Universitatea Politehnica din București

Vrije Universiteit Amsterdam

Facultatea de Automatică și Calculatoare

Centrum Wiskunde & Informatica

Master Thesis

Parallel and Distributed Computer Systems

Self-learning Whitebox Compression

Author: Bogdan Ghiță

1st supervisor:	Peter Boncz
daily supervisor:	Peter Boncz
2nd supervisor:	Diego Tomé
2nd reader:	Hannes Mühleisen

August 26, 2019

Abstract

Existing columnar database systems rely on a small set of lightweight compression techniques to achieve smaller data size and fast query execution: RLE, DICT, FOR, DELTA and their improved versions. These are hardcoded black-box approaches with limited capacity of exploiting all compression opportunities present in real data. We propose *whitebox compression*: a new compression model which represents data through elementary operator expressions automatically generated at bulk load. This is done by learning patterns from the data and associating them with operators to form expression trees, which are stored together with the compressed data and lazily evaluated during query execution. *Whitebox compression* automatically finds and exploits compression opportunities, leading to transparent, recursive and more compact representations of the data. Combined with vectorized execution or JIT code generation, it has the potential to generate powerful compression schemes in terms of both compression ratio and query execution time. Our focus is on real data rather than synthetic datasets, thus we develop and evaluate the *whitebox compression* model using the Public BI benchmark—a comprehensive human generated benchmark for database systems.

Acknowledgements

I would like to thank my supervisors Peter Boncz and Diego Tomé for their guidance and support in the past seven months. Special appreciation goes to Peter for his constructive feedback and active involvement in my work and for his vast knowledge and strive for perfection which greatly contributed to the improvement of this thesis. I further thank Stefan Manegold for his help with the creation of the Public BI benchmark and Tim Gubner for providing me with insights into VectorWise. I thank my colleagues and friends Per Fuchs and Matheus Nerone for the interesting brainstorming sessions and Richard Gankema for his advice on various aspects related to my thesis. Finally, I wish to thank Hannes Mühleisen for his feedback on my work.

Contents

List of Figures

List of Tables

LIST OF TABLES

1

Introduction

This thesis presents a new compression model for columnar database systems: *whitebox compression*. Existing DBMSs use compression to make data smaller in terms of disk space, respectively to make queries faster by reducing I/O and memory usage and operating on the compressed data directly, without decompression. State-of-the-art columnar compression methods may not exploit all opportunities offered by the data, e.g. because these methods (RLE, FOR, DICT, DELTA) work on only one column at a time and hence cannot exploit correlations between multiple columns. Another reason is that users often use "sub-optimal" datatypes to represent their data, e.g. storing dates or numbers in strings, which are harder to compress and more expensive to operate on. Achieving high compression ratios is an important factor that influences the ability of DBMSs to scale up in terms of storage space. Compressed data leads to reducing the I/O bottleneck between disk and main memory and even between main memory and CPU if done at a granular level (1). Moreover, higher computation efficiency can be achieved through compression, since operating over thinner data optimizes the use of SIMD instructions.

Actual data found in real datasets tends to exhibit phenomena not found in synthetic database benchmarks like TPC-H (2) and TPC-DS (3). Not only is data often skewed in terms of value and frequency distribution, but it is also correlated across columns. Our recent efforts led to the first fully user-generated benchmark for database systems: the Public BI benchmark (4). Its many human-generated datasets and real data distributions open new grounds for research in the direction of columnar data compression and compressed execution. Our goal is to leverage the characteristics of these datasets for finding new methods of compression which perform well on real data rather than synthetic benchmarks. We believe the best way to achieve this is through *whitebox compression*: using basic operators to create expression trees that enable efficient processing of the data.

1.1 Whitebox compression

Whitebox compression is a compression model for database systems that represents data through an expression language composed of elementary operators. Multiple operators are chained together to form an expression tree where inner nodes are operators and leaves are physical columns. A physical column is the partial representation of a logical column used for storage on disk. A logical column is the data as seen by the user. The evaluation of an expression tree on the physical columns generates the original logical columns.

Take for example, a logical column `A` which stores email addresses. It can be split by the `'@'` character into 2 new columns `A1` and `A2`. The first one will contain the local/username part of the email address and the second one its domain (e.g. for `"john.smith@gmail.com"` `A1` will store `"john.smith"` and `A2` will store `"@gmail.com"`). The second column will then be compressed with dictionary encoding to exploit the small number of unique email address domains. Stored together, email addresses are not compressible because of the uniqueness of the usernames. Through *whitebox* representation, subparts of the data can be compressed independently.

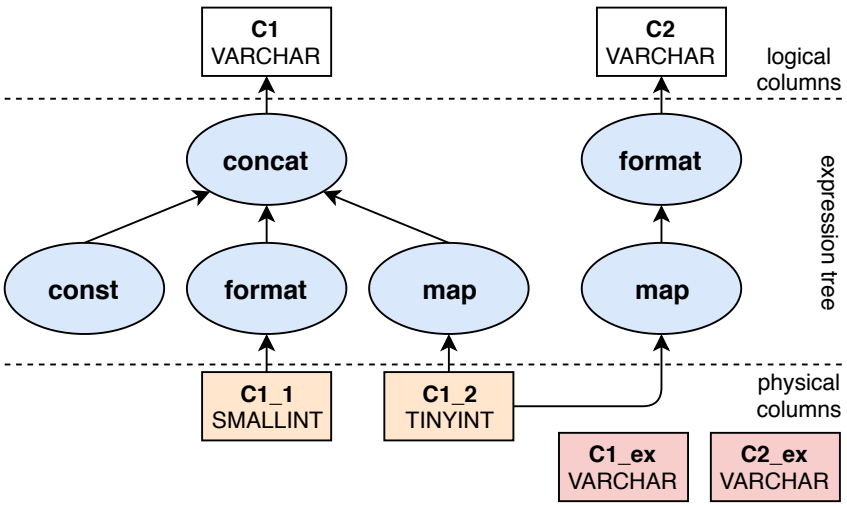


Figure 1.1: Whitebox representation example

Figure 1.1 illustrates the general concept of *whitebox compression*. `C1` and `C2` are 2 logical columns represented through elementary operator expressions as functions of the physical columns `C1_1` and `C1_2`. The operators are, in this example: `concat`, `format`, `map` and `const`. Chained together, they form *expression trees* that transparently describe the columnar transformations applied to the data. Values that do not match the expressions are stored in the exception columns `C1_ex` and `C2_ex`.

The *whitebox* representation creates compression opportunities. While in the original format, columns cannot be compressed through existing lightweight techniques, the physical columns resulting after applying the operator expressions are storing data more compactly. E.g. string columns are decomposed into subcolumns based on the different distributions of substring components and redundancy is eliminated by representing correlated columns as functions of the same physical columns. This process leads to efficient representation of the data in columns with appropriate datatypes and sparse exception columns.

The difference between *blackbox compression* and *whitebox compression* is best perceived from the system's perspective. Table 1.1 shows a comparison between the 2 models.

	Blackbox	Whitebox
Nature	hardcoded, opaque	generic, flexible, transparent
Header info	identifier (e.g. PFOR)	operator expression, self-descriptive
Output	block of data	columns (allows recursion)
Exceptions	included in the data block	separate columns

Table 1.1: Blackbox vs. whitebox comparison

A *blackbox compression* system takes as input a column and outputs a block of data, containing both the compressed values and the exceptions, stored in a format only known by the system. In contrast, a *whitebox compression* system takes as input a set of columns and outputs an expression tree and other columns. The representation of the columns is transparent and self-descriptive, which enables partial decompression of the data by pushing operators down the compression tree at query execution time. The *whitebox compression* model is generic and flexible, allowing recursive compression of columns—including exceptions. It can be easily extended with new operators and adapted according to the system's needs and data characteristics.

1.2 Research questions

The goal of our project is to explore the concept of *whitebox compression* with the purpose of finding a better way to compress human-generated data. To this end, we defined a list of scientific questions that we answer during our research.

1) What does real, user-generated data look like—specifically in the case of the Public BI benchmark?
    a) What patterns can we find in the columns of each dataset?
    b) Inefficient ways of representing data?

   c) "Wrong" type used to define data (e.g. number stored as string, etc.)?

2) How good are existing compression schemes at compressing real data?
   a) Do they make the most out of the properties of the data?
   b) Is there room for improvement?

3) Can we represent the logical columns more compactly through an expression tree composed of elementary operators?
   a) What kind of operators are suitable for expressing the data and transforming it to physical columns?
   b) What will the compression ratio be?
   c) Can we exploit correlations between multiple logical columns by sharing the physical columns in their expressions?

4) Can we create an automatic learning process that will generate suitable compression trees for each column?
   a) Will it provide a high compression ratio?
   b) How will it compare to existing solutions?

The focus of this thesis is on learning patterns from the data and automatically generating expression trees for more compact data representation—everything optimized for size. An additional, but equally important aspect of this topic is query execution time and whether it can be improved through *whitebox compression*. We leave this subject for future work. We defined the corresponding research question below for completeness, but we do not answer it in this thesis.

5) Can we achieve compressed execution with the *whitebox compression* model?
   a) Can we exploit SIMD in the scan?
   b) Will there be lazy decompression opportunities?
   c) Can we push query predicates down the compression tree?

We carried out our research and structured this thesis according to these questions. We present related work in Chapter 2. We performed a characterisation of the Public BI benchmark and then manually analyzed its datasets in search for patterns and compression opportunities not exploited by the existing systems (Chapter 3). Based on our findings, we defined the *whitebox compression* model, the expression language and all its characteristics in Chapter 4. Furthermore, we defined algorithms that automatically detect compression

opportunities and generate compression trees that represent data more compactly (Chapter 5). Finally, we evaluate our proof-of-concept implementation of *whitebox compression* against an existing database system and an estimator model baseline (Chapter 6).

2

Related work

The topic of data compression has received significant attention from the database research community over the past 25 years (1, 5, 6, 7, 8). Existing work covers a wide range of approaches to this problem: compression algorithms, efficient hardware-conscious implementations, compressed execution, all integrated into real database systems (9, 10).

The goal of reducing I/O bandwidth has directed the focus towards lightweight compression schemes: dictionary compression, run-length encoding, frame-of-reference, delta coding, null suppression (1, 5, 11, 12, 13). Zukowski et al. (1) proposed improved versions of these techniques that efficiently handle exceptions—making the compression methods less vulnerable to outliers—and achieve fast vectorized execution. Various storage formats that facilitate compression and compressed execution have been proposed. Data Blocks (6) is a compressed columnar storage format that reduces the memory footprint through hot-cold data classification. Raman et al. (14) optimized query execution time for analytics use cases through in memory query processing on the dictionary compressed column-organized format of IBM DB2 BLU.

All this work focuses on low level optimizations to either speed up query execution or improve the compression rates. These solutions revolve around the same compression schemes and rely on fine-tuned hardware-conscious implementations leveraging SIMD instructions and vectorized execution to save CPU cycles and increase CPU efficiency. In contrast, we approach the problem of compression from a different angle. Our focus is on expressing the data in a completely different way, through simple operators chained together to form data-aware compression trees. Automatic generation of these compression expressions is based on finding patterns in the data and correlations amongst different columns. Lee et al. (15) mention the possibility of exploiting the correlations among columns at query time with the purpose of performing join operations on columns with different encodings.

In contrast, we want to exploit these correlations during compression, to obtain better compression ratios.

The closest work to our research is (16), where Raman et al. exploit data properties (skewed distributions and correlations) to achieve high compression ratios. They concatenate correlated columns and encode them together using a variation of Huffman trees which preserve partial ordering. Additionally, a sequence of type specific transformations, sorting and delta coding is also applied. We see this approach as heavy-weight black-box compression as it is hard-coded and relies on multiple rounds of Huffman encoding. Moreover, all the patterns in the data need to be manually supplied by the user (correlations and domain specific transformations). Our work differs in multiple ways: 1) we exploit correlated columns by sharing the same physical columns between related logical columns; 2) we apply a wide range of domain specific operators tailored to the data, including splitting columns and then recursively applying the same procedure; 3) our process is fully automated, from determining patterns and correlations between columns, to generating custom expressions trees.

Damme et al. (17) performed an exhaustive evaluation of existing compression algorithms and concluded that compression rates are highly dependent on the properties of the data and that combinations of multiple techniques lead to the best results. These results support our reasons for better understanding the data and chaining elementary operators into complex compression trees and encourage us to see how our *whitebox* approach affects both compression rates and query execution time.

Additionally, while most compression solutions proposed so far were mainly evaluated and compared to each other on synthetic benchmarks (1, 5, 6, 14, 15), we are the first to use such a comprehensive human-generated benchmark as the Public BI benchmark (4)—which we defined as an early stage of this research project and will be described in more depth in the next chapter. Two examples of benchmarks used for evaluating database compression and compressed execution are TPC-H (2) and its successor TPC-DS (3). Both are synthetic, using uniform or stepwise uniform column value distributions, with fully independent columns in and between tables. This absence of skew, data dirtiness and correlation does not reflect the nature of real data. In contrast, the Public BI benchmark contains 386 GB of real data and 646 analytics queries available on Tableau Public (18, 19). The large volume of data, it's diversity in content and the extended character set make it suitable for evaluating compression solutions. It is an open-source benchmark and we hope it will be useful to the database community in many ways.

2.1 Contributions

This thesis brings the following contributions:

1) Public BI benchmark analysis—an analysis of real, user-generated data from the perspective of compression
2) *whitebox compression*—a new generic, extensible, recursive and transparent compression model for columnar databases, which achieves higher compression ratios by representing data more compactly through elementary operator expressions and creates opportunities for faster query execution
3) *learned compression*—automatic identification of patterns in the data and generation of compression trees: multiple pattern detectors, a cost model and compression learning algorithms

3

Public BI benchmark

The Public BI benchmark (4) is a user-generated benchmark for database systems derived from the DBTest'18 paper by Tableau (18). It contains 386GB of real data and 646 analytics queries, available at `https://github.com/cwida/public_bi_benchmark` under the MIT License. The data distributions, diversity in content and the extended character set make it suitable for evaluating compression solutions.

The benchmark was created by downloading 46 of the biggest workbooks from Tableau Public (19) and converting the data to CSV files. The SQL queries were collected from the Tableau logs that appear when visualizing the workbooks (SQL queries to the integrated HyPer (9) engine). We processed the CSV files with the purpose of making them load into different database systems. The queries contained Tableau-specific functions and syntax. We processed and adapted them in order to run on MonetDB (20) and VectorWise (10). Table 3.1 shows a summary of the benchmark.

This chapter presents a characterisation of the Public BI benchmark that we performed with the purpose of understanding what real data looks like and finding opportunities for more compact data representation. This analysis was performed solely from the perspective of compression. We did not analyze entities, relationships or the queries.

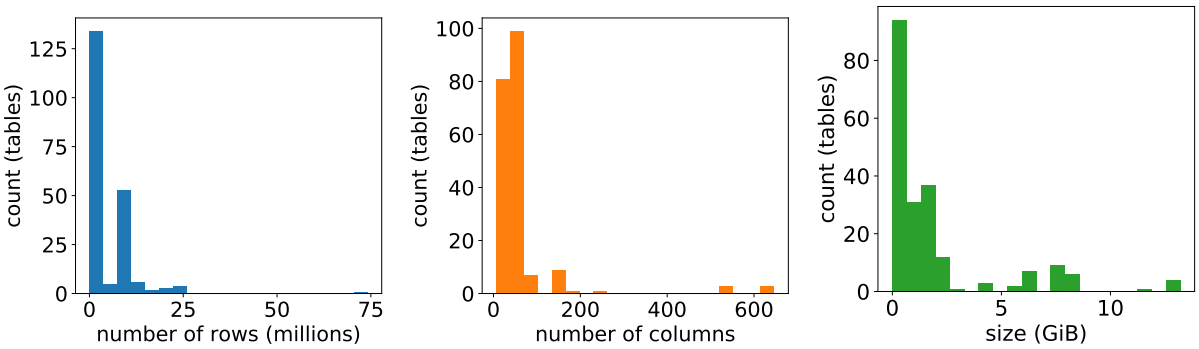
Workbook	Tables	Columns	Rows	Queries	CSV size
Arade	1	11	9.9M	1	811.4MiB
Bimbo	1	12	74.2M	2	3.0GiB
CMSprovider	2	52	18.6M	3	3.9GiB
CityMaxCapita	1	31	912.7K	10	333.0MiB
CommonGovernment	13	728	141.1M	38	102.5GiB
Corporations	1	27	741.7K	1	202.2MiB
Eixo	1	80	7.6M	24	6.4GiB
Euro2016	1	11	2.1M	1	390.6MiB
Food	1	6	5.2M	1	205.9MiB
Generico	5	215	114.1M	38	64.5GiB
HashTags	1	101	511.5K	12	640.2MiB
Hatred	1	31	873.2K	26	309.4MiB
IGlocations1	1	18	81.6K	3	6.6MiB
IGlocations2	2	40	4.3M	13	1.8GiB
IUBLibrary	1	27	1.8K	3	443.3KiB
MLB	68	3733	32.5M	95	8.2GiB
MedPayment1	1	28	9.2M	1	1.7GiB
MedPayment2	1	29	9.2M	1	1.8GiB
Medicare1	2	52	17.3M	10	3.3GiB
Medicare2	2	56	18.3M	9	3.4GiB
Medicare3	1	29	9.3M	1	2.1GiB
Motos	2	88	28.4M	24	16.1GiB
MulheresMil	1	81	7.6M	35	6.4GiB
NYC	2	108	19.2M	5	12.6GiB
PanCreactomy1	1	29	9.2M	2	2.0GiB
PanCreactomy2	2	58	18.3M	11	4.1GiB
Physicians	1	28	9.2M	1	1.7GiB
Provider	8	224	73.2M	46	13.6GiB
RealEstate1	2	56	39.1M	9	10.6GiB
RealEstate2	7	189	66.4M	23	17.1GiB
Redfin1	4	176	12.1M	5	5.3GiB
Redfin2	3	132	9.1M	4	4.0GiB
Redfin3	2	94	6.5M	3	3.0GiB
Redfin4	1	48	3.3M	2	1.5GiB
Rentabilidad	9	1266	3.6M	35	3.3GiB
Romance	2	24	3.2M	3	1.0GiB
SalariesFrance	13	695	16.2M	32	12.6GiB
TableroSistemaPenal	8	174	25.3M	23	5.3GiB
Taxpayer	10	280	91.5M	22	17.1GiB
Telco	1	181	2.9M	1	2.3GiB
TrainsUK1	4	87	12.9M	8	3.9GiB
TrainsUK2	2	74	31.1M	1	12.2GiB
USCensus	3	1557	9.4M	8	13.6GiB
Uberlandia	1	81	7.6M	24	6.4GiB
Wins	4	2198	2.1M	13	3.9GiB
YaleLanguages	5	150	5.8M	13	1.5GiB
Total	206	13395	988.9M	646	386.5GiB

Table 3.1: Public BI benchmark workbooks

3.1 Benchmark analysis

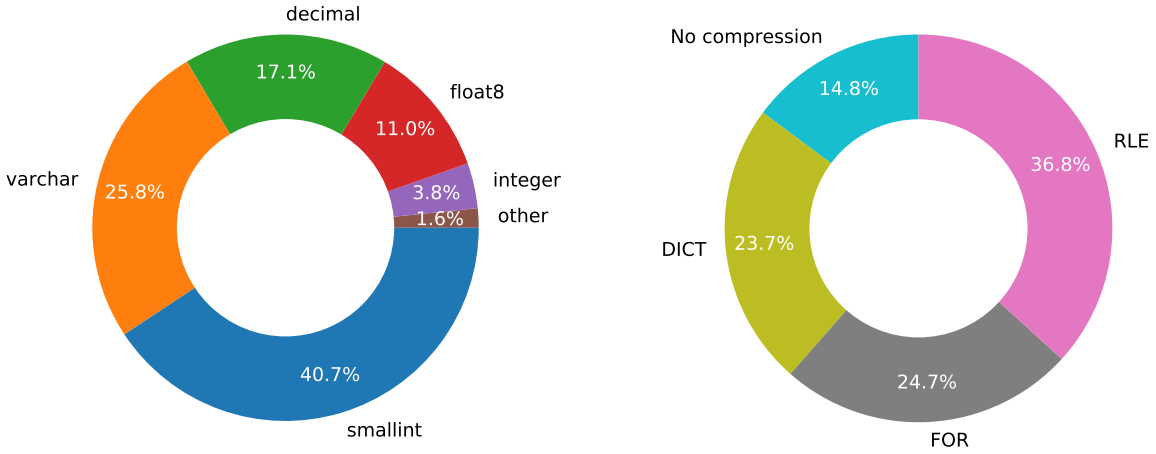
3.1.1 General characterisation

We started by analyzing the tables in terms of number or rows, columns and size. Figure 3.1 shows the distribution of these metrics over the tables. 40% of the tables have less than 1 million rows and 50% of the tables have between 1 and 10 million rows. 90% of the tables have less than 80 columns. In terms of size, the majority of the tables have less than 3GB (uncompressed), but there are also larger tables, up to 14GB.



(a) Row distribution **(b)** Column distribution **(c)** Data size distribution

Figure 3.1: Row, column and size distribution

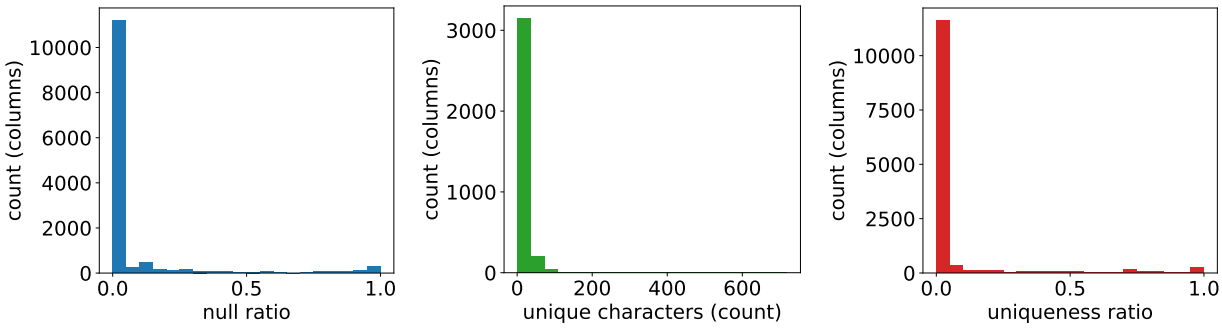


(a) Datatypes distribution **(b)** Compression potential

Figure 3.2: Column statistics (percentage of columns)

Figure 3.2a shows the distribution of datatypes across columns. The majority of the columns are numeric (73%), followed by strings (25%). The remaining columns are: `DATE` (0.86%), `TIMESTAMP` (0.41%), `BIGINT` (0.17%), `BOOLEAN` (0.1%), `TIME` (0.05%).

Given the focus of this thesis, we are interested in how suitable is the data for compression. We evaluated the potential of each column for compression with existing lightweight schemes: Run Length Encoding (RLE), Frame of Reference (FOR), Dictionary encoding (DICT). The evaluation was performed using the compression estimators defined in 5.2.2 Cost model: compression estimators, following the methodology presented in 6.2 Estimator evaluation methodology. In short, we estimated the size of each column as compressed with RLE, FOR, DICT and uncompressed—based on a sample—and marked the column as a candidate for the method that gave the smallest size. The results are presented in Figure 3.2b. 85% of the columns are good candidates for compression with lightweight schemes, while only 15% of the columns are better left uncompressed. We applied DICT for `VARCHAR` columns and RLE and FOR for `numeric` columns. Most of the numeric columns are good candidates for RLE and FOR and the vast majority of `VARCHAR` columns are dictionary compressible. The purpose of this analysis was to estimate the compression potential of the datasets in the benchmark. A more thorough analysis showing compression ratios and sizes is performed in 6.3 Results and discussion.



(a) Null ratio **(b)** Unique characters **(c)** Uniqueness ratio

Figure 3.3: Data characterisation

Figure 3.3 shows metrics extracted with the `statdump` command after loading the data into VectorWise (10). The percentage of null values is low for most of the columns. Figure 3.3b shows the number of unique characters per column—for `VARCHAR` columns: almost all of them have less than 100 unique characters. Only 8 columns have more than 255 unique characters and they contain social media content: posts or comments with hashtags and emoticons. Figure 3.3c shows the uniqueness ratio of all the columns, irrespective of their datatype. The uniqueness of a column is computed as the number of unique values divided by the total number of values: $\frac{unique_{count}}{total_{count}}$. Note that these metrics were computed based on the entire data and not based on a sample, therefore they are accurate. We notice how

almost all of the columns have a very low uniqueness, indicating a high degree of repetition. This property makes the data suitable for compression and confirms the compression estimation results.

3.1.2 Manual analysis

So far we showed a general characterisation of the benchmark, mostly from a statistical point of view. In order to better understand the properties of real data and to gain more insight about it, we manually searched for patterns—common ways that users represent the data—with the purpose of finding opportunities for more compact representations. Below is a list of our findings:

- empty/missing values that are not nulls—e.g. empty quotes, whitespace characters;
  examples: Eixo_1[1]: comunidade_quilombola, unidade_demandante
- leading/trailing whitespace, some with the purpose of ensuring a common length of the values on `VARCHAR` columns;
  examples: CommonGovernment_1[2]: contract_num, primary_contract_piid
- numbers and dates stored as strings in `VARCHAR` columns;
  examples: CommonGovernment_1[2]: contract_signeddate, agency_code
- strings with fixed structure composed of substrings from different distributions—e.g. emails, urls, strings starting with a constant and ending in a number;
  examples: CommonGovernment_1[2]: co_name, contract_num
- correlations between columns—mostly as categorical variables, but also numeric correlations; special cases: identical columns;
  examples: CommonGovernment_1[2]: short_name, ag_name, description

All of these patterns are inefficient representations of the information, in terms of data storage. Some of them could have been avoided by the user (e.g. choosing the proper datatype for numeric values), while others are determined by the nature of the data itself. These patterns represent compression opportunities that cannot be exploited through existing compression schemes.

[1] `https://github.com/cwida/public_bi_benchmark/blob/master/benchmark/Eixo/samples/Eixo_1.sample.csv`

[2] `https://github.com/cwida/public_bi_benchmark/blob/master/benchmark/CommonGovernment/samples/CommonGovernment_1.sample.csv`

3.1.3 Conclusion

The conclusion that we can draw from the analysis we performed on the Public BI benchmark is that real data is redundant and represented in inefficient ways. It is already suitable for compression with existing lightweight methods, but it has a considerable untapped compression potential that could be exploited if the data had a different representation.

4

Compression model

Whitebox compression is a compression model for columnar database systems. Its purpose is to represent data more compactly through elementary operator expressions. These operators—chained together into expression trees—form the expression language used for data representation. This chapter describes the *whitebox compression* model, its expression language and operators, the structure of expression trees and their evaluation.

4.1 Expression language

The *whitebox* compression model represents logical columns as composite functions of physical columns. We refer to these functions as *operators*. With respect to databases, logical columns are columns as seen by the database user, containing the data in its original format. Physical columns contain the physical representation of the data as it is stored on disk, in a different format.

Formally, we define an operator as a function o that takes as input zero or more columns and optional metadata information and outputs a column:

$$o\colon [C \times C \times ...] \times [M] \to C \tag{4.1}$$

The domain of o is composed of columns and metadata and the codomain is columns. A column is defined by its datatype d and the values that it contains V. The metadata is structured information of any type.

We defined our expression language based on a set of elementary column representation types, each having an associated operator. They are listed in Table 4.1.

The concatenation of two string columns c_a and c_b is the concatenation of each pair of values v_a and v_b. E.g. `"123abc"` $= concat($`"123"`, `"abc"`$)$, where $v_a =$ `"123"` and $v_b =$ `"abc"`.

Representation	Operator		
Concatenation of 2 or more columns	$concat$:	$C \times C \times [...] \to C$	$c = concat(c_a, c_b, [...])$
Formatting of another column	$format$:	$C \times M \to C$	$c = format(c_a, m_{format})$
Direct mapping of another column	map:	$C \times M \to C$	$c = map(c_a, m_{map})$
Constant representation	$const$:	$M \to C$	$c = const(m_{const})$

Table 4.1: Elementary column representation types

The representation of a string column c_a as a formatted non-string column c_b consists of the individual values v_b formatted as strings based on the format string metadata m_{format}. This can be seen as datatype change. E.g. `"-12000"` $= format($`-12000`$,$ `"%d"`$)$, where $v_b =$ `-12000` and $m_{format} =$ `"%d"` . The direct mapping representation of a column c_a as a column c_b through the mapping m_{map} is a key-value lookup in a dictionary-like data structure. E.g. `"valueoncolumnA"` $= dict[$`"valueoncolumnB"`$]$, where `valueoncolumnB` is the key and `"valueoncolumnA"` is the value in the dictionary `dict`. The constant representation of a column indicates that all its values are equal to the constant value m_{const}. The *const* operator just returns m_{const}.

These operators and transformations can be composed, resulting in operator expressions. For example, the logical columns A and B in Table 4.2, can be represented as composite functions of the physical columns in Table 4.2, through the following expressions:

$$\begin{aligned} A &= concat(map(P, dict_{AP}), const(\texttt{"\_"}), format(Q, \texttt{"\%d"})) \\ B &= map(P, dict_{BP}) \end{aligned} \tag{4.2}$$

A	B	P	Q
`"GSA_8350"`	`"GENERAL SERVICES ADMINISTRATION"`	0	8350
`"GSA_8351"`	`"GENERAL SERVICES ADMINISTRATION"`	0	8351
`"HHS_2072"`	`"HEALTH AND HUMAN SERVICES"`	1	2072
`"TREAS_4791"`	`"TREASURY"`	2	4791
`"TREAS_4792"`	`"TREASURY"`	2	4792
`"HHS_2073"`	`"HEALTH AND HUMAN SERVICES"`	1	2073
`"GSA_8352"`	`"GENERAL SERVICES ADMINISTRATION"`	0	8352
Logical columns		Physical columns	

Table 4.2: Whitebox representation example: data

We observe that column A has the following structure: a dictionary compressible prefix and a numeric suffix, separated by a the '`_`' character. If we store these logical parts separated into 3 columns C_{prefix}, C_{delim}, C_{suffix}, we can represent column A as their concatenation. Since C_{prefix} has repeated values, we can represent it more compactly as the

key	value
0	"GSA"
1	"HHS"
2	"TREAS"

$dict_{AP}$

key	value
0	"GENERAL SERVICES ADMINISTRATION"
1	"HEALTH AND HUMAN SERVICES"
2	"TREASURY"

$dict_{BP}$

Table 4.3: Whitebox representation example: metadata

mapping of column P—containing dictionary keys—and the dictionary $dict_{AP}$—presented in Table 4.3. We can represent C_{delim} through the *const* operator since all its values are equal to '_'. C_{suffix} contains numbers stored in strings. We can store these values more compactly as numbers, by changing the column datatype. Therefore, we represent $C_{(suffix)}$ based on the numeric column Q, through the $format$ operator, with the format string "%d". We move our attention to column B and observe that it is correlated with column C_{prefix}—and implicitly also to column P. We can therefore represent B as the mapping of column P and the dictionary $dict_{BP}$—presented in Table 4.3. In the end, we store only the physical columns P and Q and the metadata: $dict_{AP}$, $dict_{BP}$ and the constant string "_". The original values on the logical columns A and B can be reconstructed by evaluating the expressions in Equation 4.2.

So far, we described 4 column representation types and their associated operators: *concat*, $format$, *map* and *const*. The *whitebox compression* model does not limit itself to these representation types. It is a generic model and supports any type of column operators (e.g. mathematical operators like addition or multiplication). A practical example is the *whitebox* version of the Frame of Reference compression method: $const(reference) + C_{diff}$, where + is the numeric addition/sum operator, *reference* is the reference value and C_{diff} is the physical column containing the differences between the original values and *reference*.

The purpose of *whitebox compression* is to represent data more compactly through elementary operator expressions similar to the ones presented above. However, there are a multitude of different possible representations of the same logical columns, each one giving a different result. We will describe the optimization problem of finding the best representation for a set of columns in 5.2.1 Optimization problem.

These operator expressions create more compact representations of logical columns, exploiting the underlying compression opportunities in the data. We showed how we can remove redundancy from data by representing columns as functions of other columns through the *map* operator and how we can store numeric values in more suitable datatypes through

the $format$ operator. We are able to decompose string columns into subcolumns with values from different distributions, thus enabling independent representations. Finally, the key factor of the *whitebox* model is that it allows recursive representation of columns, ultimately leading to improved compression ratios.

4.2 Expression tree

The operators presented so far are useful for describing the data representation and for transforming the physical data into its original logical format. We call this process *decompression*. In practice, we need to transform the logical data into its physical format first—*compression*. The compression process requires a different expression, one that represents the physical columns as functions of the logical columns, through the inverse operators of the ones presented until now. Table 4.4 presents the compression operators types.

Transformation	Operator		
Split a column into multiple columns	$split$:	$C \to C \times C \times [...]$	$split(c) = c_a, c_b, [...]$
Change datatype of a column	$cast$:	$C \times M \to C$	$cast(c, m_{datatype}) = c_a$
Direct mapping of another column	map:	$C \times M \to C$	$map(c, m_{map}) = c_a$
Consume a column (constant/correlation)	$consume$:	$C \times M \to \emptyset$	$consume(c, m) = \emptyset$

Table 4.4: Compression operator types

All these operator types and their practical implementations will be discussed in detail in 5.1 Pattern detection. For now, we are interested in their definition. We notice how the formal definition of the compression operator is different from the one of the compression operators:

$$o\colon C \times [M] \to [C \times C \times ...] \tag{4.3}$$

The compression operators take as input a single column and compression metadata information and output 0 or more columns. Because of the multiple column output, representing physical columns as composite functions of logical columns is not straightforward. Therefore, we introduce the concept of *expression trees*, as an alternative representation instead of the nested operator expressions.

Expression trees are tree-like structures with 2 types of nodes: *column nodes* and *operator nodes*. We also use the term *expression node* to refer to the *operator nodes*—they are interchangeable. An *expression tree* is composed of alternating levels of *column* and *operator* nodes. Root nodes are always *column nodes*. Leaf nodes can be either *column nodes* or *operator nodes*—in the case of operators that do not output any column. An *operator*

node in an *expression tree* is the equivalent of and operator in an operator expression: it has input columns—connected through incoming edges—and output columns—connected through outgoing edges. *Expression trees* are used in the compression and decompression processes as more practical alternatives to the operator expressions. To better understand the similarities between the two, we created the equivalent *expression tree* of the operator expressions for columns A and B in Equation 4.2. Recall the expressions: $A = concat(map(P, dict_{AP}), const(\texttt{"\_"}), format(Q, \texttt{"\%d"}))$ and $B = map(P, dict_{BP})$. The equivalent *expression tree* is presented in Figure 4.1.

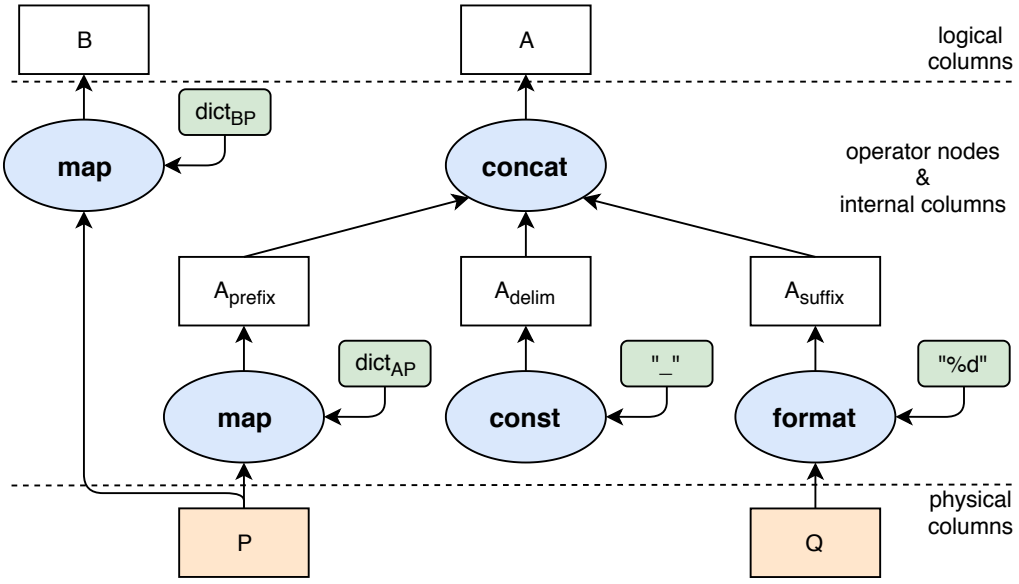


Figure 4.1: Expression tree representation

The first thing to notice is that the *expression tree* is not actually a tree, but a directed acyclic graph (DAG) with 2 root nodes. However, we chose to stick to the term *expression tree* instead of graph, since it is more intuitive. In this case the graph is connected, but in other cases it can have multiple connected components. For example, imagine that in our example we had an additional logical column C that is represented as a function of a physical column S, without any connection with the columns or operators used to represent columns A and B. Then, our graph will have 2 connected components.

Besides the graph-like structure, the *expression tree* is an equivalent representation of the operator expressions. We notice the similarities between the operator nodes and the operators in the nested expressions and the alternating levels of columns and operators. A noticeable difference is the additional columns A_{prefix}, A_{delim} and A_{suffix}. These are non-materialized internal columns that make the recursive representation possible. In terms

of representation type, this is a *decompression tree*, since the root nodes are the physical columns P and Q and the expression nodes are decompression operators. The *compression tree* will have the same structure, only that the root nodes will be the logical columns A and B and the expression nodes will be compression operators—the inverse functions of the decompression operators. The metadata will also differ. The *compression tree* can be derived from the *decompression tree* and vice versa, by using the inverse operators and transforming the metadata where it is necessary.

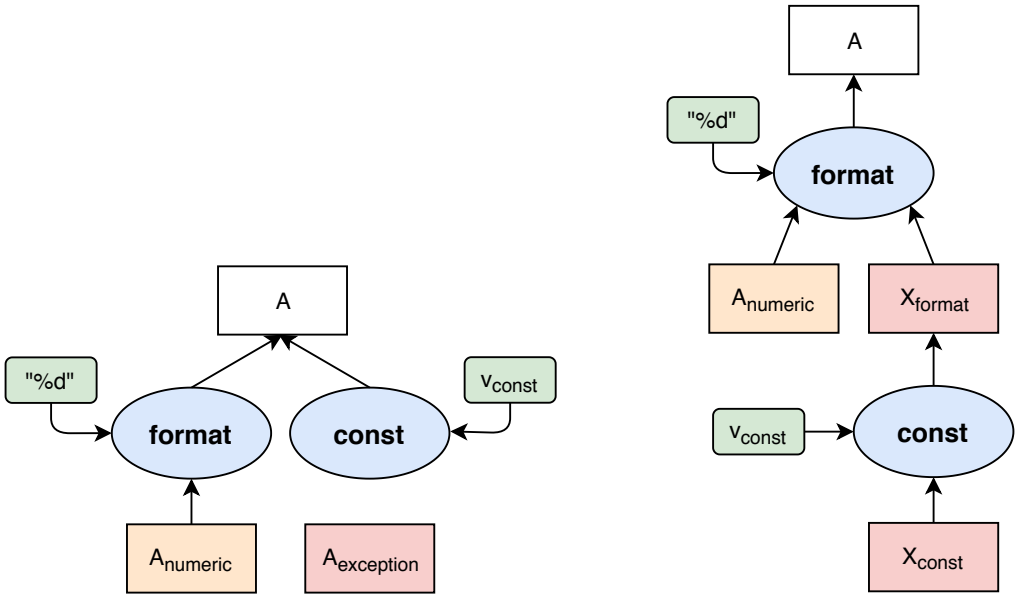
There is the case that different subsets of the values on a column come from different distributions and cannot be represented through the same expression. E.g. a string column where odd rows contain the same constant value and even rows are numbers. These situations are common in real data, as we have seen in the Public BI benchmark, where there are not many columns for which a single representation perfectly fits all the values. One option to accommodate these cases is to allow a single column to have multiple representations. The representation of the column is then the union of its multiple representations. In terms of *decompression trees*, the column will have multiple incoming edges, each one from a different operator. These multi-representation structures need to be explicitly handled in the *expression tree* evaluation process (described in 4.4 Compression and decompression). The second option of handling these cases is through recursive representation of *exception columns*, which is discussed in 4.3 Exception handling.

4.3 Exception handling

While analyzing the Public BI benchmark in search for patterns and *whitebox compression* opportunities we noticed that columns where all values perfectly fit the same pattern/representation are not very common. Instead there is a smaller or larger subset of values that do not fit the dominant pattern of the column. Let us take for example the data in Table 4.2. Image that a few values on column A did not have the `prefix-delim-suffix` structure and instead they were just arbitrary strings. Then, the representation in Figure 4.1 could not be applied on the entire column. We call these values that do not match the representation: *exceptions*. In the *whitebox compression* model *exceptions* are stored on separate *exception columns*. These are nullable columns that contain `null` on positions where the value was not an exception and the original values otherwise. Conversely, the non-exception columns are also nullable and contain null on the positions of exceptions.

We defined 2 ways of handling *exceptions*: 1) through the multi-representation approach mentioned in 4.1 Expression language; 2) through recursive representation of *exception*

columns. The first option implies having an operator expression for each subset of values that requires a separate representation. The second option implies choosing a single representation, storing *exceptions* on a separate *exception column* and then recursively applying the same process on the *exceptions*. The two options are equivalent from the physical data perspective. Only the shape of the tree differs: flat and wide trees in the first case and deeper trees in the second case. An additional difference between the two options is the number of *exception columns*: the first option requires at most one *exception column* for every logical column (to store values that do not fit any pattern), while the second option implies having a separate *exception column* for each operator node in the expression tree. The two approaches are illustrated in Figure 4.2.



(a) Multi-representation

(b) Recursive exception representation

Figure 4.2: Representation options & exception handling

Figure 4.2 shows the 2 equivalent representations of a string column A, which has 2 major subsets of values: one containing numeric values and the other one a constant value. The figure on the left shows the multi-representation approach: the 2 operators are on the same level of the tree and the exceptions—i.e. the values that do not fit any of the 2 representations—are stored separately in the $A_{exception}$ column. $A_{numeric}$ and $A_{exception}$ are both physical columns. The figure on the right shows the recursive exception representation: the subset of numeric values are represented through the $format$ operator and the rest are rejected to the *exception column* X_{format}. X_{format} now contains the subset

of constant values, and is represented through the *const* operator. The remaining values which are not constant are stored in the X_{const} *exception column*. The physical columns in this case are $A_{numeric}$ and X_{const}, while A_{format} is just an intermediate non-materialized *exception column*. The 2 representations are equivalent in terms of the physical data structure: the numeric values are stored in $A_{numeric}$ and the exceptions—values that are neither numeric nor constant—are stored in the $A_{exception}$ respectively X_{const} column.

In our implementation we used a combination of the 2 approaches: select the dominant patterns in the data and represent each column through multi-representation expressions and then store the rest of the values—which do not fit the representations—in *exception columns*. If there are more opportunities left in the exceptions, recursive representation of the *exception columns* is implicitly performed by the compression learning algorithm, since they are treated as normal columns.

4.4 Compression and decompression

The evaluation of a *compression tree* on a set of logical columns—i.e. *compression*—means applying the operators on the logical values in order to generate the physical values that will be stored in the physical columns. Conversely, the evaluation of a *decompression tree* on a set of physical columns—i.e. *decompression*—is the process of applying the operators on the physical values to obtain the original data. The 2 process are similar and we will further discuss only *decompression*.

Given a table with multiple logical columns, its expression tree (graph) will have 1 or more connected components. Each component can be evaluated independently from the other components. The decompression process starts from the root nodes and evaluates the operators on the path to the target logical/physical column, in topological order. In the case of *decompression*, exceptions are handled by checking for `null` values on the *exception column*. If the value at a given position on the *exception column* is not `null` then it was an exception, otherwise the operator needs to be evaluated. A special case is when a logical data value is `null` and also an exception. For this case we use a bitmap indicating which values were `null` in the first place. In the case of *compression*, the operators are responsible for deciding which values are exceptions and which are not: if an operator throws an exception then the value is stored on the *exception column*. The *compression* and *decompression* processes are similar for the multi-representation structures defined in the previous sections. For *compression*, the decision upon which representation fits a given value is determined by the (first) operator that does not raise an exception. If all operators

raise an exception then the value is stored on the *exception column*. For *decompression*, the physical columns that do not contain `null` values indicate the representation of each value. This process is similar to the SQL `COALESCE` function (21), which returns the first non-null value in a list of expressions.

The process of evaluating *expression trees* in topological order is suitable for vectorized execution (22) and SIMD instructions since the elementary operators can be evaluated in the same order for blocks of data, generating intermediate results which fit in the cache. Alternatively, JIT compilation (22) can be used to generate compiled code for each component of the expression tree during compression time, which is then executed for each query. However, the scope of this thesis does not cover fast evaluation of expression trees. We leave this topic for future work.

5

Automatic compression learning

This chapter presents our approach for automatically learning the best representation of a set of columns. We divided this problem in two parts: 1) automatically identifying *whitebox* representation opportunities in data; 2) automatically building a compression tree that minimizes the physical size of the data. They are described in the following sections.

5.1 Pattern detection

We use the term *pattern detection* to refer to the process of identifying patterns in the data and evaluating its *whitebox* representation potential. We introduce the concept of *pattern detector*—a modular component in a generic compression learning architecture. This section presents the generic interface of a pattern detector and 5 specialized implementations of it.

5.1.1 Generic pattern detector

The purpose of a generic pattern detector is to serve as an interface in the pattern detection and compression learning processes. Given a sample of data and its schema, an instance of a generic pattern detector searches for the presence of the pattern it is specialized in. For each column that matches the pattern, it outputs metrics and metadata that will be further used in the learning phase and during compression. This generic design allows new pattern detectors to be easily tested and integrated into the system without modifying other parts of it (e.g. the learning or compression processes). This section presents the characteristics of the generic pattern detector interface, while specialized implementations of it are described in the next sections.

The pattern detection process works in 3 phases: Phase-1: *initialization*—initializes the pattern detector and creates the data structures that will be used in the next phases; Phase-2: *scanning*—scans the sample of data and gathers information and metrics about values in the sample. Phase-3: *evaluation*—aggregates the information gathered in the *scanning* phase and produces an evaluation result.

Parameters. A pattern detector takes as input the following parameters:
a) *columns*: id, name and datatype for each column
b) *compression tree*: the compression tree built so far. Used in the *select_ column* method
c) *detection log*: history of the pattern detection process. Contains information about which pattern detectors were evaluated on each column and what was the result. Used in the *select_ column* method.
Although most pattern detectors work on individual columns, some may work on multiple columns (e.g. Column correlation). For this reason we designed the generic pattern detector to search for patterns at the table level instead of individual columns.

Methods. A pattern detector must implement the following methods:
a) *select_ column*: called in the *initialization* phase—determines whether a given column will be processed by the pattern detector. The decision process is based on: 1) data type (e.g. string-specific patterns do not apply to numeric columns); 2) path in the compression tree that led to the creation of the column (e.g. Column correlation only applies to columns that are output of a Dictionary compression node); 3) history of the pattern detection process (e.g. do not evaluate a pattern detector on a column if it was already evaluated in a previous step). Column selection rules are listed in the section of each pattern detector.
b) *feed_ tuple*: repeatedly called in the *scanning* phase—processes a tuple. Data is fed to the pattern detector one tuple at a time. This method extracts information from the tuple that will be later used in the *evaluate* method.
c) *evaluate*: called in the *evaluation* phase—aggregates the information extracted from the tuples fed so far and outputs the outcome of applying the pattern to the columns. See output details below.

Output. The result of evaluating a pattern detector on a sample of data is a list of *(expression node, evaluation result)* tuples.
a) *expression node*: describes how one or more *input columns* are transformed into one or more *output columns* by applying an *operator* (see detailed description in 4 Compression model). An important part here is the operator metadata that will be used to evaluate the *expression node* in the (de)compression process (e.g. the operator metadata for the Dictionary pattern detector is the dictionary/map object).

b) *evaluation result*: contains information used in the learning process and describes how well the *input columns* of the *expression node* match the pattern. It contains the following information: 1) *coverage* (percentage of rows that the pattern applies to; i.e. $1 - exception\_ratio$); 2) *row_mask* (bitmap indicating the rows that the pattern applies to); 3) other pattern-specific evaluation results (e.g. correlation coefficient for column correlation).

Each *(expression node, evaluation result)* tuple represents a possibility of applying the pattern on a subset of the columns. The same column may be present in multiple tuples since there may be more than one option of applying the pattern to it (e.g. the Character set split pattern detector can split a column in 2 ways, because there are 2 dominant character set patterns on the column; see 5.1.4 Character set split for more details). The *row_masks* for such a column may either be mutually exclusive or partially overlap. A pattern detector only outputs results for columns that match the pattern and are likely to produce good results in the compression process according to a pattern-specific estimator (e.g. Dictionary pattern detector only outputs tuples for columns that are dictionary compressible; see 5.1.6 Dictionary for more details).

Operators. Each pattern detector provides a *compression operator* and a *decompression operator*. They are used in the compression and respectively decompression phase to evaluate the nodes in the expression trees. The generic operator is described in 4 Compression model while the pattern-specific implementations can be found in the corresponding section of each pattern detector.

5.1.2 Constant

The Constant pattern detector identifies columns that have (mostly) a single value. We refer to these columns as being constant. An additional parameter is provided in the initialization: $constant\_ratio_{min}$ which indicates the minimum ratio of the constant value, based on which a column is considered constant or not. This pattern detector works on all data types, therefore the *select_column* method always returns *True*. This is a single-column pattern detector and columns are evaluated independently. The next paragraphs describe the pattern detection process for a single column.

The *scanning* phase is responsible for building the histogram of values on the column. The *evaluation* phase selects the most common value C as the constant candidate—the value with the highest number of occurrences. The $constant\_ratio$ of the column repre-

sents the evaluation metric and is computed as follows:

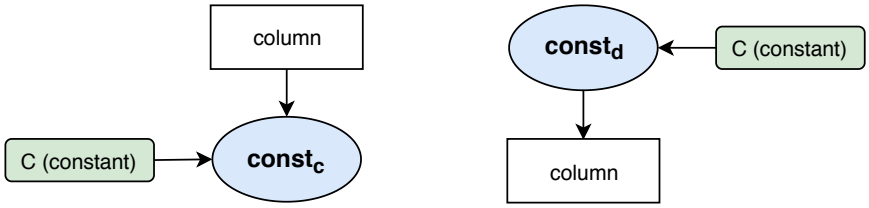
$$constant\_ratio = \frac{count_C}{count_{notnull}} \tag{5.1}$$

where:

$count_C$ = number of occurrences of the constant candidate C
$count_{notnull}$ = number of non-null values in the column

The column is considered to fit the Constant pattern if the *constant_ratio* is greater or equal to $constant\_ratio_{min}$. All values that are not equal to C are considered exceptions. The *evaluation result* is composed of the *coverage* and *row_mask*—as defined in 5.1.1 Generic pattern detector—and the *constant_ratio* as an additional pattern-specific result. The metadata needed for compression and decompression is the constant C.

The *expression nodes* for the Constant pattern are illustrated in Figure 5.1.



(a) compression **(b)** decompression

Figure 5.1: Constant expression nodes

The *compression node* takes as input the constant column and does not generate any output column. Similarly, the *decompression node* does not require any column to produce the constant column.

The compression and decompression operators are $const_c$ and $const_d$. The metadata they require is the constant C. $const_c$ takes as input a value v and checks whether it is equal to C. If *True*, then it returns nothing. Else, it raises an *OperatorException* indicating that v is an exception and should be added to the exception column. $const_d$ does not take any input value and returns the constant C.

The benefit of the Constant representation scheme is clear: we avoid storing a column on disk. Dictionary encoding could also be used to compress constant columns, however it still stores a column with dictionary ids as opposed to no column at all. A more generic alternative representation for constant columns would be a *whitebox* version of RLE that supports any data types.

5.1.3 Numeric strings

This pattern detector searches for numbers stored in string columns. Once found, it changes the column data type to a numeric one, while also preserving the string format of the numbers in additional columns. The purpose is to optimize the data type and create opportunities for numeric compression schemes. An additional purpose is to enable the push-down of numeric range predicates in data scans during query evaluation (the implementation and evaluation of all execution-related matters was kept out of scope for this thesis). The *select_ column* method only returns *True* for `VARCHAR` columns. This is a single-column pattern detector and columns are evaluated independently. The next paragraphs describe the pattern detection process for a single column.

The *scanning* phase checks for each value (v_{string}) whether it can be parsed as a number ($v_{numeric}$). If *True*, it extracts the format information and checks whether the original string value can be reconstructed. The *evaluation* phase chooses an appropriate numeric data type (e.g. `DECIMAL(p,s)`) based on the type and range of the values selected in the *scanning* phase. More details about the format preserving and data type inference processes are presented in the following paragraphs.

Format preserving. While analyzing the Public BI benchmark we noticed that the majority of string values v_{string} with a different format than their numeric representation $v_{numeric}$ contain leading or trailing characters (e.g. zeros, whitespace). Therefore, we based our format preserving technique on 3 components—*prefix*, $v_{numeric}$, *suffix*—as follows:
Step-1: cast v_{string} to number to obtain $v_{numeric}$
Step-2: check whether $abs(v_{numeric})$ is a substring of v_{string}. If *False*, the format cannot be preserved. Otherwise, v_{string} has the following format: `${`*prefix*`}`$abs(v_{numeric})$`${`*suffix*`}`, where *prefix* and *suffix* can be any strings, including the empty string.
Step-3: extract the *prefix* and *suffix* and store them as format information together with $v_{numeric}$.

v_{string} can be now reconstructed by concatenating the *prefix*, $v_{numeric}$ and *suffix* values. Table 5.1 illustrates a few examples that the format preserving technique covers (`"_"` characters represent spaces).

We currently do not yet support—and consider exceptions—the following: 1) numbers in scientific notation; 2) other notations or abbreviations (e.g. `.12` instead of `0.12`, `1_000_000` instead of `1000000`, etc.).

Data type inference. The purpose of this pattern detector is to store the numeric string values in a numeric column, which requires a numeric data type. We chose two data

v_{string}	$v_{numeric}$	*prefix*	$abs(v_{numeric})$	*suffix*	*description*
`"1.23"`	`1.23`	`""`	`1.23`	`""`	no format information
`"1.2300"`	`1.23`	`""`	`1.23`	`"00"`	trailing zeros
`"000.1"`	`0.1`	`"00"`	`0.1`	`""`	leading zeros
`"+10"`	`10`	`"+"`	`10`	`""`	+ sign
`"-10"`	`-10`	`"-"`	`10`	`""`	negative number
`"-000.5"`	`-0.5`	`"-00"`	`0.5`	`""`	negative number with leading zeros
`"__54\t\t"`	`54`	`"__"`	`54`	`"\t\t"`	leading and trailing whitespace

Table 5.1: Numeric string format preserving examples

types that can be used to represent all numbers: `DECIMAL(p,s)` and `DOUBLE`. During the *scanning* phase all $v_{numeric}$ values are interpreted as decimals. The maximum number of digits before and after the decimal point—$integer_{dmax}$ and $fractional_{dmax}$—are determined. In the *evaluation* phase, the parameters p (precision) and s (scale) of the `DECIMAL(p,s)` data type are computed as follows:

$$\begin{aligned} p &= integer_{dmax} + fractional_{dmax} \\ s &= fractional_{dmax} \end{aligned} \tag{5.2}$$

The final datatype of the numeric column is determined as follows:

$$datatype = \begin{cases} \texttt{DECIMAL(p,s)} & \text{if } p \leq p_{max} \\ \texttt{DOUBLE} & \text{else} \end{cases} \tag{5.3}$$

where:

p_{max} = maximum decimal precision supported by the system

The maximum precision value is configurable and necessary as database systems enforce it—MonetDB: $p_{max} = 38$ (23), VectorWise: $p_{max} = 39$ (24).

A string value v_{string} is considered an exception if any of the following conditions is not satisfied:

1) v_{string} cannot be parsed as a numeric value $v_{numeric}$

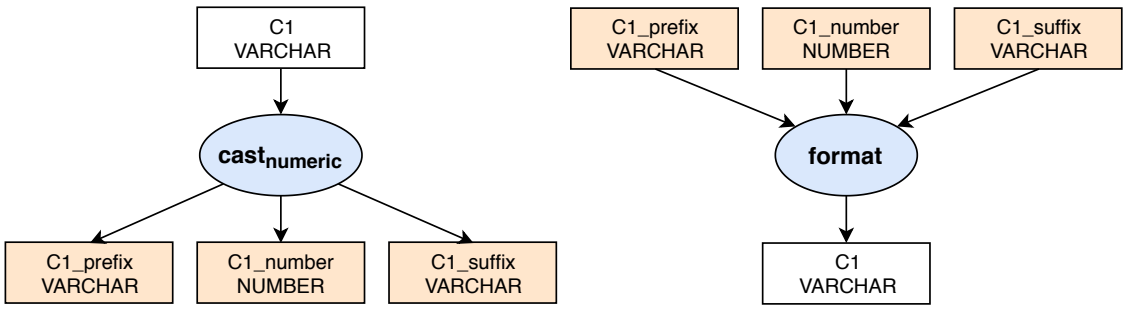
2) v_{string} cannot be reconstructed from its numeric value $v_{numeric}$ and format information

3) $v_{numeric}$ exceeds the numeric data type selected in the *evaluation* phase

There is no strict filtering condition that determines whether a column fits this pattern or not. The pattern detector outputs a result for all columns with *coverage* grater than 0, leaving the responsibility of choosing which columns to compress with this scheme to the compression learning algorithm. The *evaluation result* is composed of the *coverage*

and *row_ mask*—as defined in 5.1.1 Generic pattern detector. No additional metadata is necessary for the compression and decompression processes.

The *expression nodes* for the Numeric strings pattern are illustrated in the Figure 5.2.



(a) compression **(b)** decompression

Figure 5.2: Numeric strings expression nodes

The *compression node* takes as input the *string* column and generates three output columns: one for storing the numeric values and the other two for storing the format information as *prefix* and *suffix*. The *decompression node* takes as input the *numeric*, *prefix* and *suffix* columns and reconstructs the original string column.

The compression operator $cast_{numeric}$ takes as input v_{string} and tries to represent it as the 3 components—*prefix*, $v_{numeric}$ and *suffix*—as described above. It then checks whether $v_{numeric}$ fits in the inferred data type of the numeric column. If everything succeeds it returns the 3 values, else it raises an *OperatorException* indicating that v_{string} should be added to the exception column. The decompression operator *concat* reconstructs v_{string} by concatenating *prefix*, $abs(v_{numeric})$ and *suffix*. The two operators do not require any metadata, except for the numeric column data type.

There are three main benefits brought by this representation scheme: 1) using an optimal numeric data type instead of `VARCHAR`, leading to smaller size on disk; 2) creating opportunities to further compress the numeric column with numeric compression schemes; 3) creating push-down opportunities for numeric predicates (e.g. for the query `SELECT * FROM t WHERE (CAST c AS INT) < 42` it allows push-down on the physical numeric column, even if `c` is `VARCHAR`; this example may look contrived, but in the case of dates stored as strings it is very common). Moreover, we noticed that in practice the *prefix* and *suffix* columns get further compressed as Constant or with Dictionary encoding.

5.1.4 Character set split

The Character set split pattern detector searches for string columns where all values have the same structure in terms of character set sequences. A few examples are listed in Table 5.2 (`"_"` characters represent spaces).

customer	account	transaction
customer0001	HHSI2452____	{9AE2B97B-69D0-4A5E}
customer0002	TIRNO1017___	{891F7B57-80C4-4BAA}
customer0003	TIRNO168823_	{7C652BE6-947F-4AFF}
...	...	...
customer4735	HHSI3391____	{41AA2723-BA9D-465C}
customer4736	TIRNO41163__	{88635130-6292-4C04}

Table 5.2: Character set split examples

The *customer* column contains values that start with the constant `customer` (charset: letters) and end with a number (charset: digits). All values on the *account* column have the following structure: letters+digits+whitespace. The *transaction* column contains 3 hex numbers (charset: hex digits) separated by dashes and enclosed in brackets (charset: delimiters).

The purpose of the Character set split pattern detector is to split these columns into multiple columns based on the structure given by the character sets. For example, the *customer* column is split into 2 columns: one containing the `"customer"` constant value and one containing the numbers at the end. The *transaction* column is split into 7 columns: 1 for the open bracket, 1 for the closed bracket, 2 for the dashes and 3 columns for the hex numbers.

The pattern detector receives two additional parameters: 1) $coverage_{min}$—used in the *evaluation* phase to filter results; 2) a list of character sets (e.g. `[a-zA-Z]`, `[0-9a-fA-F]`, `[_-{}()]`, etc.). The character sets can contain any characters, but they must be disjoint sets. An additional character set—the *default* charset—is implicitly defined to represent all the other characters that are not in the sets provided as parameter. Multiple instances of this pattern detector can be used at the same time with different lists of character sets, leaving the learning algorithm to choose the one that provided the best results. This pattern detector works only with `VARCHAR` columns. It is a single-column pattern detector and columns are evaluated independently. The next paragraphs describe the pattern detection process for a single column.

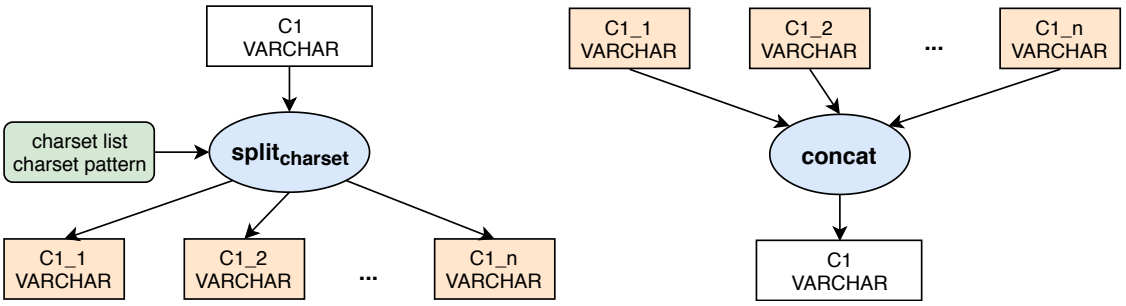
We define the *get_charset_pattern* function as follows: *input*: a string value (v_s) and a list of character sets ($charset_{list}$); *output*: the $charset_{pattern}$ of v_s. The $charset_{pattern}$ is a string that encodes the structure of v_s based on the provided $charset_{list}$. The *get_charset_pattern* function creates the $charset_{pattern}$ by replacing groups of consecutive characters from the same charset with a placeholder. For example, a group of 3 digits will be replaced by the placeholder `D`. The result of applying the *get_charset_pattern* function on the columns in Table 5.2 is shown in Table 5.3 (`"?"` is the default placeholder).

column	customer	account	transaction
$charset_{list}$	`[a-zA-Z], [0-9]`	`[a-zA-Z],[0-9]`	`[0-9a-fA-F]`
placeholders	`L, D, ?`	`L, D, ?`	`H, ?`
v_s	`customer0001`	`HHSI2452____`	`{9AE2B97B-69D0-4A5E}`
$charset_{pattern}$	`LD`	`LD?`	`?H?H?H?`

Table 5.3: Charset structure examples

In the examples in Table 5.2 the values on each column give the same $charset_{pattern}$ because they have the same structure. However, in practice this is not often the case. While analyzing the Public BI benchmark we distinguished the following cases, which depend both on the data values and the $charset_{list}$:
1) no structure: many different $charset_{pattern}$ values, with uniform distribution
2) fixed structure: a single $charset_{pattern}$
3) fixed structure with some exceptions: a single dominant $charset_{pattern}$
4) more than 1 fixed structure: a few (1-3) dominant $charset_{pattern}$ values
Among these cases, we are interested in the last 3, the first one being filtered in the *evaluation* phase.

The *scanning* phase applies the *get_charset_pattern* function to all values and builds the histogram of the resulting $charset_{pattern}$ values. The *evaluation* phase treats each $charset_{pattern}$ as an independent result as follows: 1) computes the *coverage* as the number of occurrences over the total number of non-null values; 2) computes the *row_mask* by marking the rows where the $charset_{pattern}$ is present; 3) computes an additional evaluation metric: average percentage of chars that fit in one of the charsets (100% - percentage of chars in the default charset). It then filters and returns the results that have a *coverage* greater than $coverage_{min}$, leaving the learning algorithm to decide which result or combination of multiple results is the best. The metadata necessary for compression is composed of the $charset_{list}$ and the $charset_{pattern}$. The latter is different for each result. Decompression does not require any metadata.



(a) compression **(b)** decompression

Figure 5.3: Character set split expression nodes

The *expression nodes* for the Character set split pattern are illustrated in Figure 5.3.

The *compression node* takes as input the string column and the compression metadata and outputs n `VARCHAR` columns, where $n = len(charset_{pattern})$—one column for each character group. The *decompression node* takes as input the n columns and concatenates them to reconstruct the original input column.

The compression operator $split_{charset}$ takes as input the string value v_s and the compression metadata: $charset_{list}$ and $charset_{pattern}$. It applies the $get\_charset\_pattern$ function on v_s to obtain $charset_{pattern}\_v$. It then compares $charset_{pattern}\_v$ with $charset_{pattern}$ to see if v_s has the correct structure. If they are not equal, it raises an *OperatorException*, indicating that v_s is an exception. Otherwise, it splits v_s into n substrings—each one corresponding to a charset group—and returns them. The decompression operator *concat* receives as input n substrings and concatenates them to reconstruct the original value v_s.

This representation scheme does not bring any compression benefit by itself. Instead, it creates compression opportunities by splitting columns into sub-columns that can be recursively compressed with other techniques. E.g. the columns in Table 5.2 can be compressed as follows:

1) the *customer* column is first split into 2 columns: *letters* and *digits*. Then the *letter* column is represented as a Constant and the *digits* column is represented through a Numeric strings node and further compressed with numeric compression schemes (e.g. DELTA).

2) the *account* column is split into 3 columns: *letters*, *digits*, *spaces*. The *letters* column gets compressed with Dictionary encoding. The *digits* column is compressed similarly to the one in the *customer* column. The *spaces* column may be compressed through a custom representation scheme that identifies values with a single repeated character and only stores their number, or, alternatively, through Dictionary encoding, since the number of different space padding sequences is most likely small.

3) the *transaction* column is split into 4 *delimiter* columns and 3 *hex* columns. All *delimiter* columns have only 1 constant character and are compressed as Constant. The *hex* columns can be represented through an implementation of Numeric strings that supports hexadecimal numbers, leading to 3 numeric columns that can benefit from numeric compression schemes.
In all three cases the Character set split representation transforms an uncompressible `VARCHAR` column into multiple compressed physical columns, significantly reducing the disk space required to store the data.

5.1.5 Column correlation

The purpose of the Column correlation pattern detector is to find correlations between columns and create mappings that can be used to represent a column as a function of another. The technique we describe is generic and can be applied to any data type. This pattern detector receives an additional parameter $corr\_coef_{min}$, which indicates the minimum degree of correlation between two columns and is used to filter results in the *evaluation* step. This is a multi-column pattern detector—it needs to analyze multiple columns at the same time.

Correlation types. From a statistical point of view, we can distinguish 3 types of correlations: between continuous, discrete and categorical variables. The first 2 apply to numeric values and can measure numerical similarities (e.g. sales increased as the marketing budget increased). The latter is less restrictive and works with any type of values. Categorical variables contain a finite number of values (e.g. payment method or customer satisfaction levels: low, high, very high, etc.) These can be further categorized into ordinal and nominal variables. Ordinal values have a natural ordering (e.g. tiny, small, large, huge) while nominal values have no ordering (e.g. names or colors).

While analyzing the Public BI benchmark we noticed that most correlations are between nominal values. Table 5.4 shows a representative example from the *CommonGovernment_1* dataset.

The table contains a name and its abbreviation repeated over multiple rows. The 2 columns are perfectly correlated because any value on one of the columns always has the same corresponding value on the other column (e.g. `TREAS` on the *short_name* column always determines `DEPARTMENT OF TREASURY` on the *ag_name* column).

Base on this observation, we decided to limit the scope of the Column correlation pattern detector to nominal categorical variables and interpret the values on all columns as nominal values. We leave the other correlation types as future work.

short_name	ag_name
GSA	GENERAL SERVICES ADMINISTRATION
GSA	GENERAL SERVICES ADMINISTRATION
HHS	DEPARTMENT OF HEALTH AND HUMAN SERVICES
TREAS	DEPARTMENT OF TREASURY
TREAS	DEPARTMENT OF TREASURY
HHS	DEPARTMENT OF HEALTH AND HUMAN SERVICES
GSA	GENERAL SERVICES ADMINISTRATION

Table 5.4: CommonGovernment_1 nominal correlation

Correlation coefficient & mapping. The Column correlation pattern detection process works in 3 phases: 1) compute the correlation coefficient ($corr\_coef$) for all pairs of 2 columns (c_{source}, c_{target}); 2) select the pairs with $corr\_coef$ higher than $corr\_coef_{min}$; 3) create the correlation mapping (map_{obj}) that can be used to determine c_{target} based on c_{source}.

The measure of association between 2 nominal categorical variables can be determined through existing statistical methods like Cramer's V (25) or the Uncertainty coefficient Theil's U (26). Both methods are suitable for finding correlations between columns, but they only provide the measure of how well the columns are correlated as a number between 0 and 1—the correlation coefficient $corr\_coef$. We still need the correlation mapping to be able to represent one column as a function of another.

We define the correlation mapping map_{obj} between 2 columns c_{source} and c_{target} as a dictionary with (v_{source}, v_{target}) key-value pairs, where v_{source} are values from c_{source} and v_{target} are values from c_{target}. Formally, the mapping is a total function $f: S \to T$, where an element from T can be mapped to 0, 1 or multiple elements in S, as depicted in Figure 5.4.

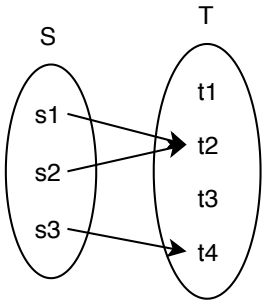
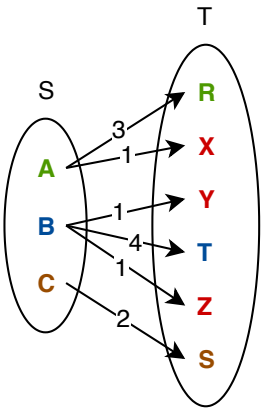


Figure 5.4: Mapping function

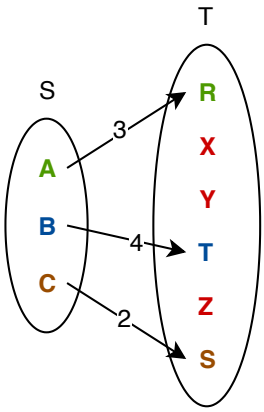
The process of creating the mapping between 2 columns c_{source} and c_{target} is depicted in Figure 5.5.

c_{source}	c_{target}
A	R
A	R
A	X
B	Y
B	T
B	T
A	R
C	S
B	T
B	Z
C	S
B	T

(a) Data values



(b) Step-1



(c) Step-2

Figure 5.5: Column correlation mapping

The correlation mapping is built in 2 steps. Step-1: for every unique value in c_{source} build the histogram of values in c_{target} it is associated with. In the example in Figure 5.5, `B` is associated with `T` 4 times and with `Y` and `Z` one time. Step-2: for every unique value in c_{source} select the value in c_{target} it is associated with the most (i.e. the one with the highest number of occurrences). In the example, `R` is the most common value for `A`, `T` is the most common value for `B` and `S` is the only value associated with `C`. These selected pairs of values form the correlation mapping map_{obj}.

We consider exceptions all the values from c_{target} that are not present in map_{obj}. The $exception\_ratio$ is given by the total number of values on c_{target} that do not respect the correlation mapping.

We observe that two columns that have $exception\_ratio = 0$ are perfectly correlated (i.e. all values in c_{target} can be determined from c_{source}). Conversely, two columns with $exception\_ratio = 1$ are completely uncorrelated. We can define, then, an alternative correlation coefficient: $corr\_coef = 1 - exception\_ratio$. We compared it with Cramer's V and Theil's U by computing the correlation coefficient for tables in the Public BI benchmark. Figure 5.6 shows the coefficients resulted from the compression learning process on the *YaleLanguages_1* table. Each image shows the correlation coefficients between every pair of columns (note that these are not logical columns, but intermediate columns resulted from the learning process—more details will follow in chapters 5 and 6). Bright colors indicate high coefficients and dark colors low coefficients. We notice how all three images have a similar shape/pattern: the white rectangles from the first image are also

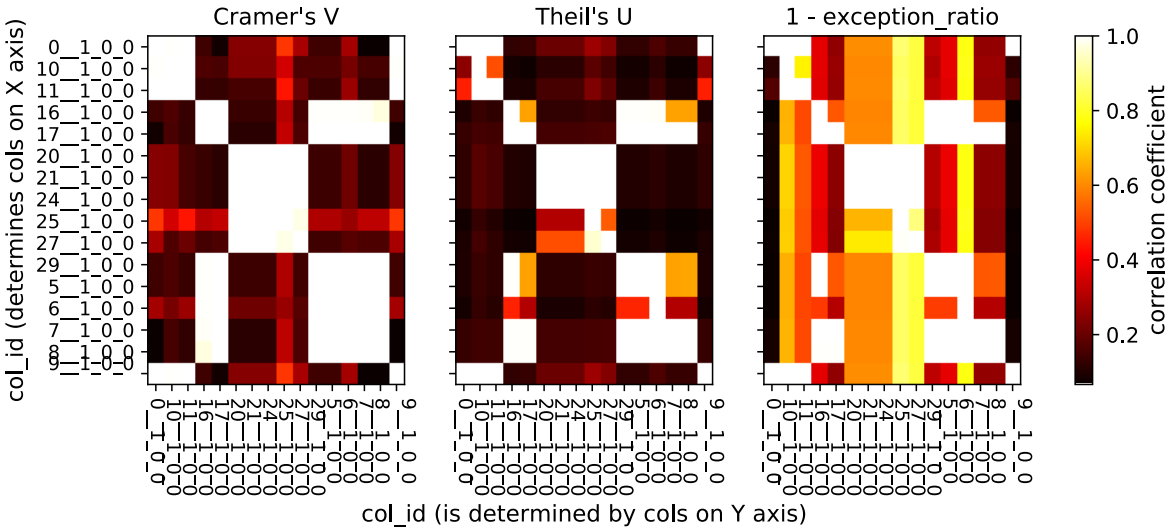


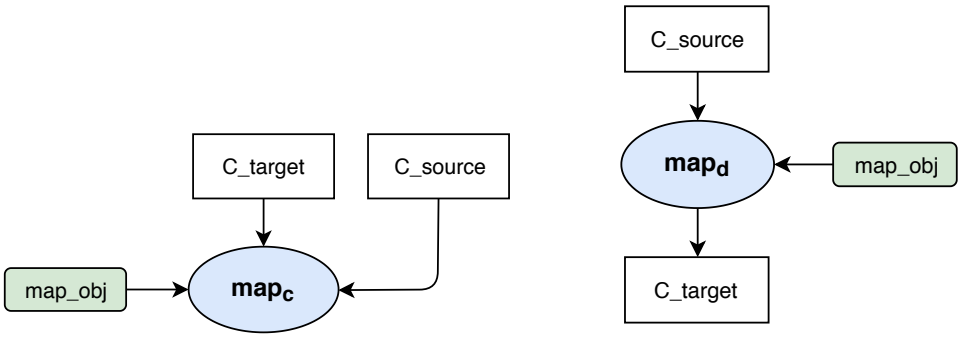
Figure 5.6: Correlation coefficients comparison (*YaleLanguages_1*)

visible in the other two and their internal shape is almost identical in the last two images. Our approach gives higher overall coefficients, but it is consistent with the other two approaches when it comes to identifying near-perfect correlations: all pairs of columns with correlation coefficient ≈ 1.0 are also identified by Cramer's V and Theil's U. Based on these results, we decided to use our own version of computing the correlation coefficient $(1 - exception\_ratio)$ instead of the other approaches for the following reason: for our purpose, correlation is only useful if we can represent columns as functions of other columns through correlation mappings. These mappings will determine the exceptions, which will, in turn, determine the size of the physical data and implicitly the compression ratio. Therefore, the correlation coefficient, as we defined it above, gives the most accurate estimation of how suitable two columns are for correlation representation. In this context, Cramer's V and Theil's U do not bring any advantage over our approach and even if we were to use them instead, we would still need to compute the correlation mapping. Moreover, we are only interested in highly correlated columns which are similarly identified by all approaches (more details about the selection of column pairs based on their correlation coefficient are discussed in 5.2.4.4 Correlation pattern selector and 6.2 Experimental setup).

Coming back to the pattern detection process, the histograms—necessary for creating the correlation mappings—are built in the *scanning* phase and the correlation coefficients and mappings are computed in the *evaluation* phase. The compression and decompression metadata resulted from this process is the correlation map map_{obj}. The pattern detector

outputs all (c_{source}, c_{target}) pairs with $corr\_coef > corr\_coef_{min}$, leaving the task of choosing between the results to the learning algorithm. The *evaluation result* is composed of the *coverage* and *row_mask*—as defined in 5.1.1 Generic pattern detector—and the correlation coefficient.

The *expression nodes* for the Column correlation pattern are illustrated in Figure 5.7.



(a) compression **(b)** decompression

Figure 5.7: Column correlation expression nodes

The *compression node* takes as input the target column (c_{target}), the source column (c_{source}) and the metadata (map_{obj}). It does not generate any output column. The *decompression node* takes as input c_{source} and map_{obj} and reconstructs c_{target} based on the mapping.

The compression operator map_c takes as input a target value (v_{target}), a source value (v_{source}) and the metadata (map_{obj}) and checks whether the key-value pair (v_{source}, v_{target}) is present in map_{obj}. If true, then it does nothing—v_{target} can be reconstructed based on v_{source}. Otherwise, it raises an *OperatorException* indicating that v_{target} cannot be retrieved from the mapping and needs to be stored in the exception column. The decompression operator map_d takes as input v_{source} and map_{obj} and returns the value from map_{obj} associated to v_{source}: $map_{obj}[v_{source}] = v_{target}$.

The benefit of the Column correlation compression scheme is clear: we avoid storing a column by representing it as a function of another column. However, this comes at the cost of storing the mapping between the 2 columns. Thus, it is only worth using this method when the mapping is small. The mapping size is dependent on the cardinality of the sets of values in the 2 columns. This is similar to Dictionary encoding and therefore we can state that Column correlation is effective only when both source and target columns are dictionary compressible. With this new constraint, we can limit the scope of the Column correlation pattern detector to output columns of *whitebox* Dictionary nodes, leading to the

following benefits: 1) reduced detection time—less column pairs that need to be checked; 2) reduced mapping size—dictionary ids instead of string values. Due to the generic nature of *whitebox* compression, this constraint can be implicitly satisfied by just adding a rule to the *select_column* method of the Column correlation pattern detector and letting the learning algorithm perform the recursive compression.

5.1.6 Dictionary

We implemented a *whitebox* version of Dictionary encoding to serve as a prior compression step before Column correlation. The pattern detector receives an additional parameter $size_{max}$—maximum size of the dictionary. We restricted its scope to `VARCHAR` columns. This is a single-column pattern detector and columns are evaluated independently. The next paragraphs describe the pattern detection process for a single column.

Dictionary encoding only produces good results on columns that have a small number of unique values. However, it is hard to reliably quantify this property when analyzing only a sample of the data. Moreover, the distribution of unique values may be skewed, with only a few values with high frequency and a long tail of low frequency values. For the purpose of our pattern detector, we addressed this issue by enforcing a maximum dictionary size (in bytes) and only keeping the most common values in the dictionary. This approach is also suitable if blocks of data are compressed independently: dictionaries need to be small as they are assigned per block. There are other (possibly better) ways of optimizing the dictionary values and size (e.g. choosing which—and how many—values to keep in the dictionary based on the resulting physical size of the sample: dictionary + compressed values + exceptions). However, this is not a core aspect for our pattern detector, since we only use it as an intermediate step required for Column correlation.

The dictionary is built as follows. The *scanning* phase creates the histogram of all the values in the sample. In the *evaluation* step we select as many values (v_s) from the histogram—in decreasing order of their number of occurrences—such that their total size is smaller or equal to the maximum size of the dictionary ($size_{max}$). The dictionary is represented as an array containing the selected values. The indices in the array represent the dictionary ids (v_{id}) used to encode the values (v_s). The dictionary represents the compression and decompression metadata. All the values that are not present in the dictionary are considered exceptions.

The pattern detector only outputs results for columns that are dictionary compressible. This decision is taken based on the estimated size of the input column, output columns

and metadata:

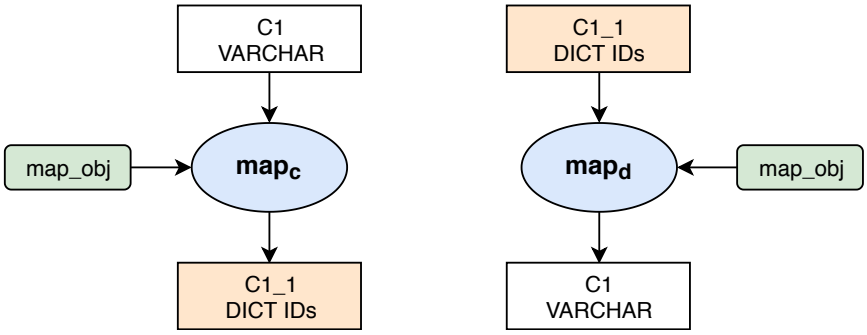
$$size_{in} > size_{out} + size_{ex} + size_{metadata} \tag{5.4}$$

where:

$size_{in}$ = estimated size of the input column
$size_{out}$ = estimated size of the dictionary ids column
$size_{ex}$ = estimated size of the exceptions column
$size_{metadata}$ = size of the dictionary

If the above condition is *True*, the input column is considered dictionary compressible and the pattern detector outputs a result for it. More details about how these sizes are computed are given in 5.2.2.3 Dictionary estimator. The *evaluation result* is composed of the *coverage* and *row_mask*—as defined in 5.1.1 Generic pattern detector.

The *expression nodes* for the Dictionary pattern are illustrated in Figure 5.8.



(a) compression **(b)** decompression

Figure 5.8: Dictionary expression nodes

The *compression node* takes as input the string column and the dictionary *map_obj*. It outputs a column containing dictionary ids. The *decompression node* takes as input the dictionary ids column and the dictionary and reconstructs the original input column.

The compression operator map_c takes as input the string value v_s and the dictionary *map_obj* and outputs a dictionary id: the index of v_s in the array *map_obj*. As an optimization for the compression phase, *map_obj* is represented as an actual dictionary with (v_s, $index\_of(v_s)$) key-value pairs instead of an array. If v_s is not found in *map_obj* an *OperatorException* is raised, indicating that v_s is an exception and should be stored in the exception column. The decompression operator map_d takes as input a dictionary id v_{id} (i.e. an index in the *map_obj* array) and returns the original value v_s ($map\_obj[v_{id}]$).

5.2 Compression learning process

We define *compression learning* as the process of learning the best representation of a dataset in terms of storage size. This section describes: 1) the general optimization problem of learning the best compression tree based on a sample of data; 2) an exhaustive learning algorithm and its cost model; 3) a greedy algorithm and the heuristics it uses for decision making; 4) an architecture for combining multiple learning algorithms.

5.2.1 Optimization problem

We can define the learning process as follows: given a sample from a dataset, its schema and a set of pattern detectors as input, output a compression tree that, when applied to the dataset, produces a compressed representation of it of minimum disk size.

The schema is a list of columns and their data types. The pattern detectors are implementations of the Generic pattern detector—receives the columns as input, evaluates the sample and returns a list of *(expression node, evaluation result)* tuples. Adding an *expression node* to the compression tree means: 1) altering the schema by deleting existing columns and creating new ones; 2) altering the sample by applying the *compression operator* to the input columns and generating new data. The learning process can go on by recursively feeding the new schema and sample data to the pattern detectors, resulting in new *(expression node, evaluation result)* tuples. This recursive process stops when no pattern detector outputs any result anymore—no pattern matches on the current schema and data.

Each decision of adding or not adding an *expression node* to the compression tree generates a new solution. This leads to a total number of 2^n possible solutions (different compression trees), where n is the total number of *expression nodes* generated by the recursive process (n binary decisions: 1 means adding a node and 0 not adding it). The total number of *expression nodes* (n) depends on how well pattern detectors match on the initial columns and the newly generated ones. This is entirely dependent on the characteristics of the data and the patterns that were evaluated on it. In the worst case, all pattern detectors will match on any column, leading to the following expression for n:

$$n = c_{in} \times (p \times avg(n_p) \times b)^h \tag{5.5}$$

$$b = avg(c_{out}) \tag{5.6}$$

where:

n = total number of expression nodes
c_{in} = number of input columns
p = number of pattern detectors
n_p = number of expression nodes returned by a pattern detector
b = branching factor of the compression tree
h = height of the compression tree
c_{out} = number of output columns of an expression node

The score of each solution is given by the size of the compressed data that resulted after applying the compression tree to the dataset. The goal of the learning process is to choose the one that gives the smallest size.

The computational effort needed for each individual *expression node* consists of: 1) applying the compression operator on its input columns to generate the new data (feeding each tuple in the sample data to the operator); 2) evaluating the pattern detectors on all the new columns (feeding each tuple in the sample data to each pattern detector). Moreover, some pattern detectors may need to evaluate combinations of columns instead of individual columns, which requires all the existing columns to be reevaluated for every newly generated column (e.g. Column correlation evaluates all pairs of 2 columns to determine the correlation coefficient between them).

5.2.2 Cost model: compression estimators

The compression learning process is an optimization problem (5.2.1) for finding the best compression tree in terms of disk size of the resulting physical columns. Solving the optimization problem requires a way to compare its solutions, i.e. estimating the final size of the compressed data. For this purpose we created a cost model which relies on leaf compression schemes size estimators (DICT, RLE, FOR, no compression) to predict the size of a column if compressed with these methods.

Methodology. The score of a solution (compression tree) is computed through the following methodology: 1) represent the sample data according to the compression nodes; 2) for each column of the new representation, estimate its size if it were compressed with: DICT, RLE, FOR or not compressed at all; 3) choose the smallest size among those. The result of this process is the smallest size of the new data representation if it were compressed with leaf compression schemes.

System assumptions. The size on disk of a compressed column depends on: 1) implementation of the compression method; 2) characteristics of the underlying database system. The former is described in the next sections as part of the estimator implementations. For the latter we defined some assumptions of the underlying system as follows:
a) *data types*: strings are stored with null terminator, therefore their size is given by the number of characters + 1. For all other data types we consider the sizes used by Ingres Vectorwise (10, 27).
b) *null handling*: we consider the same approach used by Vectorwise (10): do not store null values, instead, keep track of their positions using a bitmap. This results in 1 additional bit for every attribute (for nullable columns).
c) *exception handling*: we consider a whitebox approach: for each logical column store exceptions on a separate physical nullable column. The exception column has the same datatype as the logical column.

An additional assumption that we make about the underlying system is that it supports block-level compression, i.e. every block of data is compressed independently. This allows different compression schemes to be used on the same column, enabling the possibility to exploit local data characteristics.

5.2.2.1 Generic compression estimator

A compression estimator takes as input a column and two samples of data (*train* and *test* sample) and outputs the estimated size of the compressed column. The result can be either the size of the compressed sample or the size extrapolated to the total size of the block or column. The latter requires an additional parameter specifying the total number of rows in the full data.

The estimated size has 4 components (exemplified for Dictionary encoding):
1) $size_{metadata}$: size of the metadata (the dictionary itself)
2) $size_{data}$: size of the compressed data (the dictionary ids)
3) $size_{ex}$: size of the exceptions (the values that are not in the dictionary)
4) $size_{null}$: size required to keep track of the null values. Since exceptions are stored on a separate nullable column, the *nulls* size is implicitly increased.

The final estimated size of the test sample is the sum of the 4 components:

$$size_{sample} = size_{metadata} + size_{data} + size_{ex} + size_{null} \tag{5.7}$$

This result gives the size of the *test* sample only. It can be extrapolated to the full size of the block or entire column as follows:

$$size_{full} = size_{sample} \times \frac{count_{full}}{count_{sample}} \tag{5.8}$$

where:

$count_{sample}$ = total number of values in the test sample
$count_{full}$ = total number of values in the full block or column

The size estimation process works in two phases:
1) *training*: the estimator analyzes the *train* sample and generates the metadata needed for compression (e.g. Dictionary estimator generates the dictionary, Frame of Reference estimator determines the reference value and the number of bits needed to store the differences).
2) *testing*: the estimator simulates the compression of the *test* sample using the metadata resulted from the *training* phase and outputs an estimated size.

The two-phase estimation process is used to avoid overly-optimistic results: metadata generated based on the *train* sample is perfectly optimized for that sample (e.g. in FOR all differences will fit in the number of bits chosen to represent them). Depending on the implementation of each compression estimator, this would lead to a reduced number of exceptions or even no exceptions at all. Therefore, the *test* sample is used to provide new data for size estimation. It produces exceptions and more realistic results. This approach simulates the compression process of real database systems, where the compression metadata is created based on a sample and then applied on a full block of data or even on the entire column.

The next sections describes 4 compression estimators used in the learning process. All computed sizes will be in bytes.

5.2.2.2 No compression estimator

The No compression estimator predicts the size of the input column stored without compression. The *training* phase is not relevant since it does not generate any compression metadata. The estimation is performed in the *testing* phase, based on the size on disk of the column data type. The size components are computed as follows:

$size_{metadata}$ is 0, since there is no compression metadata

$size_{ex}$ is 0, since there are no exceptions

$size_{null}$ is 1 bit for every value in the sample:

$$size_{null} = \frac{count_{sample}}{8} \tag{5.9}$$

where:

$count_{sample}$ = total number of values in the test sample

$size_{data}$ is given by the total size of the non-null values in the sample. It depends on the data type of the column as follows:

$$size_{data} = \begin{cases} \sum_{v \neq null} len(v) + 1 & \text{if } datatype = \texttt{VARCHAR} \\ count_{notnull} \times size_{datatype} & \text{else} \end{cases} \tag{5.10}$$

where:

$count_{notnull}$ = total number of non-null values in the test sample
$size_{datatype}$ = size on disk of the column data type

5.2.2.3 Dictionary estimator

The Dictionary estimator predicts the size of the input column as compressed with Dictionary encoding. Besides the two samples, it receives an additional parameter: $size_{max}$— maximum size of the dictionary (in bytes). It only applies to `VARCHAR` columns and therefore the exception column will also be `VARCHAR`.

The Dictionary estimator is similar to the Dictionary pattern detector defined in 5.1.6. It builds the dictionary and handles exceptions in the same way. Optimizing the dictionary based on a maximum size (in bytes) also works for the estimator, since we use it to compare different ways of compressing a column and the dominant factor here is the nature of the data, not the optimization of the compression scheme.

Training phase. The dictionary (metadata) is built during the *training* in same way it is done for the Dictionary pattern detector (5.1.6): Step-1: create the histogram of all the values in the *train* sample. Step-2: select as many values from the histogram in decreasing order of their number of occurrences such that their total size is lower or equal to the maximum size of the dictionary ($size_{max}$). The dictionary is stored as an array containing the selected values. The indices in the array represent the dictionary ids used to encode the values.

$size_{metadata}$ is given by the total size of the values in the dictionary. Additionally, the number of bits required to store a dictionary id is computed as follows:

$$bits_{id} = \lceil \log_2(count_{entries}) \rceil \tag{5.11}$$

where:

$count_{entries}$ = number of values in the dictionary

Testing phase. The *testing* phase estimates the size of the compressed column by going through each value in the *test* sample and checking if it is present in the dictionary. The following variables are updated in this process: 1) $count_{valid}$: number of values that are found in the dictionary; 2) $size_{ex}$: size of the exceptions (values that are not found in the dictionary).

$size_{data}$ is computed as follows:

$$size_{data} = \frac{count_{valid} \times bits_{id}}{8} \tag{5.12}$$

$size_{ex}$ is computed by summing the size of all exceptions.

$size_{null}$ is determined by the number of resulting physical columns: one for compressed data (dictionary ids) and one for exceptions:

$$size_{null} = \frac{2 \times count_{sample}}{8} \tag{5.13}$$

where:

$count_{sample}$ = total number of values in the test sample

5.2.2.4 Run Length Encoding estimator

The Run Length Encoding estimator predicts the size of the input column as compressed with RLE. The samples are constructed with consecutive ranges of values such that RLE is triggered (see 6.1.1 Sampling for more details). Even though RLE can be applied to any data type, we limited the scope of our estimator to numeric columns. The other data types are either compressed with Dictionary encoding (`VARCHAR`) or are very rare in the Public BI benchmark and do not present compression opportunities. We use the following terminology:

1) *run* value = a data value that is repeated on consecutive rows.

2) *length* value = the number of consecutive occurrences of a *run* value

Training phase. RLE metadata is composed of: 1) the number of bits needed to represent the *run* values ($bits_{run}$) and 2) the number of bits needed to represent the *length* values ($bits_{length}$). These values are determined by scanning the *train* sample and computing all the *runs* and *lengths*. $bits_{run}$ is given by the *run* value of maximum size and $bits_{length}$ is given by the maximum *length*. $bits_{run}$ also depends on the column data type representation.

$size_{metadata}$ is between 8 and 24 bytes—the size of 2 numbers: $bits_{run}$ and $bits_{length}$—depending on the data types used to store them.

Testing phase. The *testing* phase scans all the values in the *test* sample and creates (*run*, *length*) pairs as follows: 1) if a *run* value cannot be represented on $bits_{run}$ bits: mark it as exception and skip it; 2) if a *length* exceeds the maximum value that can be represented on $bits_{length}$ bits: end the current run at this length and start a new run. The following variables are updated in this process: 1) $count_{valid}$: the number of (*run*, *length*) pairs resulted from the scanning process; 2) $count_{ex}$: the number of exceptions as defined above.

$size_{data}$ and $size_{ex}$ are computed as follows:

$$size_{data} = \frac{count_{valid} \times (bits_{run} + bits_{length})}{8} \tag{5.14}$$

$$size_{ex} = count_{ex} \times size_{datatype} \tag{5.15}$$

where:

$size_{datatype}$ = size on disk of the column data type

$size_{null}$ depends on the number of physical columns—one compressed data column (*run* and *length* are stored together) and one exception column—the same as in the case of Dictionary estimator (Equation 5.13).

5.2.2.5 Frame of Reference estimator

The Frame of Reference estimator predicts the size of the input column as compressed with FOR. It only applies to numeric columns.

Training phase. FOR metadata is composed of: 1) the *reference* value and 2) the number of bits needed to store the differences ($bits_{diff}$). In our implementation we chose the *reference* to be the smallest value in the *train* sample. $bits_{diff}$ is given by the maximum difference size, which depends on the data type representation.

$size_{metadata}$ is between 8 and 24 bytes—the size of the reference and the size of $bits_{diff}$—depending on the data types used to store them.

Testing phase. The testing phase computes all the differences between the values in the *test* sample and the *reference* and filters the ones that can be represented on $bits_{diff}$ bits. The following variables are updated in this process: 1) $count_{valid}$: the number of differences that fit in $bits_{diff}$; 2) $count_{ex}$: the number of exceptions (values that give differences larger than $bits_{diff}$).

$size_{data}$ is computed as follows:

$$size_{data} = \frac{count_{valid} \times bits_{diff}}{8} \tag{5.16}$$

$size_{ex}$ is the same as in the case of Run Length Encoding estimator (Equation 5.15).

$size_{null}$ depends on the number of physical columns—one compressed data column (differences) and one exception column—the same as in the case of Run Length Encoding estimator and Dictionary estimator (Equation 5.13).

5.2.3 Recursive exhaustive learning

This section presents an exhaustive recursive algorithm for compression learning. It uses the compression estimators (E) as a cost model (5.2.2 Cost model: compression estimators) and the pattern detectors (P) defined in 5.1 Pattern detection. The algorithm takes as input a column (c_{in}) and outputs the best compression tree (T_{out}) with respect to the compression estimators: the one that produces the smallest representation of the column when used to compress it. The algorithm is recursive and takes a depth-first approach. For this reason, it is not suitable for pattern detectors that work with more than one column (e.g. Column correlation).

In addition to the input column c_{in}, the recursive function receives a compression tree T_{in} (initially empty) and an optional max height parameter h_{max} (the maximum height of the compression tree). The compression tree T_{in} is the partial solution built in the recursive process so far. The role of the h_{max} parameter is to limit the dimension of the problem and avoid scenarios where the algorithm does not finish .

The algorithm tries all possibilities to compress the input column c_{in}: 1) with leaf compression nodes (N_{leaf}, e.g. RLE, FOR, DICT); 2) with internal (non-leaf) expression nodes ($N_{internal}$, e.g. Numeric strings, Character set split, etc.); 3) without compression. For each possibility it estimates the size of the resulting columns (c_{out}) in the following way: 1) for leaf expression nodes (N_{leaf}) and no compression, the estimators E directly provide

the size; 2) for non-leaf expression nodes ($N_{internal}$), the size is computed by recursively applying the same algorithm on the output columns c_{out} of the expression node $N_{internal}$ and adding the resulted sizes together. The recursive call returns a new compression tree T_c for each output column of $N_{internal}$. A special case is the DICT expression node, which will have 2 estimated sizes: 1) direct estimation from the Dictionary estimator E_{dict} as a leaf node; 2) recursive call estimation (other pattern detectors P can be applied on its output column, e.g. Column correlation).

Out of all the possibilities to compress c_{in}, the one which gives the smallest size is chosen and T_{in} is updated with either the leaf compression node N_{leaf} or with the set of compression trees T_c resulted from the recursive call. The algorithm returns the updated compression tree T_{out}.

The termination conditions of the recursive algorithm are: 1) all possibilities to compress c_{in} are leaf compression nodes (i.e. there is no $N_{internal}$ to generate new output columns c_{out}); 2) the max height of the compression tree h_{max} is reached. Termination condition 1) occurs when no pattern detector P outputs any compression nodes $N_{internal}$. A pattern detector P does not output any compression node $N_{internal}$ when: 1) c_{in} is not compatible with the pattern (e.g. Numeric strings pattern detector only applies to `VARCHAR` columns; see *select_column* in 5.1 Pattern detection); 2) the data in c_{in} does not match the pattern (e.g. a column with multiple values does not match the Constant pattern).

The algorithm is described in listings 5.2 and 5.3.

Listing 5.1: Naming conventions

```
c = column
T = compression tree
E = compression estimator
P = pattern detector
N = compression node
```

Listing 5.2: build_T (recursive exhaustive)

```
def build_T(c_in, T_in, P_list, E_list):
  sol_list = list()
  # estimator evaluation
  for E in E_list:
    size = E.evaluate(c_in)
    sol_list.append((size, T_in))
  # pattern detection
  N_list = list()
  for P in P_list:
```

```
    N_p_list = P.evaluate(c_in)
    N_list.extend(N_p_list)
  # recursive evaluation
  for N in N_list:
    (size, T_out) = apply_N(c_in, N, T_in, P_list, E_list)
    sol_list.append((size, T_out))
  # return best solution
  return min(sol_list, key=size)
```

Listing 5.3: apply_N (recursive exhaustive)

```
def apply_N(c_in, N, T_in, P_list, E_list):
  T_out = copy(T_in)
  T_out.update(N)
  # recursive call for all output columns
  sol_list = list()
  for c_out in N.c_out_list:
    (size_c, T_c) = build_T(c_out, T_out, P_list, E_list)
    sol_list.append((size_c, T_c))
  # merge results
  size_out = 0
  for (size_c, T_c) in sol_list:
    size_out += size_c
    T_out = merge(T_out, T_c)
  # return merged result
  return (size_out, T_out)
```

The complexity of the algorithm can be described as:

$$\mathcal{O}((p \times avg(n_p) \times b)^{h_{max}}) \tag{5.17}$$

where:

p = number of pattern detectors

n_p = number of expression nodes returned by a pattern detector

b = branching factor of the compression tree (as defined in Equation 5.6)

h_{max} = maximum height of the compression tree

The height is bounded by the h_{max} parameter. Despite the high complexity, the size of the problem remains relatively small due to the high selectivity of the pattern detectors P (see *select_column* in 5.1 Pattern detection). This is the complexity of learning the

compression tree for a single column, while for multiple columns the complexity becomes:

$$\mathcal{O}(c_{in} \times (p \times avg(n_p) \times b)^{h_{max}}) \tag{5.18}$$

where:

c_{in} = number of input columns

This complexity is significantly better than the complexity of the general optimization problem described in 5.2.1 ($\mathcal{O}(2^n)$, where n is the total number of expression nodes generated in the learning process, as described in Equation 5.5). The reason for this improvement is the single-column approach—not considering pattern detectors P that combine multiple columns. The solutions of a column c_i do not depend on the solutions of other columns c_j that are not on the path from c_i to its root input column c_r. This is because a choice made for c_j does not influence the choices that can be made for c_i. This property allows a depth-first exploration of the solutions, significantly reducing the complexity.

However, there is a cost for not considering multi-column pattern detectors: the algorithm misses compression opportunities (e.g. one column represented as a function of another through Column correlation). It is exhaustive and yet cannot produce the best result. Section 5.2.5 Iterative greedy learning presents a greedy approach that considers multi-column patter detectors. Moreover, the Multi-stage learning in section 5.2.6 addresses this issue by chaining together multiple learning algorithms.

5.2.4 Pattern selectors

Section 5.2.3 described an exhaustive learning algorithm based on compression estimators (5.2.2). A greedy learning algorithm based on *pattern selectors* will be described in section 5.2.5. Pattern selectors are decision algorithms used to make greedy choices in the compression learning process. This section describes the general characteristics of a pattern selector and specialized instances of it.

5.2.4.1 Generic pattern selector

A generic pattern selector is an abstraction used in the learning process for selecting the *expression nodes* that will be added to the compression tree. Given a list of *(expression node, evaluation result)* tuples resulted in the pattern detection phase, it outputs a subset of the expression trees provided as input. This selection process is necessary as there are multiple ways of representing the same column. The generic pattern selector chooses the

best *expression nodes* according to a set of criteria. Different implementations of pattern selectors are described below.

5.2.4.2 Coverage pattern selector

The coverage pattern selector tries to maximize the coverage of each column (i.e. minimize the exception ratio). It has 2 operation modes: 1) *single-pattern*: selects the *expression node* with the highest coverage; 2) *multi-pattern*: selects the best combination of *expression nodes* that give the largest coverage when used together on the same column.

The values on a column usually fall in different, possibly overlapping, pattern types. It is rarely the case that one pattern perfectly fits to all the values in a column. In most cases there is either one dominant pattern and the rest of the values are just noise, or multiple patterns that together cover all the values on the column. An example could be a `VARCHAR` column where half of the values are numbers and the other half are concatenations of a constant with numbers.

The *single-pattern* operation mode will select only one expression node—the one with the highest coverage. The *multi-pattern* operation mode will select multiple expression nodes, such that their combined coverage of the column is maximal. We defined 2 approaches for the *multi-pattern* mode: 1) a Greedy algorithm that selects one expression node at a time—the one with the highest coverage—until the column is fully covered or there are no more expression nodes to choose from; 2) an exhaustive algorithm that tries every combination of expression nodes and outputs the one with the highest cumulative coverage. Both approaches can further take into account the following parameter: $coverage_{min}$—minimum coverage that determines whether a pattern is considered or not.

The combined coverage of 2 patterns p_a and p_b can be computed based on their *row_ masks* rm_a, rm_b—defined in 5.1.1—through a bitwise `OR` operation: $rm_{combined} = rm_a | rm_b$. The number of bits of a *row_ mask* is equal to the number of rows in the table: r. The combined coverage is then computed as the number of 1 bits in $rm_{combined}$ divided by r.

The complexity of these algorithms depends on the number of patterns found on the column: p. The *single-pattern* operation mode and the Greedy algorithm for the *multi-pattern* mode both have a linear complexity. The exhaustive algorithm has an exponential complexity, as it tries all combinations of patterns: for $p = 10$ there are 10^3 bitwise `OR` operations on r-bit numbers, while for $p = 27$ this number grows to 10^9. Equation 5.19

shows the complexity of the exhaustive algorithm.

$$\mathcal{O}(\sum_{k=1}^{p} \binom{p}{k}) \tag{5.19}$$

We implemented a basic version of the exhaustive approach for the *multi-pattern* mode. To avoid the exponential running time, we default to the *single-pattern* mode if the number of patterns p is larger than 20. In practice, we never encountered more than 20 different patterns on the same column, therefore, the exhaustive approach is a suitable choice.

Both operation modes give similar results in terms of physical representation of the data, but the resulting expression trees have different shapes. The *single-pattern* mode selects only one expression node, moving the other values on an exception column. If the learning algorithm supports recursive expression of exception columns, then the most dominant pattern in the exception column is further selected, resulting in a new expression node on a new level of the tree. This process creates deep expression trees. In contrast, the *multi-pattern* mode adds all expression nodes on the same level of the tree, resulting in wider but shorter expression trees. The evaluation of expression trees resulted from *multi-pattern* selection is described in 4.4 Compression and decompression.

5.2.4.3 Priority pattern selector

The purpose of this pattern selector is to choose the best expression node based on pattern priorities. In addition to the *expression node* list, it also receives a set of priority classes for pattern detector types. Each priority class contains a list of pattern types (e.g. Numeric strings, Character set split, etc.) and a pattern selector that will be used to select the patterns in the same class. The selection process works in the following way:
1) for each column, select only the *expression nodes* with the highest priority—their associated pattern type has the highest priority according to the provided list;
2) further select these *expression nodes* using the pattern selector provided for their priority class and output the result.

An example of an input priority set is shown in Table 5.5.

Priority	Pattern type	Pattern selector
1	Constant	N/A
2	Dictionary, Numeric string	Coverage selector
3	Character set split	Coverage selector

Table 5.5: Priority set example

Given a column c and the priority set in Table 5.5, the Priority pattern selector will proceed as follows. It will first search for a Constant expression node that has c as input. If found, c will be represented as a constant. Since the Constant pattern detector outputs only one result per column, there is no need for a pattern selector on the constant priority class—the only constant expression node will be chosen. If there is no constant expression node, the pattern selector will search for expression nodes in the next priority class: Dictionary and Numeric strings. Both pattern detectors output a single result per column, thus there is a maximum of 2 expression nodes to choose from. The Coverage pattern selector will be used to choose between them. Finally, if no expression node was found yet, the selector moves to the last priority class. It searches for Character set split expression nodes. There can be multiple instances of the Character set split pattern detector initialized with different parameters and each instance can output multiple results for the same column. The Coverage pattern selector is used to choose the best combination of results such that the total coverage of the column is maximized.

5.2.4.4 Correlation pattern selector

This pattern selector is specialized in Column correlation *expression nodes*. The Column correlation pattern detector outputs all the correlations between the columns in the dataset, resulting in multiple possibilities of representing the same column as a function of other columns—multiple *(source, target)* pairs with the same target column. The Correlation pattern selector selects the *expression nodes* such that every *target* column is represented by a single *source* column, while also trying to maximize the average correlation coefficient and avoid circular dependencies.

We can make the following observation: the column correlation as defined in 5.1.5 is a transitive relation. We formalized this observation in Theorem 1.

Theorem 1. *Let c_a, c_b, c_c be three columns and let $c_a \xrightarrow{m_{ab}} c_b$ be the correlation relation meaning: c_a determines c_b through a mapping m_{ab}. If $c_a \xrightarrow{m_{ab}} c_b$ and $c_b \xrightarrow{m_{bc}} c_c$ then $c_a \xrightarrow{m_{ac}} c_c$.*

Proof. Let (v_a, v_b) be an entry in m_{ab}—meaning: value v_a on column c_a always corresponds to value v_b on column c_b—and let (v_b, v_c) be an entry in m_{bc}. Then, value v_a on column c_a always corresponds to value v_c on column c_c. Therefore, $\exists\ m_{ac}$—a mapping containing the entry (v_a, v_c)—and $c_a \xrightarrow{m_{ac}} c_c$. □

We define the correlation graph as follows: a directed graph with one or more connected components; nodes represent columns; *(src, dst)* edges represent correlations: column *dst*

is determined by column *src*. The weight of an edge is the correlation coefficient between *src* and *dst*. An example of a correlation graph is shown in Figure 5.9. The red edges represent an optimal selection in terms of the metrics and conditions described in the following paragraphs. This figure shows a simple correlation graph, but in practice we also encountered more complex graphs resulting from the learning process for tables with highly correlated data (see Appendix B).

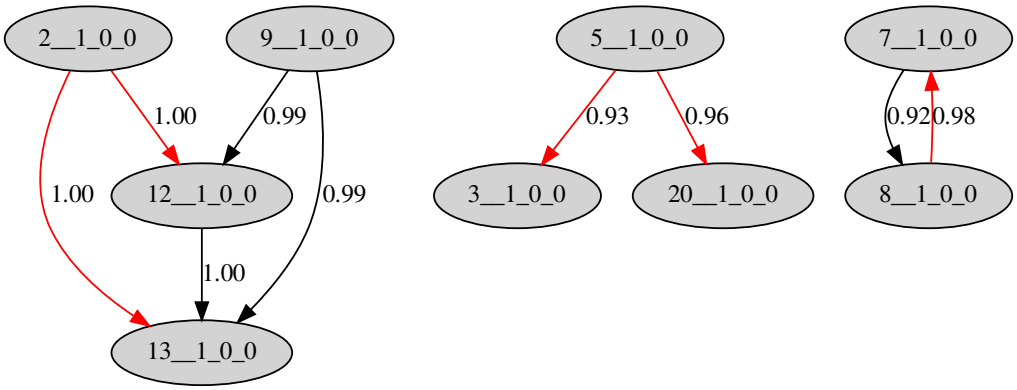


Figure 5.9: Correlation graph

The main goal of the Correlation pattern selector is to select a subset of the edges in the graph. We can define this process as the optimization problem of selecting a subset of edges in the correlation graph G such that the resulting subgraph G_s satisfies the following properties (in this order):

P1: the indegree of any node is at most 1—a column should be determined by no more than 1 other column

P2: any path in G_s is of length 1—because of the transitivity property, for every path between nodes c_a and c_b, there will also be a direct edge from c_a to c_b

P3: the number of nodes in G_s with indegree > 0 is maximal—the number of columns that are represented as functions of other columns is maximal

P4: the average weight of the edges is maximal—the edges with the highest correlation coefficient should be selected

We defined a Greedy algorithm for solving the correlation selection optimization problem. It builds the G_s graph by greedily selecting one edge at a time from G. The edge is added to G_s, G is updated and the process goes on until there are no more edges in G. The algorithm is described in listings 5.4, 5.5, 5.6.

Listing 5.4: select_correlations

```
# G = correlation graph resulted from the correlation detection process
def select_correlations(G):
  G_s = <empty correlation graph>
  while G.edges is not empty:
    # select destination node
    dst = min(G.nodes, key=get_node_score)
    # select (_, dst) edge
    (src, dst, corr_coef) = min(dst.incoming_edges, key=get_edge_score)
    # update G_s
    G_s.add((src, dst, corr_coef))
    # update G
    G.remove(dst.incoming_edges)
    G.remove(dst.outgoing_edges)
    G.remove(dst)
    G.remove(src.incoming_edges)
  return G_s
```

Listing 5.5: get_node_score

```
def get_node_score(node):
  indegree_src = min([src.indegree
                      for (src, dst) in node.incoming_edges])
  return (node.outdegree, indegree_src, node.indegree)
```

Listing 5.6: get_edge_score

```
def get_edge_score((src, dst, corr_coef)):
  return (src.indegree, -corr_coef, -src.out_degree)
```

Every iteration of the *while* loop first selects the destination node of the candidate edge based on its score. The *get_node_score* function prioritizes nodes with the minimum outdegree—ideally 0—such that the number of discarded edges as an effect of constraint *P2* is minimized. The difference between nodes with the same outdegree is made by the minimum indegree of its parent nodes (source nodes of incoming edges)—for the same reason. Finally the difference is further made by the smallest indegree of the node itself, prioritizing columns with fewer options of being represented as functions of other columns.

With the destination node fixed, the best edge is selected from its incoming edges. The *get_edge_score* function prioritizes edges that have a minimum indegree—ideally 0—such that nodes at the start of correlation paths are selected. The difference between these nodes is further made based on the highest correlation coefficient and then by the highest outdegree.

The selected edge is added to G_s and G is updated such that constraints *P1* and *P2* are satisfied: the destination node, its incoming and outgoing edges and the incoming edges of the source node are removed. This process changes the degree of the nodes in G and impacts their scores.

The complexity of the algorithm, as described above, is given by the number of nodes in G and their average indegree: $\mathcal{O}(N^2 \times avg(indegree))$, where N is the number of nodes. Every iteration of the *while* loop selects one node from G by computing the score of all nodes. The *get_node_score* function loops through all incoming edges of the node. If the selection of the *dst* node is done with a priority queue the complexity is improved: $\mathcal{O}(N \times \log(N) \times avg(indegree))$.

An alternative approach would be to relax constraint *P2* and allow paths of length higher than 1. Then apply a variation of the Dijkstra's shortest paths algorithm to shorten these paths with the goal of minimizing the complexity of the graph without affecting its average correlation coefficient. However, relaxing *P2* imposes a new constraint on G_s: it should not contain any cycles. We leave this approach for future work.

5.2.5 Iterative greedy learning

This section presents a greedy iterative algorithm for compression learning. It uses a pattern selector (S) to greedily choose how to compress columns (5.2.4 Pattern selectors) and the pattern detectors (P) defined in 5.1 Pattern detection. The algorithm takes a breadth-first approach, thus supporting pattern detectors P that work on multiple columns (e.g. Column correlation). It takes as input the list of input columns (c_{in}) and builds the compression tree one level at a time.

The algorithm builds the compression tree T in an iterative process. Let *add_level(T)* be an iterated function which adds 0 or more expression nodes (N) on a new level in T. This function is repeatedly applied to T in an iterative process. Each call of the function uses the updated version of T that it produced in the previous iteration. The iterative process converges to the final compression tree T_{out} once no more changes are made to T in an iteration.

The learning algorithm starts with P_{list} (a list of pattern detector instances), S (a pattern selector) and T (an empty compression tree, initialized with the list of input columns c_{in}). Recall that the compression tree is actually a graph with multiple connected components where each component is a directed acyclic graph (DAG) with one or more root nodes (see 4 Compression model). An additional parameter is it_{max}, representing the maximum

number of iterations. it_{max} also determines the maximum height of the compression tree (h_{max}), since every iteration adds a new level to T.

The *add_level* function adds new compression nodes (N) to T by trying to further compress its leaf (output) columns c_{leaf}. It does this in 2 steps. Step-1 pattern detection: use the pattern detectors P_{list} to generate all possibilities to compress the leaf columns, resulting in a list of expression nodes N_{list}. Step-2 pattern selection: use the pattern selector S to select a subset of N_{list} (N_{slist}), representing the best way to compress each c_{leaf}. Finally, the expression nodes in N_{slist} are added to as a new level. If no expression nodes (N) are selected (i.e. N_{slist} is empty), then the iterative learning algorithm converged to the final compression tree T.

The algorithm is described in listings 5.8 and 5.9.

Listing 5.7: Naming conventions

```
c = column
P = pattern detector
S = pattern selector
T = compression tree
N = compression node
```

Listing 5.8: build_T (iterative greedy)

```
def build_T(c_in_list, P_list, S)
  # initialize T with the input columns
  T = CompressionTree(c_in_list)
  # iterative process
  for it in range(0, it_max):
    add_level(T, P_list, S)
    if <no changes made to T>:
      break
  # return final compression tree
  return T
```

Listing 5.9: add_level (iterative greedy)

```
def add_level(T, P_list, S):
  c_leaf_list = T.c_leaf_list
  # pattern detection
  N_list = list()
  for P in P_list:
    N_p_list = P.evaluate(c_leaf_list)
    N_list.extend(N_p_list)
  # select expression nodes
```

```
N_s_list = S.select(N_list)
# stop if no expression nodes were selected
if len(N_s_list) == 0:
  return
# add selected expression nodes to T
T.update(N_s_list)
```

The complexity of the algorithm is given by the total number of columns in the final compression tree multiplied by the effort spent on evaluating the pattern detectors on it and choosing the expression node that it will be compressed with:

$$\mathcal{O}(c_{in} \times (b)^{h_{max}} \times (p + p \times avg(n_p))) \tag{5.20}$$

where:

c_{in} = number of input columns
b = branching factor of the compression tree (as defined in Equation 5.6)
h_{max} = maximum height of the compression tree
p = number of pattern detectors
n_p = number of expression nodes returned by a pattern detector

In the worst case when the maximum number of iterations is reached, h_{max} becomes it_{max}, thus the complexity is bounded by it_{max}.

Regardless of the nature of the pattern selector S, the algorithm is greedy, as it makes irreversible locally best choices which only depend on previously made choices. The breadth-first approach allows the use of multi-column pattern detectors, representing an advantage compared to the Recursive exhaustive learning algorithm, which misses compression opportunities by handling each column separately. The greedy model significantly reduces the number of solutions explored, leading only to a local best solution. At the same time, the reduced dimension of the explored solution set ensures faster termination of the algorithm. The advantages of both algorithms can be combined with the Multi-stage learning approach described in section 5.2.6.

5.2.6 Multi-stage learning

We observed that each learning algorithm has advantages and disadvantages and works better with different types of pattern detectors. E.g. the Recursive exhaustive learning algorithm has a wide coverage of the solution space of the problem, but does not support

multi-column pattern detectors. In contrast, the Iterative greedy learning algorithm does not miss multi-column compression opportunities and converges faster, but has a poor coverage of the solution space. Moreover, in the case of Iterative greedy learning, each pattern selector (S) is suitable for certain pattern detectors.

These observations lead to the conclusion that there is no *one size fits all* learning algorithm and call for a combined approach that leverages the advantages of each of them. Therefore, we defined the Multi-stage learning approach, which improves the learning process by chaining different algorithm instances together, each running in a separate stage.

An algorithm instance $A(args)$ is an application of algorithm A on the input parameters $args$. E.g. $IG(P_1)$ and $IG(P_2)$ will produce different compression trees T_1 and T_2 even though they are both instances of the Recursive exhaustive learning algorithm, because the pattern detector lists that they use P_1 and P_2 differ. This abstraction allows us to run the same learning algorithm with different pattern detectors or selectors and combine their results.

Chaining compression learning algorithm instances A_1 and A_2 means executing A_1 to get the compression tree T_1, then executing A_2 to build T_2 based on T_1. Chaining A_1 and A_2 requires that the output of A_1 can be converted to the input of A_2. This condition can be satisfied as all compression learning algorithms we proposed have compatible input and output parameters. The input of a learning algorithm is either one or more input columns or a compression tree. The output of all the algorithms is a compression tree. If A_2 requires a compression tree as input, then T_1 can be passed directly to it. If A_2 requires a list of columns, then the leaf columns of T_1 are passed to A_2. In the second case, the output of A_2 (T_2) needs to be merged with T_1 to form the final compression tree T_{out}. Additional input parameters (pattern detectors (P), pattern selectors (S), estimators (E)) are independent of the result of the previous algorithm.

This generic multi-stage design allows easy experimentation with any combination of compression learning algorithm instances. One example of chaining 2 learning algorithms is the following: first execute an instance of Recursive exhaustive learning with the single-column pattern detectors. Then execute an instance of Iterative greedy learning with the multi-column pattern detector Column correlation and the Correlation pattern selector. This will result in building the best compression tree using single-column pattern detectors and then identifying and exploiting correlations between the leaf columns of the tree, irrespective of their type (exceptions or not). While this setup misses correlation opportunities between intermediate/internal columns, it led to complex correlation graphs that

considerably reduce the number of physical columns—we present these results in 6.3.4 and Appendix B.

6

Evaluation and results

6.1 Methodology

The focus of this thesis is reducing the size of the data through *whitebox compression* as a data representation model. Therefore, our main evaluation metric will be the *compression ratio*. In terms of benchmarks, we evaluate our system on the Public BI benchmark (Chapter 3), as we target real, user-generated data rather than synthetic data. The datasets in the benchmark are available as CSV files.

We compare our implementation against two baselines: 1) a real system with enhanced compression capabilities: VectorWise (10); 2) a compression estimator model (defined in 6.1.3). We aim at showing the improvement brought by *whitebox compression* as an enhancement of existing systems: an intermediate data representation layer that remodels the data to create better compression opportunities for existing lightweight methods. For this reason we evaluate *whitebox compression* as an intermediate layer preceding these methods, instead of a stand-alone system. The following sections present the sampling method that we used for the automatic learning process and the methodology for the two baselines.

6.1.1 Sampling

The automatic learning process is based on samples. Various characteristics of the data are extracted by analyzing a small subset of rows. Real data may exhibit clustering properties (28), not only when it comes to sequences of numbers but also in terms of pattern locality. Moreover, there are compression techniques that apply to sequences of consecutive numbers (e.g. RLE). To capture these patterns and correctly evaluate the potential for different compression methods, basic single-point sampling is not enough.

To satisfy these requirements we defined a sampling technique that selects blocks of consecutive rows from random uniform points in the dataset. The input parameters are $count_{total}$—number of rows in the dataset, $count_{sample}$—number of rows in the sample, $count_{block}$—number of consecutive rows in a block. We select $\frac{count_{sample}}{count_{block}}$ random uniform positions in the dataset. From each position we select $count_{block}$ consecutive rows and add them to the sample. The uniform random selection and the blocks of consecutive rows ensure that we cover the full dataset and capture different local patterns, including sequences of values.

An additional requirement imposed by some pattern detectors is that the sample data fits in memory. To ensure this, our sampling tool can also receive a maximum size in bytes, from which it derives the $count_{sample}$ and $count_{block}$ parameters. This calculation is based on estimating the average size of a row.

6.1.2 VectorWise baseline

We chose VectorWise as a baseline for its enhanced compression capabilities. It uses *patched* versions of well known lightweight compression methods: PDICT, PFOR, PFOR-DELTA (1). *Patching* is an efficient way of implementing compression for skewed distributions, by storing outliers in uncompressed format in order to keep the size of the compressed non-outliers small. VectorWise's compression engine is similar to ours to some extent: the decision upon which compression method to use is based on analyzing a sample of the data. VectorWise handles each column separately and has the capability of independently compressing blocks of data with different methods instead of the full column.

Our evaluation methodology is based on comparing the size on disk of: 1) uncompressed data ($size_{uncompressed}$), 2) blackbox-compressed data with VectorWise ($size_{blackbox}$) and 3) whitebox-compressed data further compressed with VectorWise ($size_{whitebox}$). From these sizes we can derive the compression ratios:

$$ratio_{blackbox} = \frac{size_{uncompressed}}{size_{blackbox}} \qquad ratio_{whitebox} = \frac{size_{uncompressed}}{size_{whitebox}} \tag{6.1}$$

The evaluation architecture is depicted in Figure 6.1. $size_{uncompressed}$ is computed by loading the input data into VectorWise with compression disabled. $size_{blackbox}$ is the size of the data loaded into VectorWise with default compression options (lightweight compression only, LZ4 disabled). $size_{whitebox}$ is computed by: 1) feeding the input data to the *whitebox compression* engine, resulting in a new representation of it, then 2) loading the new data into VectorWise with the same default compression options. The *whitebox compression*

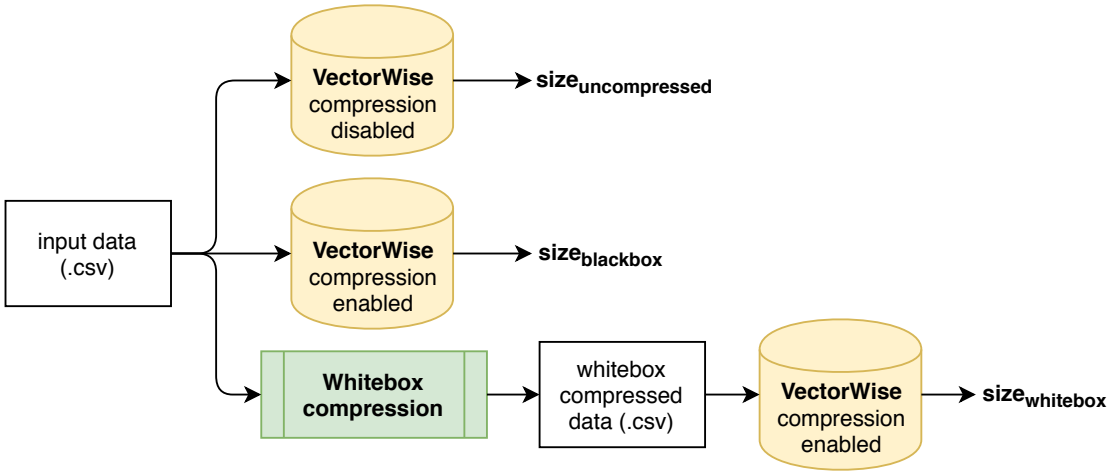


Figure 6.1: VectorWise evaluation methodology

engine learns a compression tree based on a sample and then evaluates it on the input data to materialize the new representation. The additional compression step with VectorWise applies existing blackbox compression schemes on the new representation of the data. The purpose of this 2-step compression process is to isolate and measure the capacity of *whitebox compression* to create better opportunities for existing lightweight compression schemes.

We determine the size of each table by inspecting the raw data files resulted after the bulk loading process. When computing $size_{uncompressed}$ and $size_{blackbox}$ we consider the total size of the data files associated with each table, which include the compressed values, exceptions and metadata. For $size_{whitebox}$ we get the size of all physical columns (including the exception columns) from VectorWise's data files and estimate the size of the metadata in the same way we do in 5.2.2 Cost model: compression estimators.

In most of the cases VectorWise stores each column in a separate file, allowing us to perform fine grained analysis on the size of individual columns. We noticed, however, that in some cases VectorWise stores multiple columns in the same file. Matching the column names with the data files is not a straight-forward process. The file name has a fixed format which includes the table and column names. However, some characters (mostly non-alphanumeric) tend to be replaced with other (combinations) of characters. We empirically determined some rules for this replacement (e.g. hexadecimal ASCII codes as replacements), but there are also exceptions. Therefore, we implemented a *column-to-file* approximate string matching algorithm based on the Levenshtein distance (29) and fuzzy regex matching (30)—which is out of the scope of this thesis and will not be discussed here.

Individual tables from the benchmark are loaded into VectorWise using the bulk loading utility `vwload` (31). We use the `trace point qe82` option to enable/disable compression as indicated in the performance guide (32). The size of each data file is retrieved from the Linux `stat` structure (33).

We found 2 additional information sources that might be useful for data analysis: 1) the `statdump` command (34)—general stats about tables (e.g. histogram) and 2) the compression logs—information from the compression process (e.g. which compression schemes were used). Even though the latter can be useful to understand how VectorWise compresses the data, the logs do not contain any information about which column and which block they refer to, resulting in a mix of information with missing context. A possible workaround would be to load each column as a separate table, reducing the unknown variables to the block number. This approach is feasible and might lead to interesting comparisons, however we did not proceed with it and we leave it for future work.

6.1.3 Estimator baseline

We created an alternative evaluation baseline in addition to the VectorWise one: a model which estimates the size of data based on the compression estimators defined in 5.2.2 Cost model: compression estimators. Its purpose is to simulate a stand-alone *whitebox compression* system, with whitebox versions of existing lightweight compression schemes as leaf nodes in the expression tree. The workflow is similar to the previous one: we compute the 3 sizes from which we derive the compression ratios. The only difference is that VectorWise is replaced by the estimator model, which estimates the size of each column as if it were compressed with whitebox implementations of existing lightweight compression schemes. The final size is the smallest size amongst the ones given by the 4 estimators. The methodology is depicted in Figure 6.2.

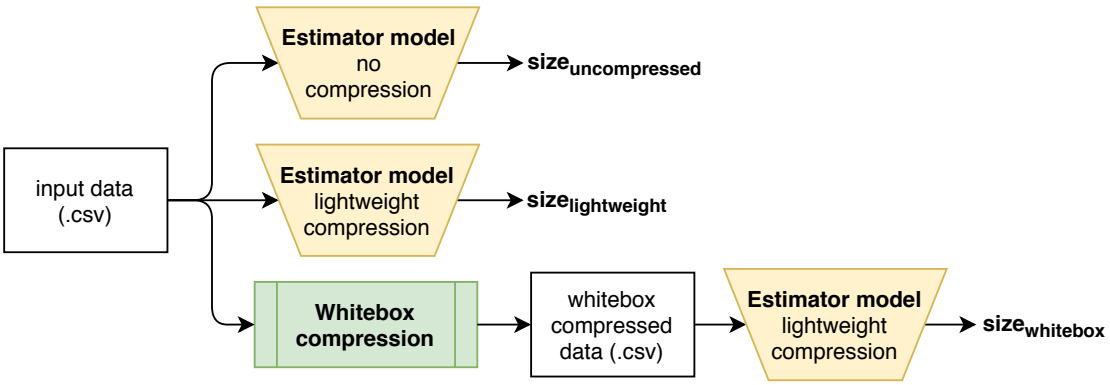


Figure 6.2: Estimator evaluation methodology

$size_{uncompressed}$ is computed by estimating the uncompressed size of the input data with the No compression estimator. $size_{lightweight}$ is the smallest size given by the compression estimators: Dictionary estimator, Run Length Encoding estimator, Frame of Reference estimator, No compression estimator. $size_{whitebox}$ is computed by first representing the input data through *whitebox compression* and then feeding it to the estimator model.

6.2 Experimental setup

We ran our experiments on a selection of 53 tables from the Public BI benchmark (4). The selection is made based on the observation that tables with the same schema have very similar data and sometimes are almost identical. Therefore, we only kept one table for each unique schema. We further removed a few more tables containing columns that could not be matched with the VectorWise data files because multiple columns are stored in the same file.

We configured our sampling tool to take samples of maximum 10MB with sequences of 64 consecutive rows. This resulted in samples of 20K rows on average and not less than 4K rows.

In terms of the learning algorithm, we experimented with both Iterative greedy learning (5.2.5) and Recursive exhaustive learning (5.2.3). We used different configurations and we present two of them—the ones that gave the best results:

Configuration A (iterative greedy): Multi-stage learning (5.2.6) with 2 stages:

Stage-1 Iterative greedy learning (5.2.5) with $it_{max} = 16$, 4 pattern detector instances—Numeric strings (5.1.3), Character set split (5.1.4), Constant (5.1.2) and Dictionary (5.1.6) —and the Priority pattern selector (5.2.4.3).

Stage-2 One iteration of Iterative greedy learning (5.2.5) with only one pattern detector—Column correlation (5.1.5)—and the Correlation pattern selector (5.2.4.4)

Configuration B (recursive exhaustive): Multi-stage learning (5.2.6) with 2 stages:

Stage-1 Recursive exhaustive learning (5.2.3) with $h_{max} = 5$, 4 pattern detector instances—Numeric strings (5.1.3), Character set split (5.1.4), Constant (5.1.2) and Dictionary (5.1.6) —and the compression estimators (5.2.2) as a cost model.

Stage-2 One iteration of Iterative greedy learning (5.2.5) with only one pattern detector—Column correlation (5.1.5)—and the Correlation pattern selector (5.2.4.4)

The parameters for the pattern detectors are the following:

Character set split: 2 charsets: *digits* and *non-digits*—isolates numbers in strings

Column correlation: $corr\_coef_{min} = 0.9$
Constant: $constant\_ratio_{min} = 0.9$—columns that are below are handled with Dictionary
Dictionary: $size_{max} = 64K$

The Priority pattern selector is configured with the following priorities: 1–Constant, 2–Dictionary, 3–Numeric strings, 4–Character set split. It selects between expression nodes in the same priority class with the Coverage pattern selector—configured in *multi-pattern* mode with $coverage_{min} = 0.2$. The Constant pattern detector has the highest priority since it is the most optimal way of compressing a constant column. The Dictionary expression nodes create opportunities for the Column correlation step in *Stage-2*. The Numeric strings pattern detector catches all string columns that contain numbers and the resulting format columns are handled in the next iteration with Constant and Dictionary. The remaining columns are left for the Character set split pattern detector which creates new compression opportunities for the other pattern detectors in the next iteration.

The compression estimators do not require additional parameters, excepting the Dictionary estimator, which uses the same maximum dictionary size as the Dictionary pattern detector: $size_{max} = 64K$.

6.3 Results and discussion

This section is structured as follows: we present and analyse the results obtained with Configuration A (iterative greedy) in 6.3.1, 6.3.2 and 6.3.3 and the results obtained with Configuration B (recursive exhaustive)—plus comparison between the two—in 6.3.4.

6.3.1 VectorWise baseline results

We evaluated the *whitebox compression* model against the VectorWise baseline according to the methodology (6.1.2). This section presents the results obtained with Configuration A (iterative greedy). Figure 6.3 shows the compression ratios—$ratio_{blackbox}$, $ratio_{whitebox}$—and the 3 sizes they were derived from—$size_{uncompressed}$, $size_{blackbox}$, $size_{whitebox}$—for each table. The tables are sorted in descending order by $ratio_{whitebox}$. In this figure we made the comparison for the full tables, including all logical columns, even though only a part of them are in the scope of *whitebox compression*.

The total size of the uncompressed tables is 131GB. The overall compression ratios are: $ratio_{blackbox} = 2.58$ and $ratio_{whitebox} = 3.58$. *Whitebox compression* has an overall ratio of 1.38 against blackbox compression. In the first chart we can observe that for one third of the tables $ratio_{whitebox}$ is significantly higher than $ratio_{blackbox}$. The rest of the

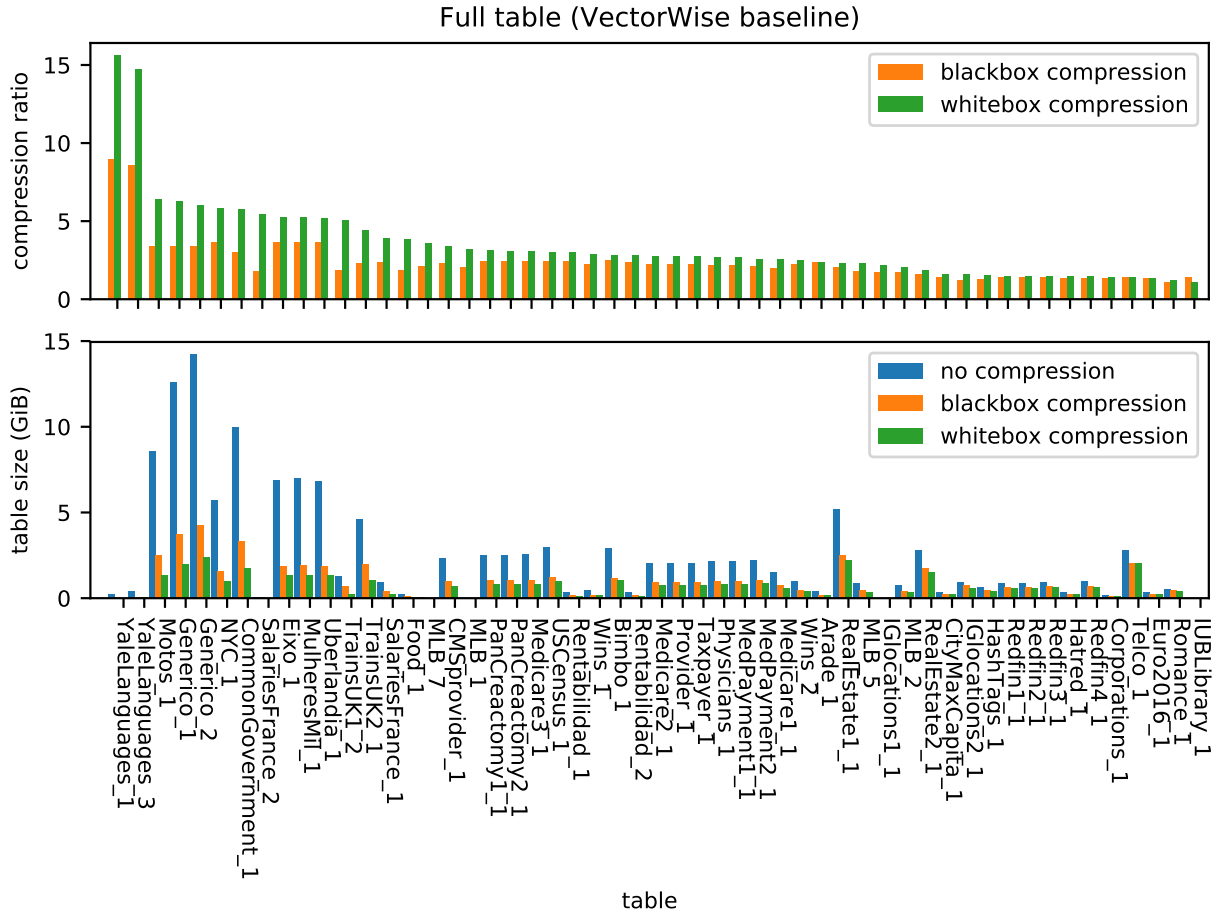


Figure 6.3: Full table comparison (VectorWise baseline)

tables have similar compression ratios. An important observation is that, even though *whitebox compression* only brings a significant improvement to a small part of the tables, it is never worse than blackbox compression. The only exception is the last table, where $ratio_{blackbox} = 1.38$ and $ratio_{whitebox} = 1.11$. However, this table is extremely small in uncompressed format: 508KB. Such small data already fits in the CPU cache and does not require compression at all. We can further notice that most of the tables with high $ratio_{blackbox}$ have even higher $ratio_{whitebox}$ and those with the lowest $ratio_{blackbox}$ have the lowest $ratio_{whitebox}$. A possible explanation for this phenomenon is that *whitebox compression* compresses the same columns as VectorWise—those with high repetition factors and redundancy—but manages to exploit these opportunities more efficiently and respectively, has a small improvement when there are no opportunities. From the second chart we can make the observation that the majority of the large tables tend to have higher compression ratios than most of the small tables, with some exceptions. This is particularly true for

whitebox compression, but also for blackbox, since the two are also somewhat correlated.

These results include all the logical columns, even tough some of them were not represented through *whitebox compression*. Our compression model does not incur any overhead on these columns and therefore we can exclude them from the evaluation with the purpose of measuring the improvement more accurately. Figure 6.4 shows the same charts but only includes the columns represented through *whitebox compression*. Measuring these results required column-level analysis. For this reason, we excluded the tables for which we couldn't match the columns with VectorWise's data files (because of multiple columns stored in the same file). Tables are sorted by the new $ratio_{whitebox}$.

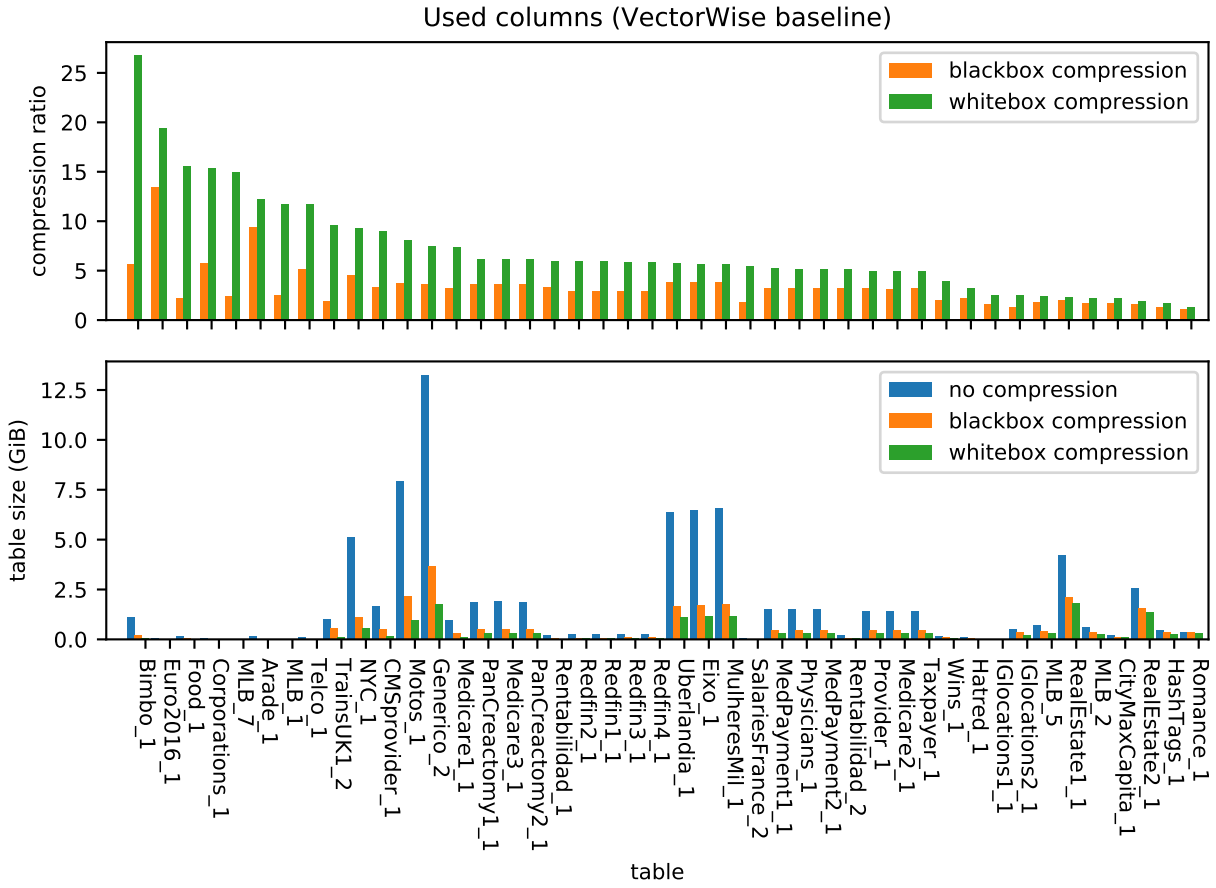


Figure 6.4: Used columns comparison (VectorWise baseline)

The results are as expected: the average compression ratio increased and *whitebox compression* brings a significant improvement for the majority of the tables. Table 6.1 shows an overall comparison between the full table results and the used columns results.

Only 68% of the total size of the data was represented through *whitebox compression*.

	$size_{uncompressed}$ (GB)	$ratio_{blackbox}$	$ratio_{whitebox}$	$\frac{ratio_{blackbox}}{ratio_{whitebox}}$
Full table	131	2.58	3.58	1.38
Used columns	77	3.16	5.16	1.63

Table 6.1: Full table vs. used columns (VectorWise baseline)

The increase of $ratio_{blackbox}$ shows that columns used by *whitebox compression* are also good candidates for blackbox compression. However, $ratio_{whitebox}$ increased more than $ratio_{blackbox}$, indicating that *whitebox compression* created opportunities for more compact representation.

The difference between the used columns evaluation and the full table evaluation is determined by the lack of opportunities for *whitebox compression* in part of the data. For a better understanding, imagine that we want to compress two columns c_a and c_b of equal size s. c_a presents no compression opportunities ($ratio_a = 1$) and c_b has $ratio_b = 10$. Even though half of the data is highly compressible, the total ratio of the two columns is much lower: $ratio_{total} = \frac{2 \times s}{s + 0.1 \times s} = 1.81$.

The conclusion that we can draw so far is that *whitebox compression* achieves high compression ratios on the columns that it represents through the expression trees and is never worse than blackbox compression alone. Additionally, the columns that do not present compression opportunities are not affected and the underlying database system operates normally on them.

6.3.2 Estimator model baseline results

We conducted an additional evaluation of the *whitebox compression* model, this time against the Estimator baseline, with the purpose of measuring the compression capabilities of a stand-alone *whitebox* system. The VectorWise blackbox compression schemes are replaced by the lightweight compression estimators defined in 5.2.2. We followed the methodology described in 6.1.3. This section presents the results obtained with Configuration A (iterative greedy). Figure 6.5 shows the compression ratios and table sizes, considering only the columns represented through *whitebox compression*.

The overall results are similar to the VectorWise baseline: *whitebox compression* is effective for part of the tables (around 40% of them in this case) and for the rest of them it gives similar compression ratios with the basic lightweight schemes. Similarly, the compression ratios are correlated: $ratio_{whitebox}$ is higher respectively lower where $ratio_{lightweight}$ is higher respectively lower. The same observation, that *whitebox compression* is never

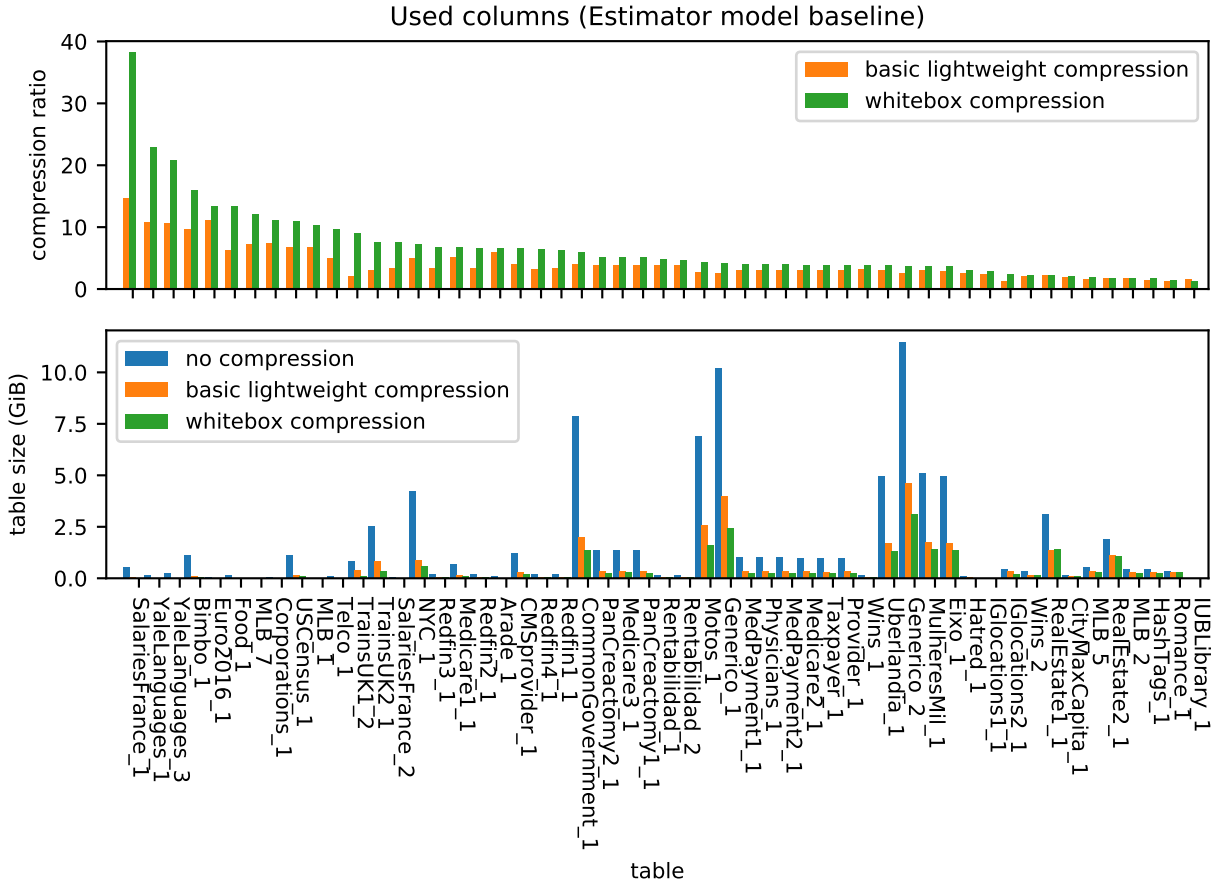


Figure 6.5: Used columns comparison (Estimator model baseline)

worse than the basic lightweight compression schemes alone, is also true for the estimator baseline, with the same exception: the last table. Table 6.2 shows an overall comparison between the 2 baselines.

	$size_{uncompressed}$ (GB)	$ratio_{blackbox}$	$ratio_{whitebox}$	$\frac{ratio_{blackbox}}{ratio_{whitebox}}$
VectorWise	77	3.16	5.16	1.63
Estimator model	83	2.87	4.04	1.40

Table 6.2: VectorWise baseline vs. estimator model baseline (used columns)

The total size of the used columns is 7% higher and the compression ratios decreased with 9% for the lightweight schemes and 21% for *whitebox compression*. The improvement of *whitebox compression* is also a bit lower (14% decrease). The Estimator baseline seems to reduce the impact of both compression systems and in particular the effect of our model. However, the objective of *whitebox compression*—to create compression opportunities for

more compact data representation—is still met, since it brings an increase in compression ratio from 2.87 to 4.04.

Figure 6.6 creates a better picture of the differences between the 2 baselines—a comparison of the table sizes for the three metrics in the methodology: $size_{uncompressed}$, $size_{lightweight}/size_{blackbox}$ and $size_{whitebox}$.

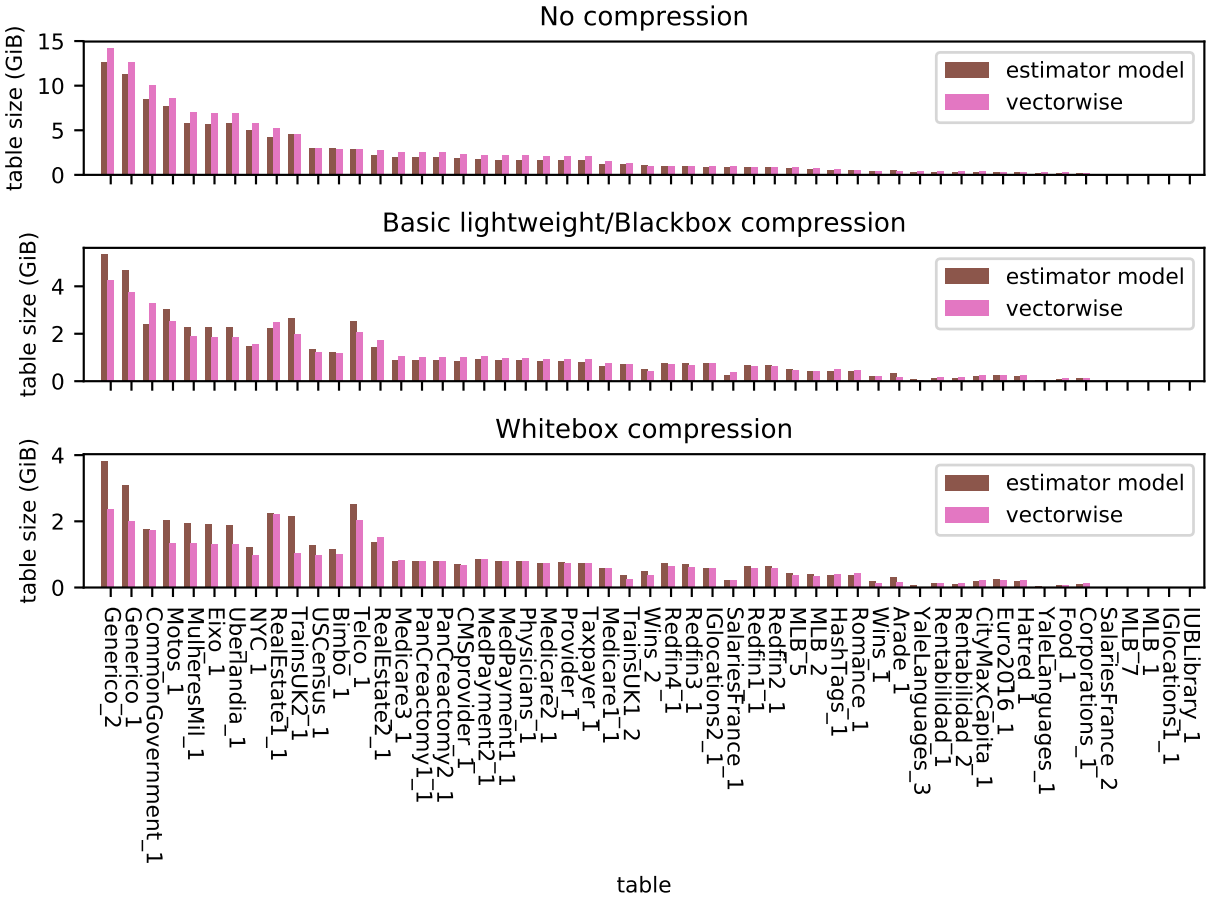


Figure 6.6: Baseline comparison (full table): Estimator model vs. Vectorwise

In the first chart we can see that the estimated uncompressed size of the data is a bit smaller than the VectorWise size for most tables. This result is not unexpected, since the storage layer of a real system is more complex and might add additional overhead compared to the theoretical estimation that we made. Moreover, the exact representation of each data type might differ from the one that we used. The second chart shows how VectorWise's blackbox compression schemes perform better than the (estimated) lightweight compression methods for some tables. This is not a surprising result, since our lightweight compression schemes are not optimized and the exception handling mechanism is different.

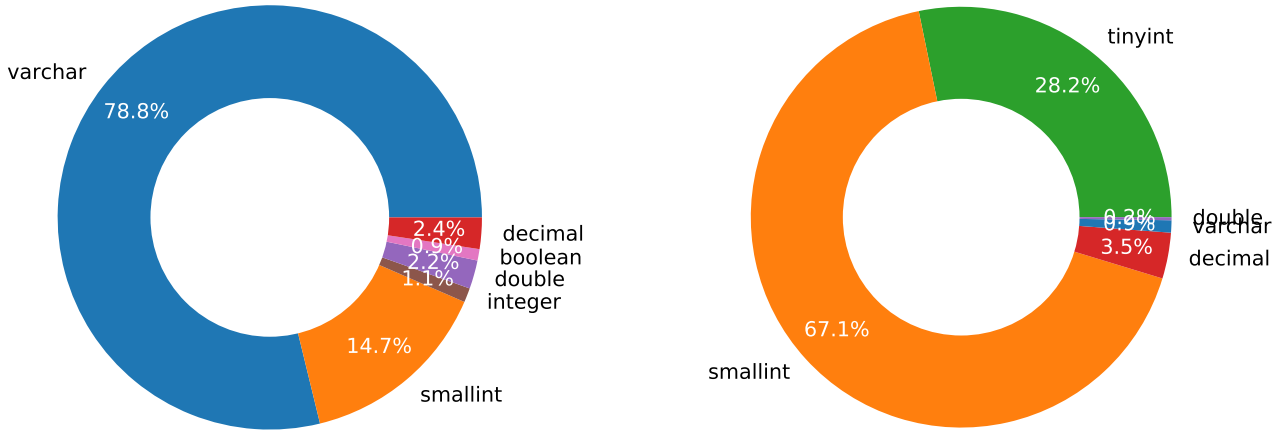
A somewhat unexpected result can be observed in the third chart: VectorWise is even better at compressing the whitebox representation of the data than the estimator model for some tables. A possible explanation might be that the number, type and size of the physical columns differ, as *whitebox compression* creates many nullable columns because of the exception handling mechanism, while VectorWise stores exceptions together with the compressed data, in blocks with custom format. Moreover, the compression metadata differs and its size is computed in a different way. An additional notable observation is that the difference between VectorWise and the estimator model is usually higher where the size of the data is larger, while for smaller tables the two models give closer results.

The overall conclusion that we can draw from this experiment is that, even though the two models give somewhat different results, *whitebox compression* still brings a significant improvement over the existing lightweight compression methods. The main reason for the lower compression ratios of the Estimator baseline is the different—and unoptimized—compression schemes. A more thorough experiment and analysis needs to be performed in order to properly evaluate the performance of a stand-alone *whitebox* system in practice—we leave this for future work. Even so, we showed that *whitebox compression* can be used to enhance existing systems—like VectorWise—as an intermediate layer before the optimized compression methods.

6.3.3 Results analysis

To better understand the impact of *whitebox compression* and how it represents the data, we performed an analysis of the physical data size components and the expression trees. We conducted this analysis on the results obtained with the VectorWise methodology (6.1.2). This section presents the results obtained with Configuration A (iterative greedy).

Figure 6.7a shows the distribution of datatypes across the logical columns represented through *whitebox compression*. The majority of columns are `VARCHAR`, since our system leverages the opportunities present in strings. The rest of the columns are numeric and boolean and are all constant columns, since the Constant pattern detector is the only one that works on other datatypes than strings. Figure 6.7b shows the distribution of datatypes across the physical columns resulted after the *whitebox* representation (excluding exception columns). With the exception of 1% `VARCHAR` columns, all the other are numeric. They resulted from Numeric strings and Dictionary representations (observation: dictionary ids are stored in SQL numeric datatypes so that they can be loaded into VectorWise; however, the main purpose of Dictionary expression nodes is to serve as an intermediate representation layer before Column correlation). The only pattern detector that outputs



(a) Logical columns (used) **(b)** Physical columns

Figure 6.7: Column datatype distribution

`VARCHAR` columns is Character set split. The other 2 pattern detectors—Constant and Column correlation—do not output any physical columns. Instead, they reduce the number of columns by consuming them and only storing metadata.

Table 6.4 shows the impact of *whitebox compression* through an analysis of the logical and physical columns in terms of numbers and sizes.

		Logical columns	Physical columns	
		Used columns	Data columns	Exception columns
Average	Count	26	8	46
	Size (MB)	1.45GB	85MB	262MB
Total	Count	1406	462	2479
	Size	77.2GB	3.7GB	11.3GB

Table 6.3: Logical vs. physical columns

On the average table, there are 26—out of 67 (38%)—logical columns used in the *white-box* representation. They are represented through only 8 physical columns containing compressed data and an additional 46 exception columns. The number of data columns is reduced because of the expression nodes that consume columns: Constant and Column correlation. The number of exception columns is very high due to our option to keep an exception column for each expression node in the tree and to allow recursive compression of exception columns. However, these columns are very sparse and contain mostly null values—which are effectively stored by the underlying database system through a bitmap. In terms of size, the average of 1.45GB of input data is represented through only 85MB of

compressed data and 262MB of exceptions. Therefore, 1.19GB of the data (non-exceptions: 1.45GB - 262MB) is represented through 85MB of compressed data—plus the size of the metadata, which is insignificant (see Figure 6.8). These results show the high degree of redundancy present in real data—and that we can squeeze this redundancy out of the data if we have a proper exception handling mechanism. Table 6.3 also shows the same analysis for the overall results—total of all tables—instead of the average.

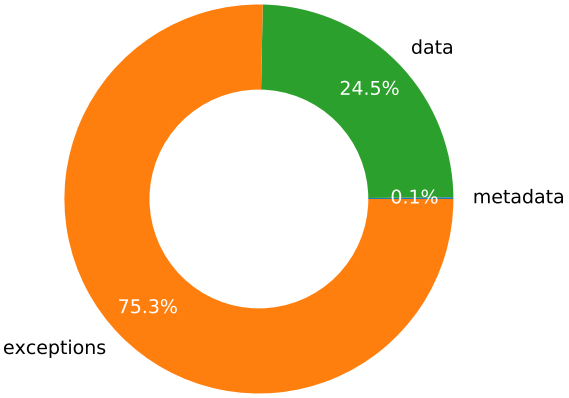


Figure 6.8: Physical size distribution

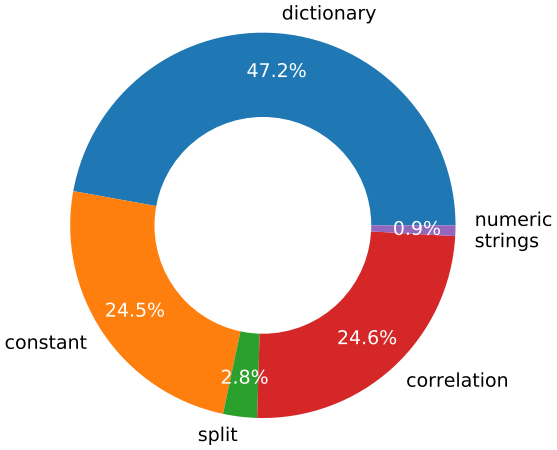


Figure 6.9: Expression node types distribution

Figure 6.8 shows a better picture of the physical data size distribution and the components that make it up. The exceptions sum up to 75.3% while the compressed data represents 24.5% of the total physical data size. The compression metadata (e.g. dictionaries, correlation maps, etc.) is insignificant, since we used a single compression tree for the entire table. The alternative—handling blocks of data separately—would allow a more fined grained representation with possibly simpler compression trees, at the expense of increasing the total metadata size.

We also computed the exception ratio in terms of numbers instead of size (i.e. how *many* exceptions there are) as follows: $\frac{count_{exception}}{count_{input}}$, where $count_{exception}$ is the total number of values on all leaf exception columns (i.e. physical columns, not further represented through other operators) and $count_{input}$ is the total number of values on the corresponding parent columns from which the exceptions originated, which can be either inner or logical columns. We obtained an overall exception ratio of 0.16. Given the fact that exceptions make most of the physical size, reducing the exception ratio may be a way of achieving higher compression ratios. A more thorough analysis of these results, with a focus on exceptions, should be performed in order investigate ways of reducing the exception ratio—we leave it for future work.

For the rest of our analysis we focused our attention on the expression trees. Recall that the so called expression tree is actually a directed acyclic graph (DAG) with multiple root nodes and connected components (4 Compression model). Each connected component represents a subset of the logical columns as a function of physical columns. Table 6.4 shows the characteristics of the average expression tree.

Connected components (on average per table)	Expression nodes (on average per connected component)	Depth	Logical columns	Physical columns
12.5	3.3	1.4	1.7	0.8

Table 6.4: Expression tree statistics

There are 12.5 connected components, with an average of 3.3 expression nodes and a depth of 1.4 levels. This shows that expression trees are not very complex and relatively fast to evaluate. A connected component represents on average 1.7 logical columns as a function of 0.8 physical columns (excluding exception columns)—the 0.8 value is due to the expression nodes that do not output any physical columns: Constant and Column correlation. Here we see again the capacity of *whitebox compression* to reduce the number of columns by storing metadata instead.

Finally, Figure 6.9 shows the distribution of the expression node types in the average expression tree. In this analysis we considered both internal and leaf nodes. We notice that the majority is composed of Constant, Dictionary and Column correlation, while only a small percentage are Character set split and Numeric strings. This is due to the high Dictionary compression potential of the data and to the high correlation between columns—high redundancy in other words. The split operators create opportunities for compressing subcolumns and thus, for every Character set split node there are many Constant, Dictionary and Column correlation nodes. Also note that all Column correlation nodes take as input—and consume—Dictionary expression nodes. Therefore, the majority of the Dictionary nodes are internal nodes of the expression tree. An additional reason for this distribution of expression nodes is the priority configuration that we used for this experiment—recall that Constant and Dictionary have the highest priority (6.2 Experimental setup). We also experimented with other configurations of the Priority pattern selector which resulted in more even distributions of the expression node types, but decided to show the results of this configuration because they were better. Moreover, we will see in the next section that this choice of expression node types is also optimal with respect to our cost model (5.2.2), since the Recursive exhaustive learning algorithm produced a similar distribution, while trying to minimize the physical data size.

The take-away message from this analysis is that we can represent many logical columns—mostly `VARCHAR`—as functions of fewer physical columns—mostly numeric—at the expense of many exception columns—mostly nulls—and achieve high compression ratios—all of this automatically learned.

6.3.4 Recursive exhaustive learning results

We repeated the same experiments presented so far, this time with Configuration B (recursive exhaustive). This section presents an analysis of the results in comparison to the ones obtained with Configuration A (iterative greedy). As expected, the non-greedy, cost model-based approach brought an improvement in terms of compression ratio.

		$size_{uncompressed}$ (GB)	$ratio_{blackbox}$	$ratio_{whitebox}$	$\frac{ratio_{blackbox}}{ratio_{whitebox}}$
Full table	Iterative greedy	131	2.58	3.58	1.38
	Recursive exhaustive			3.69	1.43
Used columns	Iterative greedy	77	3.16	5.16	1.63
	Recursive exhaustive	98	3.35	6.45	1.92

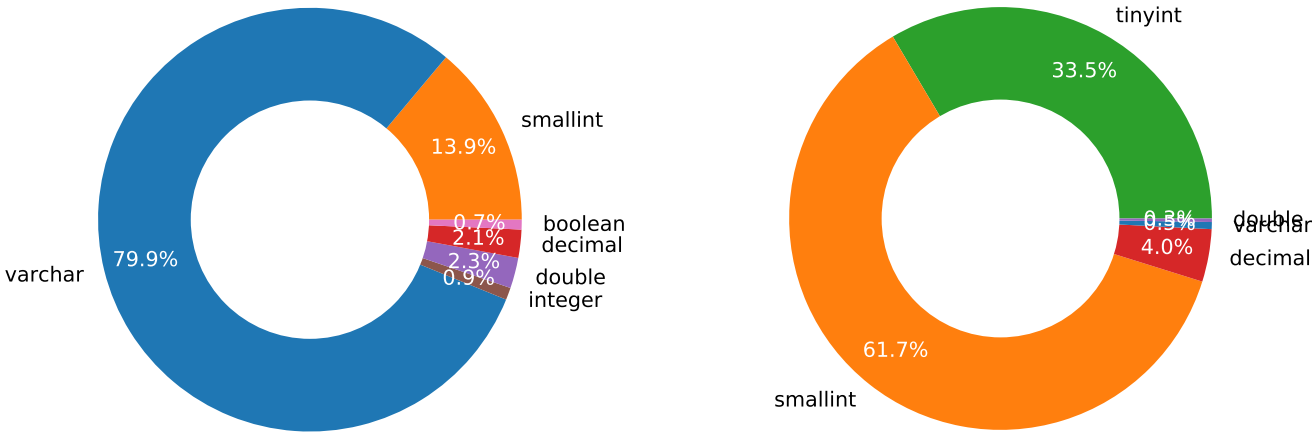
Table 6.5: Iterative greedy vs. Recursive exhaustive (VectorWise baseline)

Table 6.5 shows the sizes and compression ratios obtained with the two configurations. For the full data, Configuration B gave a slightly higher compression ratio, leading to an overall improvement of 1.43 over VectorWise. The two algorithms resulted in different compression trees and thus the columns represented through *whitebox compression* are not the same. Because of this, $size_{uncompressed}$ and $ratio_{blackbox}$ differ for the used columns: the recursive exhaustive algorithm selected a larger subset of the data (74% of the total size), which is also compressed slightly better by VectorWise. *Whitebox compression* achieves a significantly higher compression ratio of 6.45 on the used columns—an improvement of 1.92× over the existing blackbox methods.

		Logical columns	Physical columns	
		Used columns	Data columns	Exception columns
Average	Count	22	7	41
	Size (MB)	1.96GB	56MB	256MB
Total	Count	1072	373	2046
	Size	98.1GB	2.7GB	12.5GB

Table 6.6: Logical vs. physical columns (recursive exhaustive learning)

Table 6.6 and Figure 6.10 show a analysis of the logical and physical columns. Configuration B used less logical columns than Configuration A, but with a larger overall size. The



(a) Logical columns (used) **(b)** Physical columns

Figure 6.10: Column datatype distribution (recursive exhaustive learning)

distribution of datatypes across these columns is similar in both cases: ≈ 80% `VARCHAR` and ≈ 20% other datatypes. In terms of physical columns, the compressed data columns are both fewer in numbers and smaller in size. The total number of exception columns is 17% smaller but their total size is 9% higher—in a similar trend with the logical columns. The datatype distribution across the physical columns is also similar: >99% numeric, with an increase of the `TINYINT` percentage at the expense of `SMALLINT`. Figure 6.11 shows how the total physical size is split between compressed data, exceptions and metadata. Similarly to Configuration A, metadata is insignificant and exceptions make up most of the size. The exception ratio (in terms of numbers) is slightly smaller: 0.15 instead of 0.16.

Connected components (on average per table)	Expression nodes	Depth	Logical columns	Physical columns
	(on average per connected component)			
11.5	3.6	1.3	1.9	0.7

Table 6.7: Expression tree statistics (recursive exhaustive learning)

Figure 6.12 and Table 6.7 present the results of the expression tree analysis. The distribution of expression nodes is similar to the one given by Configuration A, with a slight increase of the Column correlation percentage and decrease of Dictionary and Constant percentages. Moreover, the expression trees are comparable in terms of structure (number of connected components, nodes, depth, logical and physical columns). These results confirm that the rules and heuristics used in the Iterative greedy learning algorithm and the configuration of the Priority pattern selector are suitable choices, since the greedy algo-

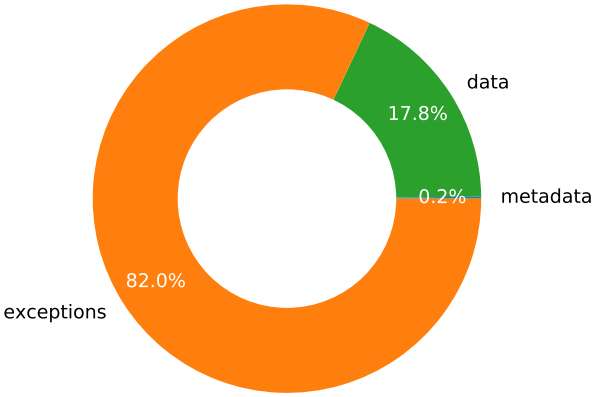


Figure 6.11: Physical size distribution (recursive exhaustive learning)

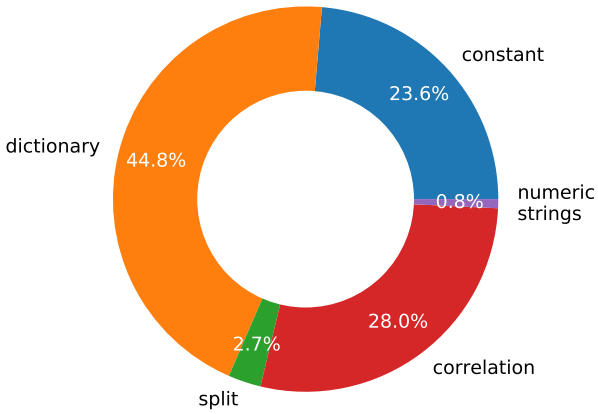


Figure 6.12: Expression node types distribution (recursive exhaustive learning)

rithm is capable of exploiting compression opportunities in similar ways as the exhaustive algorithm does.

We have seen how Configuration B (recursive exhaustive) achieves higher compression ratios than Configuration A (iterative greedy) by using an exhaustive search based on the compression estimation cost model (5.2.2) instead of making greedy choices based on pattern selection heuristics (5.2.4). This improvement comes at the cost of compression learning time. Our `Python` implementation of the learning engine takes around 1 minute with Configuration A and around 5 minutes with Configuration B to learn the compression tree for a table, depending on the sample size, number and type of columns. Most of this time is spent by analysing the sample for each (sub)column with the pattern detectors and, in the case of Configuration B, by estimating the size of each (sub)column with the compression estimators. The execution time of the recursive exhaustive algorithm is limited by the maximum height threshold that we imposed on the expression tree. Adjusting this threshold might result in better compression ratios at the cost of longer learning times.

All in all, we defined and evaluated two different *whitebox compression* learning algorithms which create compact representations of the data. Their execution times can be improved through more efficient implementations in lower level programming languages, but even so, they are practical, as compression learning happens during bulk loading of the data.

7

Conclusion and future work

7.1 Conclusion

In this thesis we explored the concept of *whitebox compression*. We defined and analysed the Public BI benchmark in search for compression opportunities, by studying real, user-generated datasets. Based on our findings we defined a new compression model which uses elementary operators to represent data more compactly. We further defined an automatic compression learning process and created a proof-of-concept implementation to measure its feasibility and compression potential. Let us recall the research questions together with their answers.

What does real user generated data look like—specifically in the case of the Public BI benchmark? From our analysis we concluded that real data is redundant and represented in inefficient ways, from "suboptimal" datatypes, to highly correlated or even duplicate columns. Most of the string columns in the Public BI benchmark have a high repetition factor, making them suitable for dictionary-like encoding techniques.

How good are existing compression schemes at compressing real data? Due to the high number of numeric columns and the low number of unique values in string columns—present in the Public BI benchmark—the data is already suitable for compression with existing lightweight methods. VectorWise achieves an overall compression ratio of 2.58 on the benchmark. However, real data has a considerable compression potential that is not exploited by these systems.

Can we represent the logical columns more compactly through an expression tree composed of elementary operators? Based on the compression opportunities found in the datasets we defined an expression language that enables more efficient representation of the data. The *whitebox compression* model achieves high compression ratios

by splitting columns into subcolumns, storing data in the appropriate format, and representing columns as functions of other columns. This complex compression schemes are actually the result of simple recursive representation of columns through elementary operators. Combined with a transparent mechanism of handling (and recursively compressing) exceptions, we manage to store data more compactly.

Can we create an automatic learning process that will generate suitable compression trees for each column? We defined and implemented a set of pattern detectors which identify compression opportunities present in the data and evaluate its potential for *whitebox* representation. We further defined the optimization problem of finding the best representation for a set of columns and proposed algorithms that solve it using an estimator cost model and greedy heuristics. We validated our compression model with a proof-of-concept implementation, achieving an overall compression ratio of 3.69 on the full data and 6.45 on the columns represented through *whitebox compression*—an almost factor 2× (1.92) improvement of VectorWise.

Besides improved compression ratios, *whitebox compression* in itself is an important contribution to the database research field as a new compression model. The transparent representation of logical columns as functions of physical columns through operator expressions simplifies the compression layer of database systems. It allows implicit recursive compression through an unlimited number of methods, while handling exceptions in a generic way. On top of this, *whitebox compression* has the potential of improving query processing times by exposing the data representation to the query execution engine, allowing predicate push-down and lazy evaluation of the compression tree. This model can be implemented either as a stand-alone system with *whitebox* versions of the existing compression methods or as an intermediate representation layer which creates compression opportunities for the optimized blackbox schemes.

On a more general level, *whitebox compression* and the learning of representation models can be seen as part of a trend to adapt database storage and query processing methods to the data, similarly to *learned indexes* (35) or *learned systems* (36, 37). This idea does have important system-wide consequences as well: a system that reads *whitebox* compressed data will read compression expressions from the block header and will have to quickly instantiate some query evaluation infrastructure to parse it, performing decompression and/or predicate push-down. We can think of methds that employ JIT code generation for this, as well as vectorized execution methods leveraging SIMD. Keeping the latency and overhead of this process low is a research question that we leave for future work.

7.2 Future work

For this thesis we explored and demonstrated the potential of *whitebox compression* through a basic implementation. There is room for improvement and further development in all the components: pattern detectors, pattern selectors, learning algorithms, estimators. However, even with this initial exploratory approach we obtained good results, which only encourages future work in this direction, towards a highly optimized *whitebox compression* model implemented in real systems.

Firstly, the Public BI benchmark deserves a more thorough characterisation, as it is a representative benchmark for database systems. In terms of data, an interesting result would be the distribution types of the numeric columns: skweness and kurtosis coefficients plotted on a Cullen and Frey graph. Moreover, outlier characterisation and the range and domain of values might also prove useful. Special attention should be directed towards the queries, to understand real use-cases in analytical systems.

There are compression opportunities in some datasets that are not yet covered by the current compression learning process, but could be easily exploited with a few improvements: 1) support for hexadecimal numbers in the Numeric strings pattern detector (hex-formatted columns in RealEstate* datasets); 2) a dedicated pattern selector for Character set split which takes into account the number of characters that are not in the default charset; 3) padding whitespace detection and isolation on separate columns through a split instance to enable future compression with Dictionary; 4) column correlation support for continuous and discrete variables to leverage mathematical dependencies between numeric columns; 5) a different column split pattern detector based on frequencies of n-grams in string columns—approach and preliminary results are presented in Appendix A.

Whitebox compression has potential for achieving fast query execution. However, this potential needs to be properly evaluated. Most importantly, we need efficient compression and decompression implementations. So far we implemented unoptimized versions of these procedures to evaluate the compression ratios and validate the correctness of our implementation by reconstructing the original data. An important step towards improved execution time is to use a different cost model for the learning algorithms—one that also takes into account the complexity of the compression tree (e.g. in terms of depth and branching factor). Going further, we can design a compression tree optimization process with the purpose of reducing its complexity (similar to query plan optimization). Furthermore, we should answer our 5th research question mentioned in the introduction: Can we achieve compressed execution with the *whitebox compression* model? We need to study

predicate push-down opportunities from the perspective of both data and queries, with the hope that we can (partially) skip decompression and operate directly on compressed data.

Finally, a *machine learning* approach for pattern detection and solving the compression learning optimization problem might lead to interesting results, provided that we can extract relevant features from the datasets and properly define our problem so that it fits the *machine learning* models.

References

[1] Marcin Zukowski, Sandor Heman, Niels Nes, and Peter Boncz. *Super-scalar RAM-CPU cache compression*. IEEE, 2006. 1, 7, 8, 66

[2] Peter Boncz, Thomas Neumann, and Orri Erling. **TPC-H analyzed: Hidden messages and lessons learned from an influential benchmark**. In *Technology Conference on Performance Evaluation and Benchmarking*, pages 61–76. Springer, 2013. 1, 8

[3] Raghunath Othayoth Nambiar and Meikel Poess. **The making of TPC-DS**. In *Proceedings of the 32nd international conference on Very large data bases*, pages 1049–1058. VLDB Endowment, 2006. 1, 8

[4] **Public BI Benchmark**. `https://github.com/cwida/public_bi_benchmark`. Accessed: 2019-03-13. 1, 8, 11, 69

[5] Daniel Abadi, Samuel Madden, and Miguel Ferreira. **Integrating compression and execution in column-oriented database systems**. In *Proceedings of the 2006 ACM SIGMOD international conference on Management of data*, pages 671–682. ACM, 2006. 7, 8

[6] Harald Lang, Tobias Mühlbauer, Florian Funke, Peter A Boncz, Thomas Neumann, and Alfons Kemper. **Data blocks: hybrid OLTP and OLAP on compressed storage using both vectorization and compilation**. In *Proceedings of the 2016 International Conference on Management of Data*, pages 311–326. ACM, 2016. 7, 8

[7] Orestis Polychroniou and Kenneth A Ross. **Efficient lightweight compression alongside fast scans**. In *Proceedings of the 11th International Workshop on Data Management on New Hardware*, page 9. ACM, 2015. 7

[8] Goetz Graefe and Leonard D Shapiro. **Data compression and database performance**. In *[Proceedings] 1991 Symposium on Applied Computing*, pages 22–27. IEEE, 1991. 7

[9] Alfons Kemper and Thomas Neumann. **HyPer: A hybrid OLTP&OLAP main memory database system based on virtual memory snapshots**. In *2011 IEEE 27th International Conference on Data Engineering*, pages 195–206. IEEE, 2011. 7, 11

[10] Marcin Zukowski, Mark Van de Wiel, and Peter Boncz. **Vectorwise: A vectorized analytical DBMS**. In *2012 IEEE 28th International Conference on Data Engineering*, pages 1349–1350. IEEE, 2012. 7, 11, 14, 46, 65

[11] Jonathan Goldstein, Raghu Ramakrishnan, and Uri Shaft. **Compressing relations and indexes**. In *Proceedings 14th International Conference on Data Engineering*, pages 370–379. IEEE, 1998. 7

[12] Daniel Lemire and Leonid Boytsov. **Decoding billions of integers per second through vectorization**. *Software: Practice and Experience*, **45**(1):1–29, 2015. 7

[13] Mark A Roth and Scott J Van Horn. **Database compression**. *ACM Sigmod Record*, **22**(3):31–39, 1993. 7

[14] Vijayshankar Raman, Gopi Attaluri, Ronald Barber, Naresh Chainani, David Kalmuk, Vincent KulandaiSamy, Jens Leenstra, Sam Lightstone, Shaorong Liu, Guy M Lohman, et al. **DB2 with BLU acceleration: So much more than just a column store**. *Proceedings of the VLDB Endowment*, **6**(11):1080–1091, 2013. 7, 8

[15] Jae-Gil Lee, Gopi Attaluri, Ronald Barber, Naresh Chainani, Oliver Draese, Frederick Ho, Stratos Idreos, Min-Soo Kim, Sam Lightstone, Guy Lohman, et al. **Joins on encoded and partitioned data**. *Proceedings of the VLDB Endowment*, **7**(13):1355–1366, 2014. 7, 8

[16] Vijayshankar Raman and Garret Swart. **How to wring a table dry: Entropy compression of relations and querying of compressed relations**. In *Proceedings of the 32nd international conference on Very large data bases*, pages 858–869. VLDB Endowment, 2006. 8

[17] Patrick Damme, Dirk Habich, Juliana Hildebrandt, and Wolfgang Lehner. **Lightweight Data Compression Algorithms: An Experimental Survey (Experiments and Analyses).** In *EDBT*, pages 72–83, 2017. 8

[18] Adrian Vogelsgesang, Michael Haubenschild, Jan Finis, Alfons Kemper, Viktor Leis, Tobias Muehlbauer, Thomas Neumann, and Manuel Then. **Get real: How benchmarks fail to represent the real world**. In *Proceedings of the Workshop on Testing Database Systems*, page 1. ACM, 2018. 8, 11

[19] **Tableau Public**. `https://public.tableau.com`. Accessed: 2019-03-14. 8, 11

[20] Peter A Boncz, Marcin Zukowski, and Niels Nes. **MonetDB/X100: Hyper-Pipelining Query Execution.** In *Cidr*, **5**, pages 225–237, 2005. 11

[21] **Database SQL Language Reference. COALESCE**. `https://docs.oracle.com/cd/B28359_01/server.111/b28286/functions023.htm`. Accessed: 2019-07-29. 25

[22] Timo Kersten, Viktor Leis, Alfons Kemper, Thomas Neumann, Andrew Pavlo, and Peter Boncz. **Everything you always wanted to know about compiled and vectorized queries but were afraid to ask**. *Proceedings of the VLDB Endowment*, **11**(13):2209–2222, 2018. 25

[23] **MonetDB data types**. `https://www.monetdb.org/book/export/html/187`. Accessed: 2019-07-05. 32

[24] **VectorWise decimal data type**. `https://docs.actian.com/ingres/10s/index.html#page/SQLRef/Decimal_Data_Type.htm`. Accessed: 2019-07-05. 32

[25] Harald Cramir. **Mathematical methods of statistics**. *Princeton U. Press, Princeton*, page 282, 1946. 38

[26] William Press, Brian Flannery, Saul Teukolsky, and William Vetterling. **Numerical Recipes: the Art of Scientific Computing (3rd ed.)**. *Cambridge U. Press, Cambridge*, page 761, 1992. 38

[27] **Ingres 10.2. OpenAPI User Guide**. `https://supportactian.secure.force.com/help/servlet/fileField?id=0BEf3000000PLPU`. Accessed: 2019-07-02. 46

[28] Lefteris Sidirourgos and Martin Kersten. **Column imprints: a secondary index structure**. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, pages 893–904. ACM, 2013. 65

[29] Vladimir I Levenshtein. **Binary codes capable of correcting deletions, insertions, and reversals**. In *Soviet physics doklady*, **10**, pages 707–710, 1966. 67

[30] **regex 2019.06.08**. https://pypi.org/project/regex. Accessed: 2019-07-18. 67

[31] **vwload Command–Load Data into a Table**. https://docs.actian.com/ingres/11.0/index.html#page/CommandRef%2Fvwload_Command--Load_Data_into_a_Table.htm. Accessed: 2019-07-18. 68

[32] **Performance Tips**. https://docs.actian.com/vector/4.2/index.html#page/User/Performance_Tips.htm. Accessed: 2019-07-18. 68

[33] **Linux Programmer's Manual. STAT(2)**. http://man7.org/linux/man-pages/man2/stat.2.html. Accessed: 2019-07-18. 68

[34] **statdump Command–Print Statistics in iistats and iihistogram Catalogs**. https://docs.actian.com/vector/5.0/index.html#page/User/statdump_Command--Print_Statistics_in_iistats_an.htm. Accessed: 2019-07-18. 68

[35] Tim Kraska, Alex Beutel, Ed H Chi, Jeffrey Dean, and Neoklis Polyzotis. **The case for learned index structures**. In *Proceedings of the 2018 International Conference on Management of Data*, pages 489–504. ACM, 2018. 84

[36] Stratos Idreos and Tim Kraska. **From auto-tuning one size fits all to self-designed and learned data-intensive systems**. In *ACM SIGMOD*, 2019. 84

[37] Tim Kraska, Mohammad Alizadeh, Alex Beutel, Ed H Chi, Jialin Ding, Ani Kristo, Guillaume Leclerc, Samuel Madden, Hongzi Mao, and Vikram Nathan. **Sagedb: A learned database system**. 2019. 84

Appendices

Appendix A

n-gram frequency analysis

During our analysis of the Public BI benchmark we noticed several `VARCHAR` columns containing strings that are concatenations of data from different distributions. These columns can be stored more efficiently in separate subcolumns, in order to allow independent compression of their constituent parts. We approached this problem with the Character set split pattern detector (5.1.4) and obtained good results, but we did not cover all the variations of such columns. Table A.1 shows two samples of data from columns that have this property but we did not cover.

ds_email	ds_tipo_beneficiario
ic.caetano@hotmail.com	Outro: especificar - bolsa formação estudante
arnaldo2moraes@yahoo.com.br	Outro: especificar - Programa Nacional de [...]
s_heique266@yahoo.com.br	Outro: especificar - cadastro unico
silasvkj@hotmail.com	Outro: especificar - aluno ensino Médio
leonel_noga@hotmail.com	Beneficiário de [...] - Beneficiário de [...]
alexelidi@yahoo.com.br	Outro: especificar - aluno da rede publica
jdcn488@hotmail.com	Outro: especificar - aluno escola publica
elton.t.santos@gmail.com	Atendimento prioritário - Atendimento prioritário
kassiosgoda@hotmail.com	Outro: especificar - Outro: especificar
valdireneandrade1@hotmail.com	Outro: especificar - Outro: especificar
diego.shynomory@gmail.com	Outro: especificar - ESTUDANTE REDE PUBLICA
fer-ars@hotmail.com	Outro: especificar - nao é perfil cadunico
rosilda_limactba@hotmail.com	Outro: especificar - CADÚNICO
utfpraasma@hotmail.com	Outro: especificar - possui renda inferior [...]

Table A.1: Eixo_1 data samples

Both columns are composed of two parts: one with high repetition factor—which can benefit from compression—and one rather unique—which cannot. They are separated by a delimiter character: `'@'`, respectively `'-'`. While the Character set split could cover these columns with a delimiter character set (e.g. `[@-]`), there are also cases with no explicit

separator character but other structures instead (e.g. fixed number of characters from the start of each string).

To cover all cases in a more generic way, we tried to develop a different approach based on n-gram frequencies. We represented each string in a different way, such that we can identify its constituent substrings more easily, as follows: 1) we computed the frequencies of all 3-grams based on their number of occurrences on the entire column; 2) we replaced each character in each string with the frequency of the 3-gram starting at the position of the character. The results are presented in Figures A.1 and A.2.

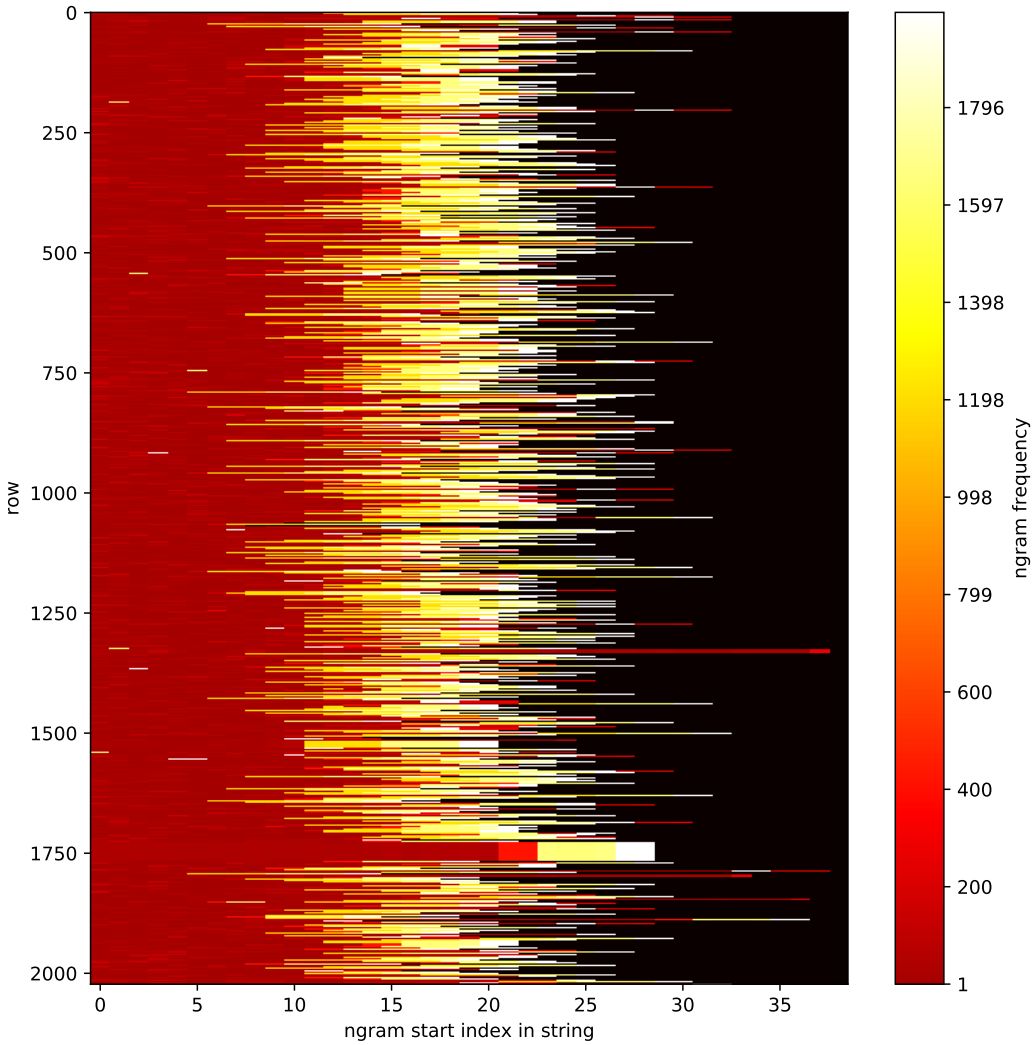


Figure A.1: *ds_ email* 3-gram frequencies

The figures show the frequency of the 3-grams in the strings in the two columns presented above (larger samples than in the example). The Y axis represents each string on the

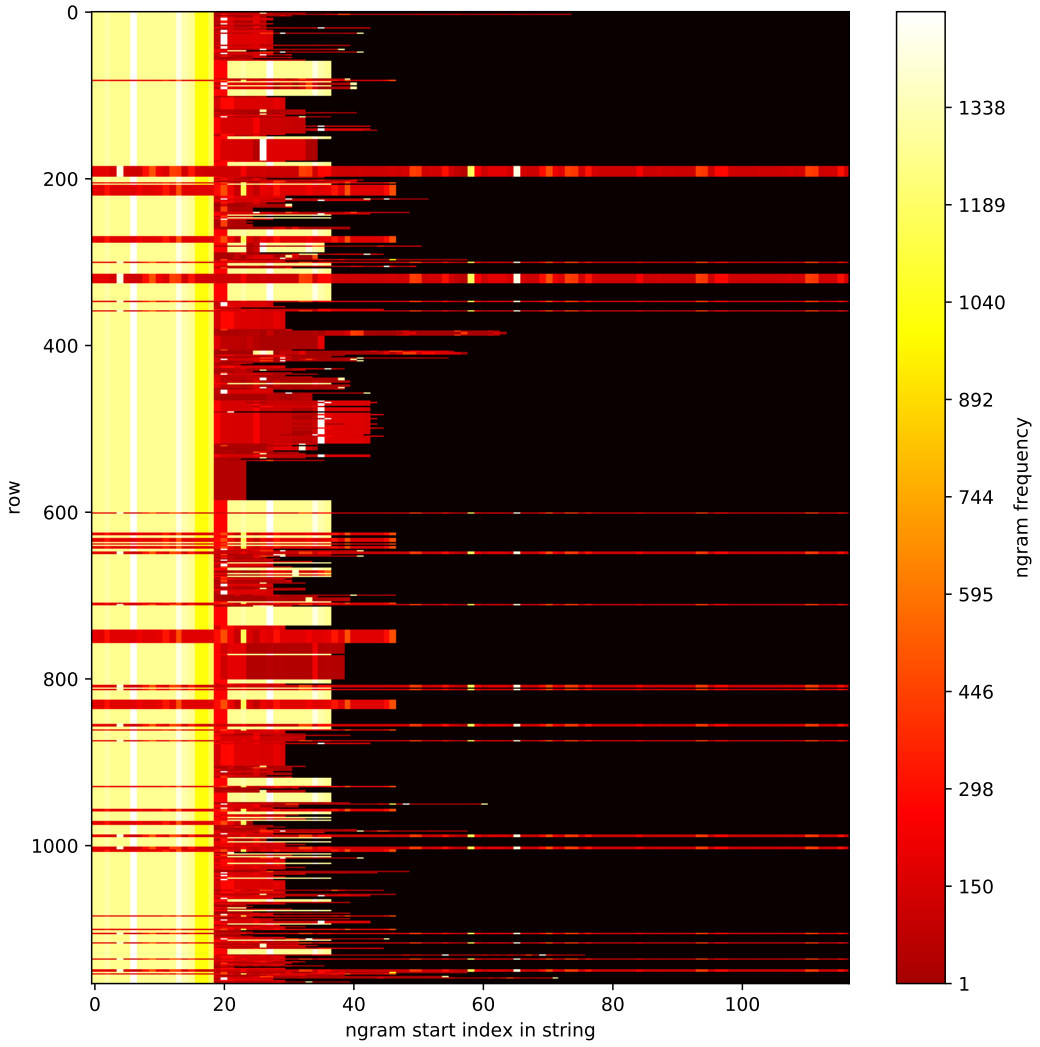


Figure A.2: *ds_tipo_beneficiario* 3-gram frequencies

column and the X axis contains the positions of the characters in the strings. The colors represent the frequencies: white = high frequency, red = low frequency. Black is used as padding for shorter strings. Figure A.1 shows the results for the *ds_email* column. We notice how all the emails start with low frequencies (the local/username part) and end with high frequencies (the domain part). Moreover, we can also observe the different components of the domain: we have the top level domains with the highest frequencies (e.g. `.com`, `.br`, etc.), followed by the subdomains (e.g. `gmail`, `hotmail`, etc.). Even more, the word `"mail"` is also representative in itself, since it is common for most email subdomains. Figure A.2 shows the results for the *ds_tipo_beneficiario* column. It is clear how the majority of the strings start with the frequent substring `"Outro: especificar"`

and end in a low frequency substring.

The 3-gram frequency representation emphasizes the constituent subparts in which a column should be split in order to enable independent compression. The goal is to automatically find the split points, which is not a trivial task. The approach that we propose relies on the fact that we can view each string as a series of numbers (3-gram frequencies). The series contain edges and plateaus. An edge is a significant increase or decrease in frequency. After identifying the edges in each string, we need to find the common edges amongst the majority of the strings. A common edge is an edge that can be identified either by the same character (e.g. the `'@'` delimiter) or by the same absolute position in all the strings. Once we found the common edges, we define the split points based on the character or position. During compression, we split each string according to these identifiers and mark as exception the values that cannot be split.

Due to time constraints, we decided not to proceed with the implementation and evaluation of this approach and leave it for future work.

Appendix B

Column correlation graphs

The figures below show two examples of Column correlation graphs as defined in 5.2.4.4. Figure B.1 shows a common correlation graph resulted from Configuration B (6.2) after applying the Column correlation pattern detector (5.1.5) on the leaf columns of the tree produced by the Recursive exhaustive learning algorithm (5.2.3) in *Stage-1* (table: *YaleLanguages_1*). Figure B.2 shows a highly complex correlation graph, resulted from Configuration A (6.2) (table: *Generico_1*). Both configurations resulted in graphs of similar complexities for the same tables. An important note is that these correlations are not directly between the logical columns, but between subcolumns resulted after decomposing them and separating the values into different subsets and exceptions as well. As we see in the figures, this process creates many correlation opportunities—and most of them could not be exploited on the original representation of the data.

Columns with high indegree can be determined by many other columns. This happens to columns that are (almost) constant and—in our case in particular—to exception columns with very high null ratios. The explanation is that, if a column has the same constant value (e.g. `null`), any other column can determine it since all its values will map to the constant value. Columns with high outdegree can determine many other columns. This happens to columns with very low repetition factor (i.e. rather unique), since there is a perfect mapping between a unique column and any other column. However, this does not happen in our case, since we only use Column correlation on Dictionary compressible columns. Besides these special cases of nearly constant or unique columns, high degrees result from actual correlations between (sub)columns created by the learning algorithms.

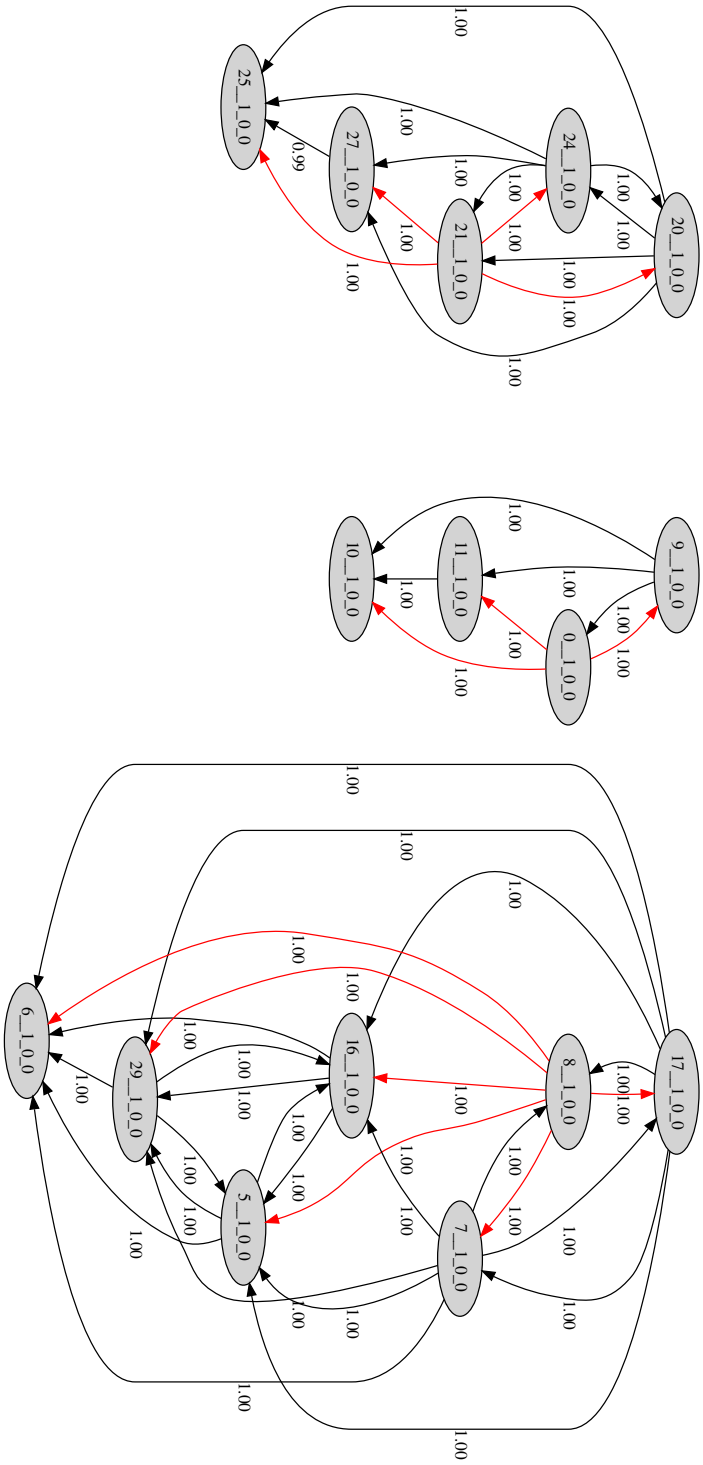


Figure B.1: *YaleLanguages_1* correlation graph

Figure B.2: *Generico_1* correlation graph