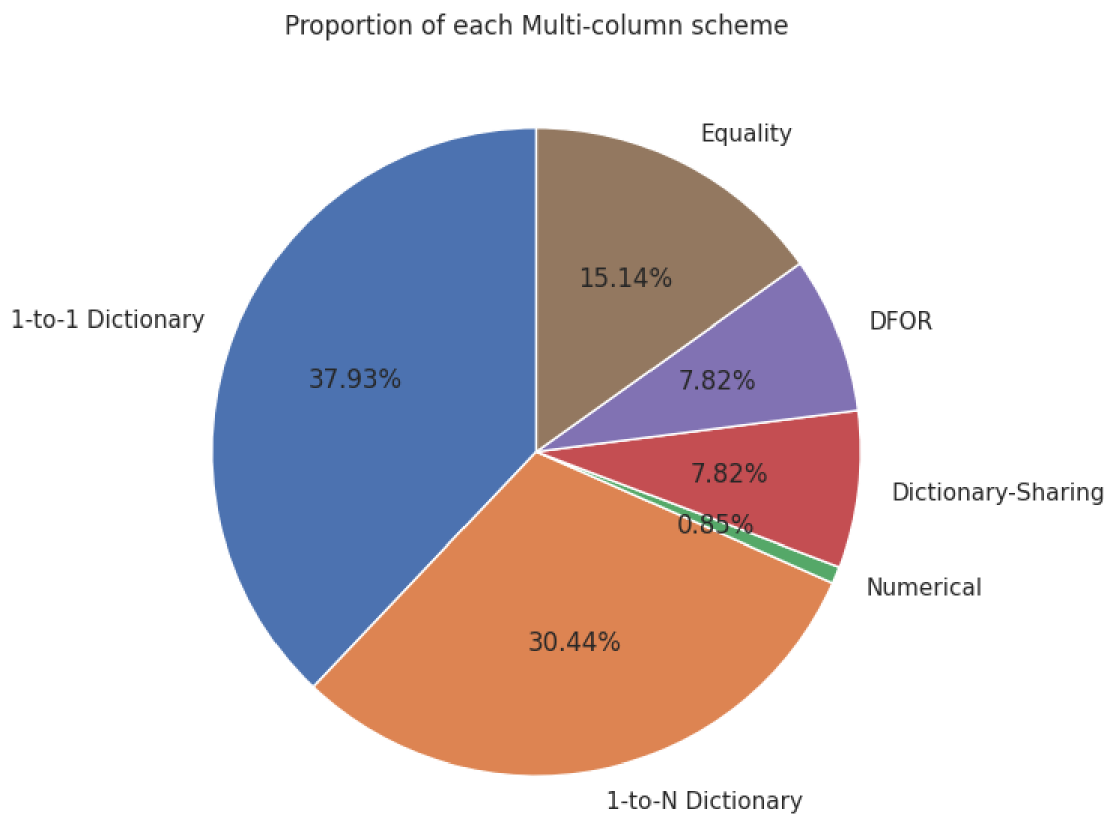

Figure 5.3: Proportion of how often each multi-column scheme is used, when all schemes are enabled and with sample size 65536.

accurate estimated compression ratios, but due to the greedy algorithm picking a more optimal combination of multi-column schemes from the correlation graph. This shows that there is potential to further increase the compression ratios by implementing a more sophisticated algorithm for picking schemes from the correlation graph.

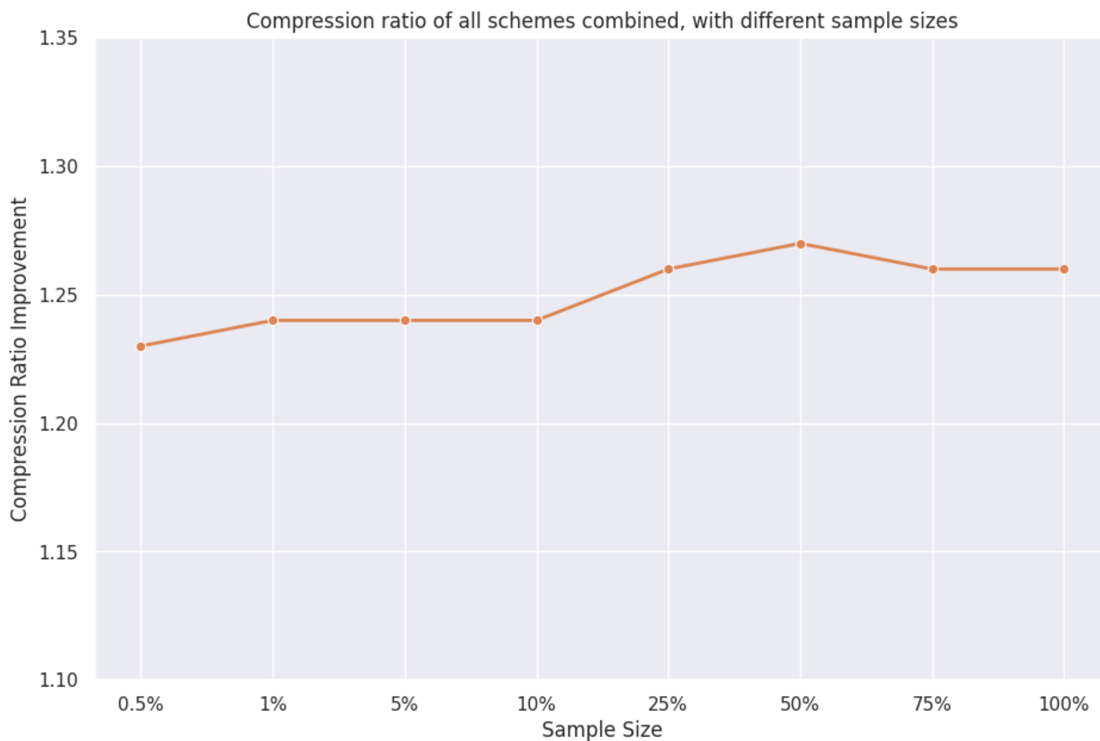

Figure 5.4: Compression ratio improvement when using all schemes combined.

5.3.1 Reversing Bad Multi-Column Schemes

We repeated the same test with the mechanism of reversing multi-column schemes turned off. Figure 5.5 shows the compression ratio improvement with and without reversing schemes. For sample sizes between $0.5 - 10\%$, the compression ratio improvement improves by up to 5% with scheme reversing enabled. The smaller the sample size, the more it benefits from being able to reverse multi-column schemes. From sample sizes of 25% and onwards, there is no difference in compression ratios, signifying that a sample size of 25% is large enough to avoid any significant amount of false positives when searching for correlations.

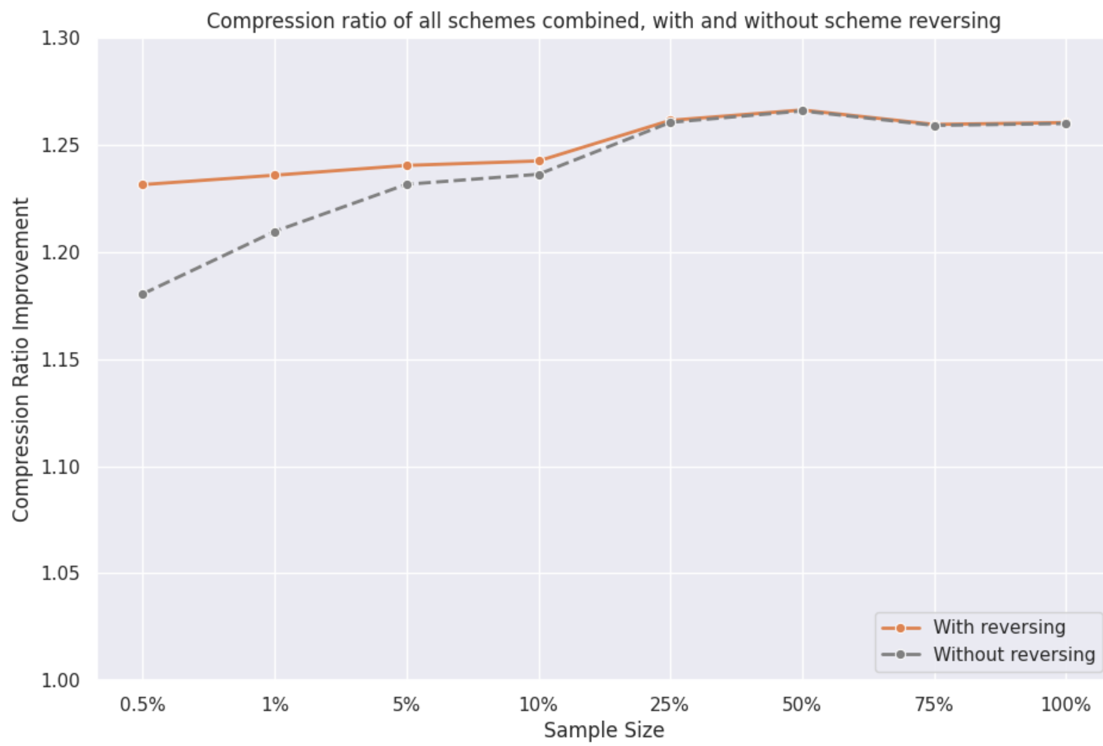

Figure 5.5: Compression ratio improvement with scheme reversing enabled and disabled. The bar graph shows the final number of schemes used, and the number of reversed schemes.

5.3.2 Sample run size

For all sample sizes, we only changed in the number of sample runs, but kept the sample run size equal to 1. Contrary to single-column schemes, multi-column schemes do not exploit patterns between consecutive tuples. Therefore, we do not need to sample runs of samples and individually sampled tuples should be enough. To confirm this, we run tests with a total sample size of around 1% (660 tuples), and use different configurations for the number of sample runs and the sample run size. Figure 5.6 shows the results, confirming that using sample runs does not improve the overall compression ratio. The highest compression ratio was achieved by sampling individual tuples.

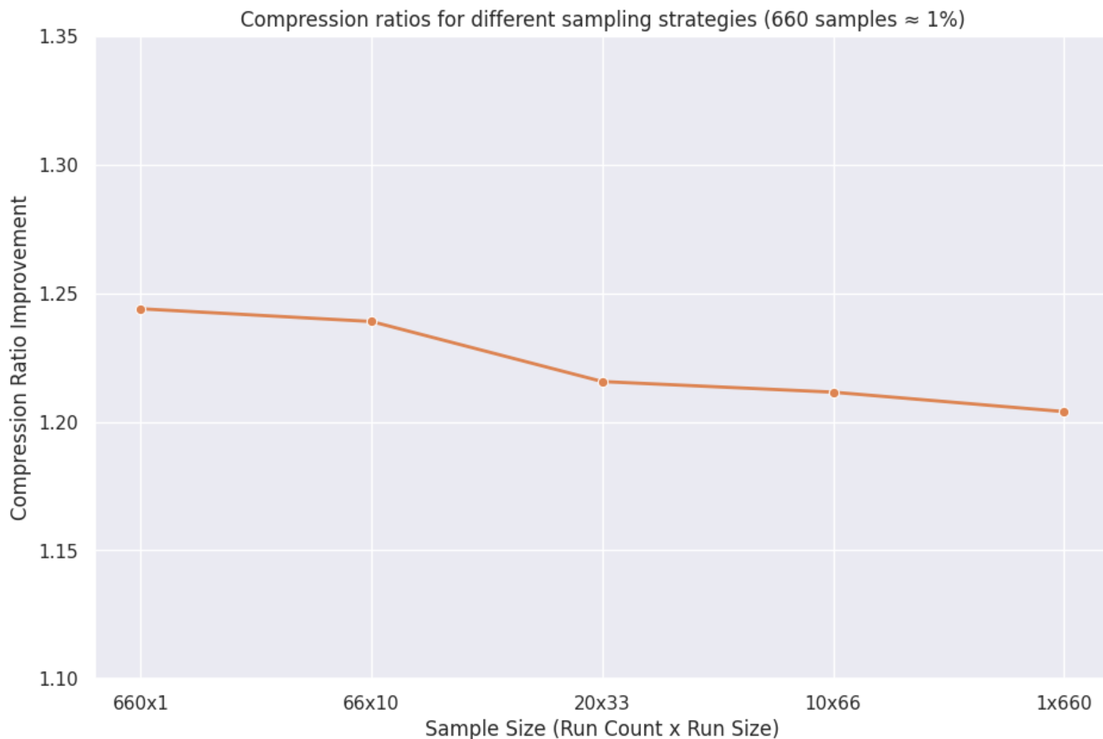

Figure 5.6: Compression ratio improvement with different sampling strategies.

5.3.3 Multi-Row Group: Sharing Correlations

To evaluate the effectiveness of sharing correlations between row groups, we run tests with up to 10 row groups for each table. The average number of row groups per table in this test is 9.1, since some tables have less than 10 row groups in total. For the Hatred table, we include only the first row group, since the second

row group contains contents that can not be parsed by the BtrBlocks CSV parser. Figure 5.7 shows the average compression ratio improvements with different options for sharing correlations between row groups. The compression ratio improvement for sharing the correlation graph and sharing only the final schemes is similar and is around 1% lower compared to not sharing any correlations and finding them from scratch for every new row group. The bar graph shows the total amount of expected compression ratios that are computed for each option, with scheme sharing needing to compute roughly only 10% the amount of expected compression ratios compared to no scheme sharing. The results show that reusing correlations found in one row group and sharing these with other row groups can drastically reduce the overhead of finding correlations, while sacrificing only very little compression ratio.

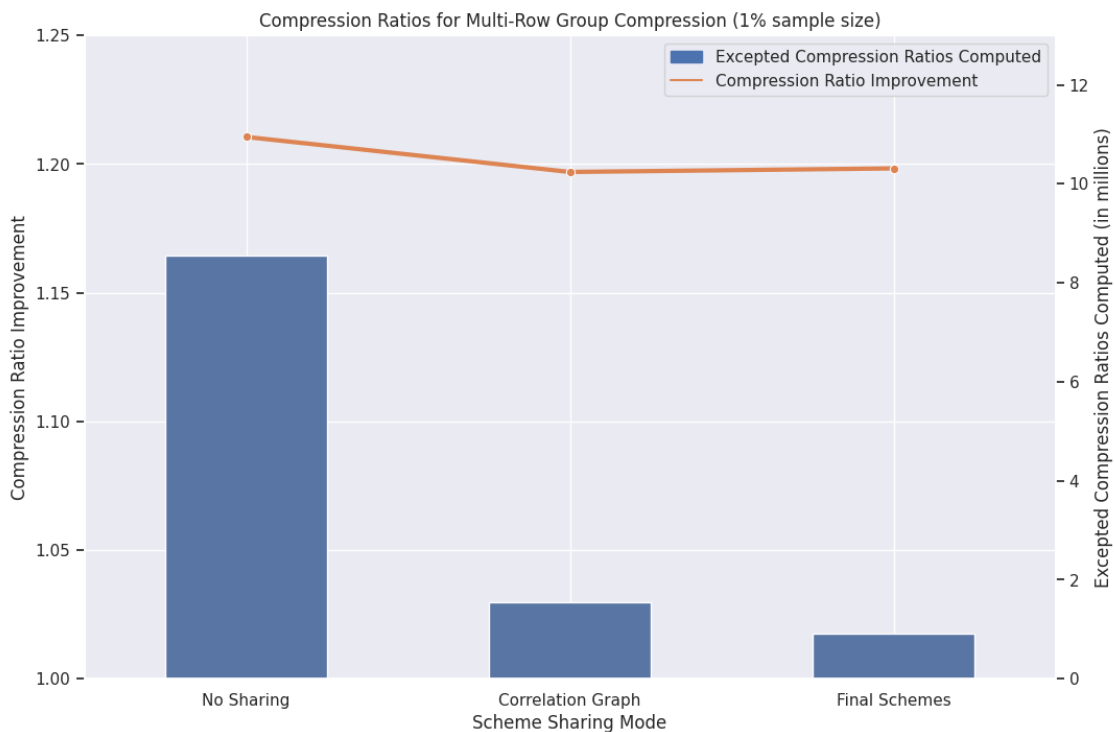

Figure 5.7

5.3.4 Compression Ratio Improvement per Table

Figure 5.8 shows the compression ratios of only using single-column compression schemes and the improved compression ratios of using both single- and multi-column schemes. The results show that the benefit of multi-column schemes

is highly dependent on the presence of column correlations. Many tables do not benefit at all from multi-column schemes, suggesting that those tables do not contain any correlations that are exploitable by our multi-column schemes. The compression ratio improvements shown in prior figures are the average improvement over all tables. The table with the most exploitable correlations is compressed 2.75x better by using our multi-column schemes. The compression ratio improvement of the 10 most improved tables is 1.6x.

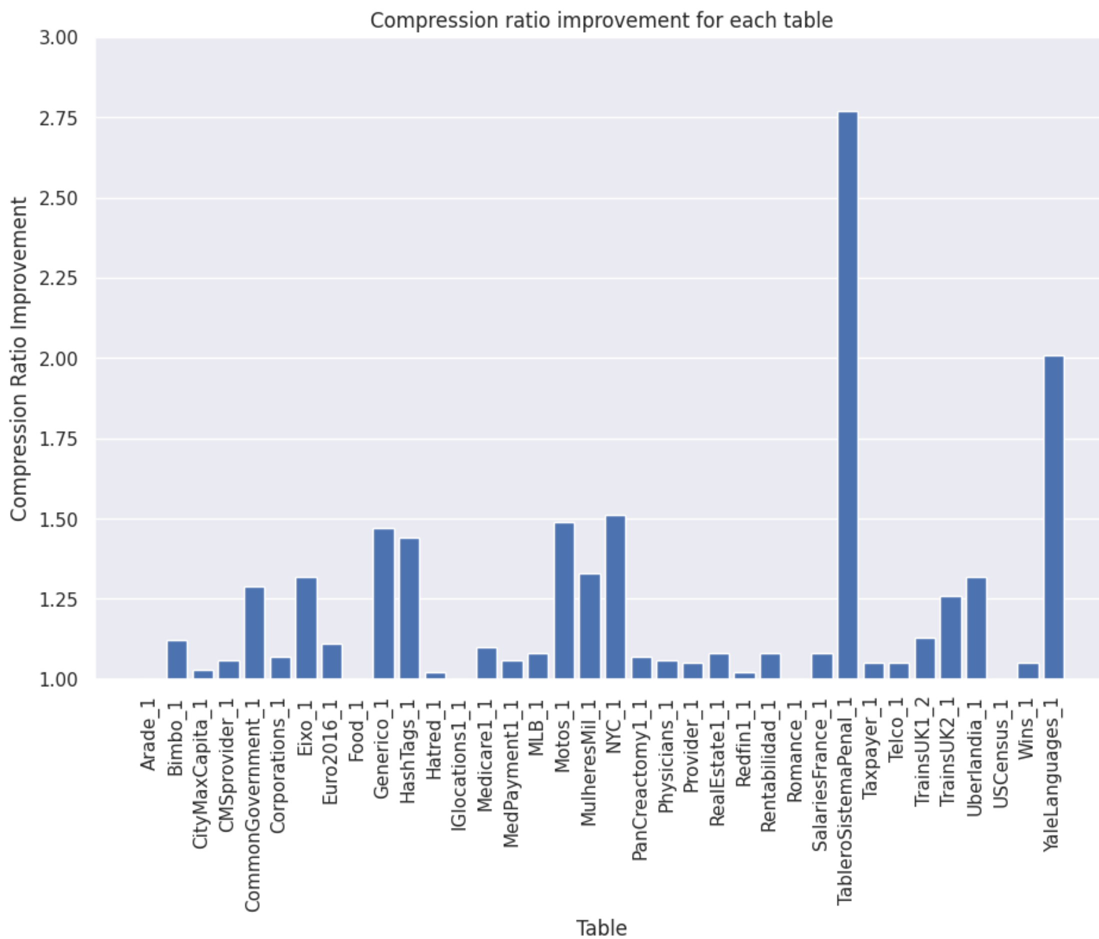

Figure 5.8: The compression ratio improvements by using multi-column schemes for each table in the dataset.

6 Conclusion

In this thesis, we explored the potential of exploiting column correlations for columnar LWC. We revisit our research questions defined in section 1.1 and discuss future work.

6.1 Research Questions

1. How can we exploit column correlations to improve compression? We proposed and implemented six multi-column compression schemes that improve compression by exploiting different kinds of correlations. Each scheme exploits correlations between a pair of source and target columns. The schemes we proposed are the Equality Scheme, 1-to-1 Dictionary Scheme, 1-to-N Dictionary Scheme, Numerical Scheme, Dictionary-Frame-of-Reference Scheme, and the Dictionary-Sharing Scheme.

2. How can column correlations be efficiently detected? Our results on the PublicBI benchmark show that a small sample size of only 0.5% is able to detect column correlations. When combined with a mechanism to reverse falsely detected correlations, the 0.5% sample size improved compression ratios by 1.23x, only slightly lower than the 26% gained by using the full row group as a sample.

3. How can multi-column LWC schemes that exploit column correlations be used in combination with typical single-column LWC schemes? We extended a typical LWC framework to be able to support our multi-column compression schemes. This included collecting column statistics, computing the expected compression ratios for multi-column schemes for all column pairs, greedily picking the most beneficial multi-column schemes from a correlation graph, and reversing the use of multi-column schemes if necessary.

4. How much compression benefit does exploiting column correlations provide in real-world datasets compared to existing LWC schemes? We tested our implementation on the Public BI Benchmark containing real-world datasets and achieved an average compression ratio improvement of around 20%. We saw that many tables in the benchmark contained little to no exploitable correlations. The 10 tables with the most correlations had improved compression ratios of

60% on average.

5. If compressing multiple row group groups, can we amortize the overhead of finding correlations over multiple row groups by reusing correlations found in one row group with other row groups? Using the PublicBI benchmark, we showed that by sharing the correlations found in the first row group with up to 10 following row groups, we could amortize the overhead of finding correlations and reduce the number of expected compression ratios computed by a factor of 10. The compression ratio improvement only decreased by 1%, confirming that the large majority of correlations hold over multiple row groups.

6.2 Future Work

During the design and implementation of our multi-column compression schemes and the compression framework, we used results on the Public BI benchmark to optimize multiple design parameters. By using the Public BI benchmark both as a training and testing dataset, our parameters may be overfitted specifically to the Public BI benchmark. It would therefore be valuable future work to confirm the results by testing the benefits of our multi-column compression schemes on other benchmarks with real-world data. Additional future work for exploiting correlations for compression includes benchmarking the compression and decompression speeds for the multi-column schemes. The authors of the Whitebox Compression paper claim that most correlations they found were only present after they applied transformations such as splitting prefixes of strings into separate columns [15]. We plan to evaluate how much compression ratios improve when combining these transformations with our multi-column schemes. Lastly, we believe that there are many more types of exploitable correlations to be found in real-world datasets, that are not covered by the six schemes we proposed here.

List of Figures

2.1 An integer column is compressed using one-value encoding. . . 6
2.2 A string column is compressed using Frequency Encoding. . . . 7
2.3 A string column is compressed using Run-Length Encoding. . . 7
2.4 An integer column is compressed using bit-packing. 8
2.5 An integer column is compressed using Frame of Reference encoding. 9
2.6 A float column is compressed using Pseudo-decimal encoding. . 10
2.7 A string column is compressed using Dictionary encoding. . . . 11
2.8 A string column is compressed using FSST encoding. "UM" is replaced with the symbol "0", "BRA" is replaced with the symbol "1". 11

4.1 Overview of the compression workflow. Columns of a row group are analyzed to collect statistics, which are used in conjunction with samples to compute estimated compression ratios (ECR) for all compression schemes. The best schemes are selected and applied to compress the columns. 23
4.2 An example of an encoded string column containing 6 strings. Given n strings, n+1 integer offsets are used to find the start and end of each string. All string characters are stored in a single chunk after the offsets. 25
4.3 A pair of columns encoded with the Equality compression scheme. 27
4.4 A pair of columns encoded with the Numerical compression scheme. 29
4.5 A pair of columns encoded with the 1-to-1 Dictionary compression scheme. 31
4.6 A pair of columns encoded with the 1-to-N Dictionary compression scheme. 33
4.7 A pair of columns encoded with the DFOR compression scheme. 35
4.8 A pair of columns encoded with the Dictionary-Sharing compression scheme. 37

4.9 An example of a correlation graph. Nodes represent columns and weights on the edges represent the estimated amount bytes saved by using the multi-column scheme over single-column schemes. For example, the 1-to-N Dictionary scheme can be applied between the source column 7 and target column 10, and would save an estimated 32994 bytes, compared to using the best single-column schemes on both columns. 42

4.10 The final edges chosen by the greedy algorithm, applied to the correlation graph shown in Figure 4.9. The algorithm picks edges greedily in an attempt to maximize the number of bytes saved by using multi-column schemes. The order of edges picked is $(11 \rightarrow 10)$, $(11 \rightarrow 3)$, $(7 \rightarrow 2)$, $(7 \rightarrow 5)$, $(7 \rightarrow 4)$. 43

4.11 Format of a compressed column, including a fixed size header and variable sized data chunk. 49

4.12 Format of a value chunk compressed with single-column compression schemes, including a fixed size header and variable sized data chunk. 50

4.13 Format of a column compressed with Equality compression schemes, including a fixed size header and variable sized data chunk. . . . 51

4.14 Format of a column compressed with Numerical compression schemes. 52

4.15 Format of a column compressed with 1-to-1 Dictionary compression schemes, including a fixed size header and variable sized data chunk. 53

4.16 Format of a column compressed with 1-to-N Dictionary compression schemes, including a fixed size header and variable sized data chunk. 54

4.17 Format of a column compressed with DFOR compression schemes, including a fixed size header and variable sized data chunk. . . . 54

4.18 Format of a compressed dictionary, including a fixed size header and variable sized data chunk. 55

4.19 Format of compressed exceptions, including a fixed size header and variable sized data chunk. 56

4.20 Format of a compressed nullmap, including a fixed size header and variable sized data chunk. 56

5.1 Compression ratios for each table using only single-column compression schemes. 58

5.2 Compression ratio improvement of using individual multi-column compression schemes compared to the baseline result, using different sample sizes for computing estimated compression ratios of the multi-column schemes. 60
5.3 Proportion of how often each multi-column scheme is used, when all schemes are enabled and with sample size 65536. 62
5.4 Compression ratio improvement when using all schemes combined. 63
5.5 Compression ratio improvement with scheme reversing enabled and disabled. The bar graph shows the final number of schemes used, and the number of reversed schemes. 64
5.6 Compression ratio improvement with different sampling strategies. 65
5.7 66
5.8 The compression ratio improvements by using multi-column schemes for each table in the dataset. 67

List of Tables

2.1 Compression schemes and data types supported by BtrBlocks. Many schemes support cascading compression, where the output can be recursively compressed further. 13
2.2 Tables of the Public BI benchmark subset, with statistics on column count and data type. 15

4.1 The single-column compression schemes of BtrBlocks, modified to represent regular LWC schemes with limited cascading compression. 24
4.2 Data type requirements for multi-column compression schemes. 39
4.3 Column-pair pruning rules for multi-column compression schemes using column statistics. 40

5.1 How often each multi-column scheme was used and the average compression ratio improvement when scheme is used individually. Using 100% sample size. Dataset contains 2223 columns in total. 59

Bibliography

[1] *Apache Parquet*. accessed: 27.11.2023. URL: https://parquet.apache.org/.

[2] *Apache ORC*. accessed: 27.11.2023. URL: https://orc.apache.org/.

[3] M. Kuschewski, D. Sauerwein, A. Alhomssi, and V. Leis. "BtrBlocks: Efficient Columnar Compression for Data Lakes". In: *Proceedings of the ACM on Management of Data* 1.2 (2023), pp. 1–26.

[4] D. Abadi, S. Madden, and M. Ferreira. "Integrating compression and execution in column-oriented database systems". In: *Proceedings of the 2006 ACM SIGMOD international conference on Management of data*. 2006, pp. 671–682.

[5] M. Zukowski, S. Heman, N. Nes, and P. Boncz. "Super-scalar RAM-CPU cache compression". In: *22nd International Conference on Data Engineering (ICDE'06)*. IEEE. 2006, pp. 59–59.

[6] Facebook. *Facebook/ZSTD: Zstandard - fast real-time compression algorithm*. accessed: 27.11.2023. URL: https://github.com/facebook/zstd.

[7] D. Abadi, P. Boncz, S. Harizopoulos, S. Idreos, S. Madden, et al. "The design and implementation of modern column-oriented database systems". In: *Foundations and Trends® in Databases* 5.3 (2013), pp. 197–280.

[8] J. Goldstein, R. Ramakrishnan, and U. Shaft. "Compressing relations and indexes". In: *Proceedings 14th International Conference on Data Engineering*. IEEE. 1998, pp. 370–379.

[9] P. Boncz, T. Neumann, and V. Leis. "FSST: Fast Random Access String Compression". In: *Proc. VLDB Endow.* 13.12 (July 2020), pp. 2649–2661. ISSN: 2150-8097. DOI: 10.14778/3407790.3407851. URL: https://doi.org/10.14778/3407790.3407851.

[10] CWIDA. $cwida/public_b i_b enchmark : BIbenchmarkwithusergenerateddataandqueries$. accessed: 27.11.2023. URL: https://github.com/cwida/public_bi_benchmark.

[11] P. G. Brown and P. J. Haas. "BHUNT: Automatic discovery of fuzzy algebraic constraints in relational data". In: *Proceedings 2003 VLDB Conference*. Elsevier. 2003, pp. 668–679.

[12] I. F. Ilyas, V. Markl, P. Haas, P. Brown, and A. Aboulnaga. "CORDS: Automatic discovery of correlations and soft functional dependencies". In: *Proceedings of the 2004 ACM SIGMOD international conference on Management of data*. 2004, pp. 647–658.

[13] H. V. Nguyen, E. Müller, P. Andritsos, and K. Böhm. "Detecting correlated columns in relational databases with mixed data types". In: *Proceedings of the 26th International Conference on Scientific and Statistical Database Management*. 2014, pp. 1–12.

[14] B. Ghita, D. G. Tomé, and P. A. Boncz. "White-box Compression: Learning and Exploiting Compact Table Representations." In: *CIDR*. Vol. 1. 2020, p. 27.

[15] B. Ghita. *Self-learning Whitebox Compression*. 2019. URL: https://homepages.cwi.nl/~boncz/msc/2019-BogdanGhita.pdf.

[16] X. Lyu, A. Kipf, P. Pfeil, D. Horn, J. Giceva, and T. Kraska. "CorBit: Leveraging Correlations for Compressing Bitmap Indexes". In: (2023).

[17] V. Raman, G. Attaluri, R. Barber, N. Chainani, D. Kalmuk, V. KulandaiSamy, J. Leenstra, S. Lightstone, S. Liu, G. M. Lohman, et al. "DB2 with BLU acceleration: So much more than just a column store". In: *Proceedings of the VLDB Endowment* 6.11 (2013), pp. 1080–1091.