Otto von Guericke University of Magdeburg

Department of Computer Science

Master's Thesis

Key-Based Self-Driven Compression in Columnar Binary JSON

Author:

Oskar Kirmis

November 4, 2019

Advisors:

Prof. Dr. rer. nat. habil. Gunter Saake

M. Sc. Marcus Pinnecke

Institute for Technical and Business Information Systems / Database Research Group

Kirmis, Oskar:
Key-Based Self-Driven Compression in Columnar Binary JSON
Master's Thesis, Otto von Guericke University of Magdeburg, 2019

Abstract

A large part of the data that is available today in organizations or publicly is provided in semi-structured form. To perform analytical tasks on these – mostly read-only – semi-structured datasets, Carbon archives were developed as a column-oriented storage format. Its main focus is to allow cache-efficient access to fields across records. As many semi-structured datasets mainly consist of string data and the denormalization introduces redundancy, a lot of storage space is required. However, in Carbon archives – besides a deduplication of strings – there is currently no compression implemented.

The goal of this thesis is to discuss, implement and evaluate suitable compression techniques to reduce the amount of storage required and to speed up analytical queries on Carbon archives. Therefore, a compressor is implemented that can be configured to apply a combination of up to three different compression algorithms to the string data of Carbon archives. This compressor can be applied with a different configuration per column (per JSON object key). To find suitable combinations of compression algorithms for each column, one manual and two self-driven approaches are implemented and evaluated.

On a set of ten publicly available semi-structured datasets of different kinds and sizes, the string data can be compressed down to about 53% on average, reducing the whole datasets' size by 20%. Additionally, read operations are accelerated by approximately 6.5% on average. This means that the application of key-based compression can reduce storage costs and increase the query performance. Consequently, this technique should be used with Carbon archives, especially when working with large datasets.

Contents

List of Figures

List of Tables

Listings

1 Introduction

Motivation

The amount of data that is available in organizations or on the internet continues to grow. The majority of that data is not accessible through traditional database interfaces, though, but through web APIs like REST [Ric07], either because the data is provided by third party webservices or because an organization itself has decided to use a service-oriented IT architecture approach like microservices. While each service itself may be using one or more traditional database management systems (DBMS), these are not directly exposed and the data is serialized to data exchange formats like JSON [Bra14] or XML [BPSM+08] at the interfaces of the service.

However, these formats are designed to be used as exchange formats and are therefore focused on properties like being human readable and writable or easy – not fast – to parse instead of efficiency, both in terms of storage and processing performance. That results in a large overhead when reading the datasets. Also, storing a document in its original, record-oriented structure can lead to bad caching behavior when accessing the same attributes across many records. Both aspects make exchange formats unsuitable for directly running analytical workloads on them. Nevertheless, without being able to analyze it properly, especially for large datasets, the data often becomes worthless. For that kind of tasks, DBMS have been optimized, e.g. for their use in data warehouse applications. One of these optimizations is the use of column-based storage layouts. These have become popular because they can achieve higher performance when data is required from only a small portion of columns for a query's execution, which is typical for analytical workloads [AMH08].

That leaves a gap in the workflow: semi-structured data that is available in an exchange format needs to be loaded into a database with a column-oriented storage engine. This requires either a manual conversion of the input data into an – often relational – model by the user or the database is able to accept the semi-structured data and the storage engine converts it automatically into a columnar layout which is than used for storage and query execution. The latter approach is obviously more convenient from the user's perspective, but also allows the storage engine to perform more optimizations as it gets more information on the original structure of the data. The conversion to a columnar layout can either be performed at runtime, depending on the queries being executed on the database (*dynamic approach*), e.g. by using adaptive vertical partitioning [SLC+19], or during the import of a dataset (*static approach*).

The Carbon ("Columnar Binary JSON") framework was created to implement the static approach [Pin19a]. Its longtime goal is to be used as a DBMS' storage engine for the scenarios described above. It currently provides tools to import JSON documents to its own, column-oriented storage format, convert it back to JSON or to execute analytical queries. But it still lacks some features of modern storage engines, such as transparent, lossless compression[1].

This technique aims to allow more data to be stored on the same hardware storage system and to accelerate queries that are limited by the I/O performance of the system [WKHM00]. The latter often applies to analytical queries, which makes it interesting for the usage in Carbon. However, some part of the data might be better compressible than others and which compression algorithm performs best depends on the input to be compressed. To adapt the compression to the data, many column-oriented storage engines like the ones used by Amazon Redshift [Amab] or Kudu [The19a] allow the compression to be manually configured on a per-column basis.

While this can work pretty well, it requires some deep understanding of databases, compression techniques and the data that needs to be stored. As the same problem exists throughout many other database topics like suitable index creation, partitioning etc., self-driving database technologies are one focus of research in the database community [Cam19] and recently became more mainstream, with large DBMS manufacturers massively promoting their systems to be self-driving and to run "autonomously" [Orab]. The users will therefore expect the DBMS to take care of such low-level details like applying suitable compression techniques automatically depending on their use cases.

[1] The basic compression framework already exists, but there is no working compression implemented so far [Pin19d].

Thesis Goals

This thesis' purpose is to transfer the approach of speeding up I/O-bound operations and saving storage space through lightweight compression from the relational data model to the columnar data model for semi-structured data as implemented in Carbon. While much research of the database community is dedicated to compressing numerical values in databases, all work in this thesis will be focused on strings, as they make up the largest part of the data in typical semi-structured datasets.

Reasearch Question
What is the effect of autonomous (textual) compression applied to a columnar semi-structured data model with respect to storage space and query performance?

Goals
In our thesis, we aim for the following:

(a) Setup of a powerful, yet small set of basic but representative compression techniques as building blocks

(b) Design a working technique for combination of different compression building blocks to construct new (higher-order) compression techniques towards a richer search space for self-driven compression. This especially includes exploitation of synergy effects, such as merging of redundant prefix tables.

(c) Construction and evaluation of a decision component for an optimizer that chooses the most reasonable compression technique, which may include higher-order compression techniques

To verify and evaluate the results of these analyses, the goal is to integrate the ideas in the Carbon framework and benchmark them using real-world datasets that origin from publicly available web APIs.

Thesis Structure

This thesis is structured as follows:

Chapter 2
The next chapter provides an overview of the Carbon framework, lossless data compression in general and a brief review of the most popular algorithms and their application in popular DBMS.

Chapter 3
The third chapter focuses on the individual algorithms that have been selected for the use in the Carbon framework as well as their implementation and integration. Additionally, the combination of the algorithms into one single, yet configurable compression algorithm and its key-based application is described.

Chapter 4
Chapter 4 discusses different approaches on how the algorithms from the previous chapter can be selected to perform best on the input data and how this process can be optimized in terms of runtime performance.

Chapter 5
The approaches and implementations from the previous two chapters are evaluated in chapter 5. The evaluation setup and procedure is described and the compression and runtime performance results of the individual components are presented and discussed.

Chapter 6
The last chapter summarizes the findings and compares it to the goals set in Chapter 1 and describes the effects of the use of compression techniques in the Carbon framework. It will also mention where further research may be required and what could be optimized in the implementations presented in this thesis.

Definitions

This section provides definitions for some terms that will be used throughout the whole thesis but may be defined differently by other researchers.

Definition 1 (Lossless data compression)**.** Lossless data compression *aims to reduce the size of a given uncompressed message by encoding it in a different way, preserving the same information. The result of this operation is a compressed message.*

Definition 2 (Decompression)**.** Decompression *is the process of restoring the uncompressed message from a given compressed message.*

Definition 3 (Compressor)**.** *A* compressor *is a function that takes an uncompressed message, performs a compression algorithm on that message and returns a compressed message.*

Definition 4 (Decompressor)**.** *A* decompressor *is a function that takes a compressed message, performs a decompression algorithm on that and returns the uncompressed message. For lossless data compression, the message restored is exactly the same message that has been compressed by the corresponding compressor.*

Definition 5 (Compression Ratio)**.** *The* compression ratio *describes the ratio between the size of a compressed message and its uncompressed size.*

$$compression\ ratio = \frac{size(compressed\ message)}{size(uncompressed\ message)}$$

A compression ratio less than one therefore indicates a reduction of the message in size through the compression process.

2 Background

This chapter will briefly describe the Carbon framework [Pin19a], present basic compression techniques and popular compression algorithms and finally reviews their use in various DBMS.

2.1 The Carbon Archive Format and Framework

In this section the goals and use cases of the Carbon framework and the Carbon archives are described first (Section 2.1.1). Then the archives' basic structure is explained in Section 2.1.2 as well as the format's differences to other serialization formats for semi-structured data (Section 2.1.3).

2.1.1 Goals and Use Cases of the Carbon Archive File Format

The Carbon archive file format is a binary file format to store columnar data and is developed, specified and maintained by Marcus Pinnecke [Pin19c]. It is designed to support read-intensive ("read-mostly" [Pin19b]) workloads, like analytical queries, on the data. The reference implementation, *libcarbon* [Pin19a], at the time of writing, is only capable of converting between JSON [Bra14] and Carbon archives. Therefore, a typical application would be to run analytical queries on large, static JSON datasets like the Microsoft Academic Graph [SSS+15] or dumps of REST-APIs [Ric07]. The main goal of the project, however, is to use the Carbon archive format as a storage backend for NoSQL in-memory databases [Pin19b].

Because of its use cases, even though the typical source of Carbon archive files are JSON files, it does not compete with JSON as it is not designed as a data exchange format. This is also true for various binary representations of JSON like BSON[Mon09]

or UBJSON[Kal], which are designed as exchange formats, not for analytical tasks, and are therefore record-oriented instead of column-oriented like Carbon archives. A more detailed comparison of these formats with Carbon archives can be found in Section 2.1.3.

In addition to the Carbon archive file format ("Carbon archives"), there is a separate, row-oriented Carbon file format. They have fundamentally different design goals, as the Carbon archives are designed to serve analytical workloads, whereas the Carbon (non-archive) files are designed to work as a key-value-store. Both may be used together to serve different needs of a database backend, but for this thesis, only the Carbon archives are considered because these are designed for analytical and therefore often more I/O-intensive tasks.

2.1.2 Basic Structure of a Carbon Archive File

A Carbon archive file is divided into two parts: the record table and the string table.

```
[
  {
    "name": "Bob",
    "age": 25,
    "height": 1.8
  },
  {
    "name": "Alice",
    "age": 26,
    "height": 1.7
  },
  {
    "name": "Mallory",
    "age": 24,
    "height": 1.75
  }
]
```

(a) Example columnar JSON document

String Table	
1001	name
1002	age
1003	height
1004	Bob
1005	Alice
1006	Mallory

Record Table
type: ObjectArray
colname: 1001 type: TextArray values: [1004, 1005, 1006]
colname: 1002 type: Int8Array values: [25, 26, 24]
colname: 1003 type: FloatArray values: [1.8, 1.7, 1.75

(b) Simplified Carbon archive structure

Figure 2.1: Example of a columnar JSON document and its corresponding Carbon archive representation

The record table stores all properties grouped by JSON object key (column) and data type – and therefore column-wise – as research has shown that column-oriented storage systems are better suited for analytical workloads than classic row-oriented designs (e.g. [AMH08], [IGN+12]). The whole record table only contains fixed-length values for fast processing and iteration. Variable-length values (e.g. strings) are represented as fixed-length references to the string table [Pin19b]. As for the typical use cases mentioned in Section 2.1.1, the string values take up most of the space, so the record table is typically only a small part of the whole file and therefore should fit into main memory.

The string table maps the record table's string references to their actual values [Pin19b]. It does not contain duplicates, meaning that the strings are deduplicated and identical strings are represented by the same reference in all parts of the record table. For datasets containing a lot of identical strings, this encoding can already reduce the size of the Carbon archive significantly. As the entries are stored as tuples of the form $< reference, value >$ (additional meta data is left out here for simplification), the order of the entries can be changed without breaking the lookup of the strings in the table.

The file format in general stores an indicator on how the string table is compressed. It also provides a flexible framework to implement compressors. In the beginning of this thesis, the only options implemented were *no compression*, which just stores the strings as they are, and *plain huffman coding* [Huf52], but the latter was designed to be a demonstration of the compression framework and is not ready to be used in production [Pin19d].

Due to the strict separation of the two parts of the file, when used as a database storage backend as described in Section 2.1.1, the record table can be loaded into the memory and the strings are only fetched from the disk when needed. This design allows many operations to be performed with very little disk access, even when the file itself exceeds the main memory size.

2.1.3 Differences to other Serialization Formats

There are specifications other than Carbon, defining how semi-structured documents can be encoded in an efficient way. However, they differ in their design goals and therefore in the resulting properties.

- **BSON**
  While Carbon performs a transformation into a columnar storage layout, BSON [Mon09], as used by MongoDB [Mon], is just a binary encoding of the same document structure. It extends the JSON type specification by more specialized types like timestamps and JavaScript code blocks, while Carbon extends the JSON types only by the *(custom) binary data* type and specialized numerical ones. BSON's design goals, according to its specification, were to create a lightweight (space-efficient) format, that is easily traversable and fast to encode and decode. It was not designed to perform analytical queries on the data.

- **UBJSON**
  UBJSON [Kal] is designed as a universal, space-efficient, easy and fast to parse representation of JSON documents. It supports the same types as JSON, however, it differentiates between different numerical types (e.g. signed/unsigned,

integers/float). It shares these properties with Carbon. Another commonality is them both being marker-based formats and using strongly-typed arrays. But in comparison to Carbon, UBJSON lacks support for custom binary fields and does not store variable-length values separated from fixed-length ones. A more detailed comparison is available the Carbon at specification page [Pin].

- **Smile**
  The Smile [Fas13] format originates from the Jackson project [Fas] and is designed to be fully JSON compatible but very space-efficient [Fas17]. Efficient traversal of the document is explicitly listed as a "non-goal", as the Jackson Project provides only sequential APIs for all of its backends. Smile keeps the document structure as it is (as opposed to Carbon), but uses different compression techniques to reduce the amount of required space. Like Carbon, it allows deduplication of identical strings (called "shared strings" in the specification), but with two major differences: Firstly, no explicit dictionary is set up, instead, back-referencing is used. Secondly, the feature is completely optional.

There are many other formats, e.g. CBOR [BH13] and MessagePack [Fur08], but they all aim to serve as an alternative to JSON by representing the data in the same way without any transformation to a columnar layout.

2.2 Lossless Data Compression Algorithms

This section starts in Section 2.2.1 with the description of basic (non-database-specific) data compression algorithms which are still the building blocks of most modern general purpose compressors that are presented afterwards in Section 2.2.2. Finally, in Section 2.2.3, the most common data type-specific encoding schemes that aim to either support compression or that already compress the values on their own will be discussed as a bridge to Section 2.3.

2.2.1 Basic Data Compression Algorithms

Data compression algorithms try to reduce redundant information in a given input, often called message. That can be either done by replacing sequences of symbols (recurring patterns) or by representing the symbols using less space.

The most simple implementation of the first approach is *run length encoding*, where consecutively repeated values are replaced by the value and the number of its occurrences, e.g. "aaabbbb" would be represented by "3a4b". Run length encoding is fairly

limited for compression of natural language text, but it is suitable for some special applications and often combined with other compressors.

To encode recurring patterns that contain multiple characters (as opposed to run length encoding), *dictionary encoding* can be used, where frequently occurring sequences are stored in a separate space, e.g. at the beginning of the file, as a so called "dictionary" that maps these sequences to shorter replacements like integers from one to the number of entries stored in the dictionary. The message is then kept as it is, except for the sequences defined in the dictionary. They get replaced by their shorter replacements which leads to a more space-efficient representation. An in most cases even more space-saving and therefore in text compression more widely used alternative to dictionary coding, is *back-referencing*. Instead of separately storing a dictionary, it uses references within the text. The first time the algorithm encounters a sequence, it is kept as it is. When detecting the sequence a second time, it inserts a marker, referencing back to the starting position where it has seen the text before and the length of the text to replace. A major challenge of this approach is to efficiently find these recurring sequences and to encode the back-referencing markers in a space-efficient manner. Solutions to these challenges are described in [ZL77] and [RPE81].

To represent symbols in a more compact form, entropy encoders do not store them as fixed-length (e.g. 8 bit) symbols. Instead, in *Huffman coding*, they are represented by a variable-length code per symbol. Huffman coding uses a prefix code and assigns the codes depending on the probability of the symbols' occurrences with frequently occurring (more probable) symbols getting assigned shorter codes than less probable ones. The exact method of creating the codes is described in Huffman's paper [Huf52].

Another approach is used by *arithmetic coding* [RL79], which assigns one code to the whole message or block by recursively subdividing the interval from zero to one n times (with n being the length of the message/block) into ranges that are proportionally sized to their represented symbols' frequencies. The resulting interval is the one that is contained in all intervals that correspond to the symbol to be encoded at each level. That interval is then encoded in binary form. In terms of compression ratio (see Definition 5: Compression Ratio), arithmetic coding is at least as good as Huffman coding and better in some cases [WNC87]. It can represent the data more compact compared to Huffman coding if the probability of some symbols in the message is very high. The reason is that Huffman replaces one symbol in the original message with one variable-length code in the output and this code can obviously not be shorter than one bit, while arithmetic coding assigns one code to a whole chunk of the input message and can therefore achieve a representation with less than one bit per symbol on average. For both, Huffman and

B	A	N	A	N	A
A	N	A	N	A	B
N	A	N	A	B	A
A	N	A	B	A	N
N	A	B	A	N	A
A	B	A	N	A	N

(a) Initial BWT matrix

A	B	A	N	A	**N**
A	N	A	B	A	**N**
A	N	A	N	A	**B**
B	A	N	A	N	**A**
N	A	B	A	N	**A**
N	A	N	A	B	**A**

(b) Sorted BWT matrix

Symbol	Alphabet	Index
N	ABN	**2**
N	NAB	**0**
B	NAB	**2**
A	BNA	**2**
A	ABN	**0**
A	ABN	**0**

(c) MTF on BWT output

Figure 2.2: Burrows-Wheeler-Transform and Move-to-front coding applied to the message "BANANA"

arithmetic coding, variations with dynamic updates of the symbols' probabilities exist [Knu85].

As arithmetic coding was partly covered by patents for some time, *range coding* [Mar79] is often used instead which basically follows the same principle. Because both involve CPU-expensive division operations, *finite state entropy coding* [Col13] was developed which is based on the asymmetric numerical systems theory [Dud13] and achieves the same compression ratios as arithmetic coding but only with additions, shifts and bit-mask operations, which can be executed significantly faster by CPUs [LR81].

A completely different approach that is not compressing data itself but transforms the data into a better-compressible form for the algorithms described above, is *Burrows-Wheeler-Transformation* [BW94] (or *BWT* for short) in combination with *move-to-front coding* (or MTF) [BSTW86]. The BWT is a reversible transformation that tries to group the same characters of a message closely together. For a message of the length n, a $n \times n$ matrix is set up. The first row contains the symbols of the message in their original order and every following line is (circularly) shifted left by one symbol. The lines are then sorted in lexicographical order. The output of this method is the last column of the matrix (output message) and the position of the first occurrence of the output string's first symbol in the original message. BWT is demonstrated on the example message "BANANA" in Figure 2.2a and Figure 2.2b. The result is $< NNBAAA, 3 >$.

As the output string of the transform has the same length as the input message, there is no space-saving achieved by applying BWT itself. In fact, as the index of the first symbol of the input has to be stored in addition to the message, the output is even larger than the input. But due to the grouping of the symbols by BWT, MTF can be applied effectively: MTF uses a dynamic alphabet initialized with the symbols of the original message in lexicographical order. Whenever the algorithm encounters a symbol,

its current index in the alphabet is written instead of the symbol. The alphabet is then updated by removing the character from its current position and moving it to the first place of the alphabet. As BWT has grouped the same characters closely together, the emitted indexes tend to be small, which means that in the output message (the indexes) the chance of encountering low values is higher. That can be exploited by arithmetic or Huffman coding. Also, when the same characters were grouped directly together, the output will contain sequences of zeros which can be encoded using run length encoding. This behavior can also be observed when applying MTF to the "BANANA" example from above which results in $< 202200, 3 >$, as shown by Figure 2.2c. In general, the longer the message is that is encoded with BWT and MTF, the more effective the algorithm becomes.

2.2.2 Modern Data Compression Algorithms

The modern compression algorithms are mainly based on variations and combinations of the algorithms explained in Section 2.2.1. The most common algorithms will be described briefly ordered by their release date.

2.2.2.1 Deflate

The deflate algorithm was originally developed by Phil Katz in 1989 for the ZIP file format [PKW19]. The algorithm as well as the bitstream format were standardized in 1996 via IETF RFC 1951 [Deu96a]. It combines back-referencing within a window of at most 32 KiB with Huffman coding. For most implementations, a compression level can be specified that influences how much effort is spent on finding the longest matching string to reference. That allows a user-defined trade-off between compression speed and ratio.

The popular *zlib* [lGA06] library implements the same-named format which is just the deflate bitstream with two bytes of header and four bytes of checksum added. The *gzip* format [Deu96b] also embeds the deflate bitstream.

The popularity of deflate today is more likely due to it being one of the first standardized algorithms/bitstream formats as technically there are algorithms like Zstandard [CK18] (see below) that are performing better in nearly every benchmark (e.g. [SB18]) with respect to all important criteria: compression ratio, compression and decompression speed. Applications of deflate include lossless images (PNG [Bou97]), network data (HTTP with GZIP compression [Res15]), documents (PDF [Int17a]) and much more.

2.2.2.2 bzip2

The bzip2 compressor was created by Julian Seward in 1996. It splits the data to be compressed into blocks and then applies the Burrows-Wheeler-Transformation, move-

to-front coding, run length encoding and Huffman to it (see Section 2.2.1). bzip2 is the successor of bzip, whose development had stopped because of patent issues due to its use of arithmetic coding.

In most benchmarks, bzip2's compression ratio is significantly better than deflate's (e.g. gzip), but worse than LZMA's (see below) while the runtime performance of all three behaves inverted to the compression ratio ([Her16], [SB18]). The original bzip2 compressor provides a compression level setting that influences the size of the blocks in which the input data is split. As mentioned in Section 2.2.1, larger blocks to which BWT is applied to usually lead to better compressible outputs and therefore improved compression. The influence on the runtime is measurable yet not very high, but the memory consumption increases noticeably with higher compression levels.

bzip2 is mainly used to compress single files or tar archives [Fre] in Linux/UNIX environments.

2.2.2.3 LZMA

The Lempel-Ziv-Markov-Algorithm (LZMA), developed by Igor Pavlov since 1998, can be seen as an enhanced version of the deflate algorithm. It uses back-referencing (a LZ77 [ZL77] derivation), range coding and Markov chains to more accurately estimate the symbols' probabilities as their context (previous symbols) is respected. Another change compared to deflate that can lead to better compression ratios is the increased sliding window size of up to 4 GiB. These changes combined make LZMA one of the best-compressing algorithms regarding compression ratio ([Her16], [SB18]). In terms of runtime, the algorithm is slower than many others but very asymmetric as decompression can be up to 20 times as fast as the compression process. That property is useful for scenarios where data is only compressed once and distributed to many clients that decompress it. Like for deflate compressors, a compression level setting influencing the back-reference search is often implemented to empower the user to trade compression ratio for runtime performance.

There is no official specification of the LZMA bitstream and algorithm, but a draft version by Pavlov exists [Pav15]. In Linux/UNIX operating systems, LZMA is often used with the *xz* file format, for which a specification is available [Tuk09].

LZMA's main application is in archive formats like 7z [Pav], for which it was originally designed. Due to its asymmetric runtime behavior and good compression, one important application is the compression of operating system packages, e.g. for Arch Linux (pacman [Arc]) and Slackware (slapt-get [Woo]), as well as the distribution of source code packages.

2.2.2.4 LZ4

The LZ4 [Cola] compressor was developed by Yann Collet in 2011 and is designed for very high compression and decompression speed. It is based on LZ77 (back-referencing), using a very simple bitstream format [Colb]. It compresses $\approx 5 - 8\times$ as fast as gzip (at gzip's lowest compression level) and decompresses $\approx 10\times$ as fast [Her16]. The compression ratio is worse than gzip's/deflate's in common scenarios but there exists a mode called LZ4_HC ("LZ4 High Compression"). In this mode, LZ4 achieves similar compression ratios as deflate but at lower compression speed, while the decompression speed does not change compared to standard LZ4.

Due to its high speed (or low computational resource requirements), its main application is in I/O performance enhancement, e.g. as a transparent compressor for databases (see Section 2.3) or in filesystems like ZFS [Ric] or SquashFS [Lou13].

The snappy [Goob] compressor was released at about the same time and shares its focus and basic design. As LZ4 provides higher compression/decompression speed at nearly the exact same compression ratio, snappy will not be covered separately.

2.2.2.5 Brotli

The Brotli compressor and bitstream format [AS16] were developed in 2013 by Jyrki Alakuijala and Zoltán Szabadka. It uses a derivation of LZ77 algorithm (back-referencing) and Huffman coding that respects, like LZMA, the context of the symbols for a better estimation of the symbols' probabilities. Additionally, due to some improvements to the data format, some space-savings were achieved. The original Brotli (C-)implementation [Gooa] supports compression levels from 1 to 11 that have significant impact on the compression ratio and runtime performance. In the authors' benchmark [AKSV15], when using compression level 1, it is slightly faster than zlib's deflate implementation while still beating it in terms of compression ratio. When using the highest level (11), it achieves a slightly better compression ratio than LZMA while being significantly slower ($\approx 8\times$) than LZMA and approximately 200 times slower than compression level 1 – but only for compression, decompression only drops by $\approx$ 10 to 20%. Therefore, the algorithm is very scalable.

It was originally developed for the Web Open Font Format 2.0 [LL18]. With the growing support for Brotli in webservers and browsers, it is mainly used as HTTP encoding (instead of gzip/deflate) to deliver static web content like CSS frameworks or JavaScript libraries.

2.2.2.6 Zstandard

Another compression algorithm developed by Yann Collet is the 2015 released Zstandard [CK18] (often called "zstd"). While LZ4 was focussed on high runtime performance

only, Zstandard was designed for deflate-comparable compression ratios at higher compression and decompression speeds. Like LZ4 and deflate, it is based on LZ77 but with a large window for back-reference search (up to 2 GiB vs. deflate's 32 KiB). For entropy coding it uses Huffman coding for literals and finite state entropy (FSE) coding for sequences. A unique feature is the ability to work with previously created dictionaries, which is useful when many small standalone messages (e.g. files) usually containing similar content have to be exchanged over the network, because there might be no recurring patterns within one message but across the messages.

As FSE is significantly faster than arithmetic coding and the whole algorithm was designed for modern CPU (mostly branchless, optimized for out-of-order execution), the compression/decompression runtime performance is much higher than deflate's [SB18]. Like most other compressors, the reference implementation [Fac] allows to specify a compression level (fastest: -5, best-compressing: 22) and has a great influence on the runtime performance and compression ratio, making it one of the most scalable compression algorithms.

With Zstandard being officially standardized by RFC 8478 as a HTTP content encoding scheme, it could be used for compressing HTTP payloads. However, at the time of writing, none of the major browsers (Microsoft Edge, Mozilla Firefox, Google Chrome) support it. Its compression/decompression speed also makes it suitable for transparent file system compression like in BTRFS [Lina] and for database applications (see Section 2.3).

2.2.3 Compressing and Compression-supporting Encoding Schemes for Same-Type Value Sequences

While the algorithms presented in Section 2.2.2 are designed to compress text or binary data without knowing its structure, this part will explain methods that help to compress lists of same-typed values, e.g. entries of a column in a database. Not all approaches described here are applicable to all data types and some do not even compress the data but transform it, so general purpose compression algorithms can compress them well.

One approach already described in Section 2.2.1 to encode numerical values is *run length encoding*. Given an ordered list of values, identical, consecutive entries will be replaced by the number of consecutive occurrences and the value itself (but only once). For example, this can be useful when importing sales containing a date field into a data warehouse: all sales of the same day will likely be stored sequentially, so 100.000 dates of sales of the same day could be represented by `<100000, "2019-05-04">` instead of 100.000 values. When the order of the entries does not matter (which is rarely the case

in database scenarios), then sorting the values before applying run length encoding can dramatically increase the compression ratio.

It might not often be the case that the consecutive values are identical, but they might be close together. In a sales table, storing an order number that increases by one with every row would not profit from run length encoding in any way. In this case, however, *delta coding* could be applied: instead of storing the absolute values, only the difference to its predecessor is stored (except for the very first value). In the order number example, all values following the first one would be one, as the order number always only increases by one. A list of ones could then be compressed very well using run length encoding. If the differences are not always constant but in a limited range, then compressing the values using Huffman coding or similar approaches can reduce the amount of space required. Just like for run length encoding, if there is a possibility to sort the values before, that could greatly improve the compression ratio. One major drawback of delta coding is that to decode a value, all previous values have to be decoded first. That can be limited if not just the first entry is stored as an absolute value but every n-th one, too. In that case, on average $\frac{n}{2}$ entries have to be read to decode an entry at a randomly chosen position. Also, for many analytical workloads, many consecutive entries have to be read, such that the predecessor values are known anyway.

A variation of delta coding is *frame of reference coding* [GRS98] (FOR). Instead of storing the difference to the direct predecessor, a reference value is selected for a chunk of data and all entries within that chunk are stored relative to that reference value. This value can be either an existing entry or an additionally stored one. This technique is useful when the data cannot be sorted because the order of the entries has to be preserved but the values only vary within a certain range. Like for delta coding, this results in smaller values that can be stored in a more compact form. The *patched frame of reference* variation is more resistant against outliers as values that differ from the reference value more than a given threshold are encoded as absolute values instead of relative to the reference value [ZHNB06].

To enable better compression of numerical values using general purpose compression algorithms, the *bitshuffle* algorithm can be applied. Instead of storing the bits for the first value, then for the second and so on, the bits are stored according to their position: first all most significant bits are written for all values of the block, then the second-most significant and so on. This does not compress the data, but this representation can be compressed much better, e.g. by LZ4 (see Section 2.2.2), when the values only vary by a small amount (see example at Figure 2.3).

Decimal	2^4	2^3	2^2	2^1	2^0
6	0	0	1	1	0
5	0	0	1	0	1
7	0	0	1	1	1
4	0	0	1	0	0

Figure 2.3: Bitshuffle example: Binary representation of the sample values $\{6, 5, 7, 4\}$ using 5 bit each. Reading the table top to bottom, left to right, leads to eight zeros in a row, followed by five ones, which makes them easily compressible.

The encodings discussed so far were all related to numerical values. To store repeated values of any type (e.g. strings), *dictionary encoding* can be applied. This is pretty similar to the dictionary encoding for text compression described in Section 2.2.1: all distinct values are stored in a dictionary that maps them to a (shorter) replacement, e.g. integers. To encode a list of values, the replacements instead of the original values are stored. For long, repeated values, this encoding can lead to significant space-savings.

A special encoding for string types is *front coding* [WMB99] (also called *prefix encoding* or *incremental encoding*). Like delta coding, it encodes a value relative to its predecessor: for each value to encode, it stores the length of the common prefix with its predecessor followed by the remaining part of the string (see Figure 2.4 for an example). The same restrictions and solutions as for delta coding apply to them : to decode a value, all values that share at least one symbol in the prefix have to be decoded. This problem can be solved by storing a full value (with no common prefix) every n entries. The effectiveness of front coding can be increased if the values can be sorted before applying front coding.

Original Value	Common Prefix Length	Remaining Value
`https://www.google.com`	0	`https://www.google.com`
`https://www.github.com`	13	`ithub.com`
`https://www.ovgu.de`	12	`ovgu.de`
`https://en.wikipedia.org`	8	`en.wikipedia.org`

Figure 2.4: An example of encoding URLs using front coding. The encoded version could be stored using 58 bytes instead of 87 (saving $\approx 33\%$).

If there are many values that share common prefixes in a list, but due to their order, none or not many of them occur consecutively and the order cannot be changed, front coding will not provide good compression. A solution to that problem can be a variation of dictionary encoding: common prefixes are stored in a dictionary and whenever such

a prefix occurs, the entry in the dictionary is referenced, followed by the remaining value (see Figure 2.5 for an example). With more entries encoded, the effectiveness will increase as the values in the dictionary only have to be saved once. This encoding has no prevalent name, but for rest of this thesis, it will be called *prefix dictionary coding*.

Prefix	ID
`https://www.`	1
`http://www.`	2

(a) Dictionary for URLs

Value	Prefix ID	Rem. Value
`https://www.google.com`	1	`google.com`
`http://www.mit.edu`	0	`mit.edu`
`https://www.ovgu.de`	1	`ovgu.de`
`http://www.apache.org`	0	`apache.org`

(b) Encoded URLs using the dictionary of Figure 2.5a

Figure 2.5: An example of encoding URLs using prefix dictionary coding. The encoded version could be stored using 63 bytes instead of 80 (saving $\approx 21\%$).

2.3 Compression in Databases

This section explains the motivation (Section 2.3.1) and requirements (Section 2.3.2) for data compression in databases and describes the two main approaches of how it is implemented in DBMS in Section 2.3.3 and Section 2.3.4. It closes with an overview of other applications of data compression in DBMS in Section 2.3.5.

2.3.1 Motivation for Compression in Databases

The reason for the application of data compression techniques in general is to reduce the amount of data to be either transferred or to be stored. This can have several effects.

On the one hand, as databases may grow very large, storage costs can become an important economic factor which makes data compression attractive for economic reasons.

On the other hand, query execution performance is very important. While compression algorithms always introduce some computational overhead, it is worth analyzing what the real bottlenecks of database management systems are. As summarized in Table 2.1, data that is only present in main memory, but not in the CPU cache, takes significantly longer to load, both in terms of latency and throughput (bandwidth). This effect is even more conspicuous if the data has to be fetched from the SSD or HDD storage, which adds some orders of magnitude in latency and bandwidth, depending on the storage technology used. As the faster memory systems, that are closer to the CPU, have less capacity, compression can help keeping more data close to the CPU, e.g. in main

memory, and therefore reduce the number of HDD/SSD accesses required [RHS95]. Back in 2000, Westermann et al. demonstrated performance gains of up to 55% for I/O-intensive queries of the TPC-D benchmark [WKHM00]. While storage systems have improved a lot since then, many database operations today are I/O-bound and compression can accelerate them. Even with the rise of main memory database systems, data compression can still reduce the query execution time as many operations are memory-bound [Bro19].

Memory	Latency	Bandwidth	Capacity	Price (EUR per GiB)
L1 Cache	$1ns$	$1\ TB/s$	$32\ KB$	n.a. standalone
L2 Cache	$4ns$	$1\ TB/s$	$256\ KB$	n.a. standalone
L3 Cache	$40ns$	$> 400\ GB/s$	$8 - 32\ MB$	n.a. standalone
Main Memory	$80ns$	$100\ GB/s$	$1 - 1024\ GB$	≈ 4
NVMe SSD	$90\mu s$	$3\ GB/s$	$256 - 2048\ GB$	≈ 0.15
HDD	$4ms$	$210\ MB/s$	$500 - 8000\ GB$	≈ 0.03

Table 2.1: Memory and storage latencies and bandwidth according to [Bre16], [XSG+15] and [Sea16]. All numbers reflect the state of technologies as of 2015/2016 for better comparability. "Price" reflects the approximated market price as of 2019.

To sum this section up, data compression can positively affect two parameters of a database system: storage consumption and query performance. It allows to trade computing power for data access time to make best use of the hardware.

2.3.2 Requirements for Data Compression in Databases

As the motivation for the use of compression described in Section 2.3.1 has two aspects, namely storage savings and query performance enhancement, the requirements resulting from each will be discussed separately in this section. Afterwards, the relationship between the requirements will be discussed.

When focussing on storage savings, the main aspect for data compression algorithms to consider is optimizing the compression ratio (see Definition 5: Compression Ratio). In most cases that means trading more computation for less storage, but this does not scale very well: in an examplatory benchmark, improving the compression ratio by $\approx 16\%$ increases the time required for the compression by 770%[1] [Her16]. From a storage optimization perspective, requiring 16% less space may be worth the computational overhead.

[1] *xz* compressor with default options: $r = 0.184$, compression time: $32.2s$; *xz* compressor with "extreme" options: $r = 0.155$, compression time: $280s$

With respect to query performance, the requirements for data compression algorithms are more complex to define. Ideally, during query execution, all available resources are used equally, so the system is neither I/O-bound nor compute-bound. As explained in Section 2.3.1, data compression allows the system to trade computational resources for I/O and can therefore be used as a tool to balance them. While the compression could be done in real-time or when the system is not under high load (e.g. through table reorganizations [IBM]), decompression has to be performed in real-time, so lightweight (de-)compression algorithms need to be used. But it is not possible to define in general, how much computational effort should be spent on data compression to speed up the data access, as it heavily depends on the available hardware resources and the workloads (queries). On a system with a slow HDD and a fast CPU, reducing the amount of data to be fetched from the disk might be worth spending more compute power on data compression than on a system with a slower CPU but a fast SSD as its primary data storage. Also, for queries including heavy computations, the database system might be already compute-bound without any data compression algorithms involved, so their application would slow down the system. On the other hand, a query that has to sum up a certain value for billions of records will most likely be I/O-bound and data compression may help loading more records using the same I/O resources in the same time which would result in lower query execution times. All these considerations lead to the requirement that the data compression of a database system has to be configurable with respect to the consumption of computational resources.

Regarding recent research on self-driving databases (e.g. as summarized by Campero [Cam19]) the latter configuration could be done by the database management system itself, e.g. by automatically analyzing the bottlenecks during the most frequent queries and then tune the compression settings for them. However, it cannot decide how important storage savings are to the administrator compared to system performance, as that is mostly an economic decision.

For that reason, we can conclude that a lightweight data compression algorithm that can be configured by the administrator of the database system according to his or her own preferences is required.

2.3.3 Page-based and File-based Compression

Although most database management systems basically share the five-layer-architecture [Hä05] (see Figure 2.6), they have come up with different strategies on how to integrate compression into their architecture: it is often either performed at page-level or at file-level. The reason why it is often not integrated on higher levels of abstraction is most certainly because it would be a less generic solution, as it would have to be implemented

on top of the record logic for different data types. Because of that, implementing it on the lower levels of the architecture can result in a cleaner code base. The drawback is that some structural information and meta data is not available that might be exploited by the data compression algorithms. The lack of this database-specific information results in the usage of non-database-specific, general purpose compression algorithms, as the examples below will show. For that reason, some vendors have developed more specific solutions that are presented in Section 2.3.4.

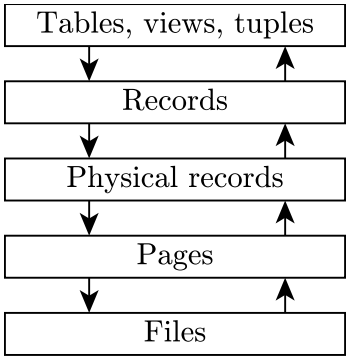


Figure 2.6: Objects in (R)DBMS mapping hierarchy, adapted from [Hä05]. While the uppermost layer is specific to relational systems, the rest of this architecture also applies to other types of DBMS.

Relational Database Management Systems

The following list of open source and commercial RDBMS products is not exhaustive, but contains the most popular ones and also represents the variety of possible approaches to page- and file-based compression:

- **MySQL/MariaDB with InnoDB storage engine**
  MySQL and MariaDB (a fork of MySQL) support two mechanisms for page compression. The legacy one, although called "compressed row format", is not operating at physical record-level, but at page-level. It compresses the data using a derived LZ77 algorithm [ZL77] (zlib [lGA06]) while keeping the compressed and the uncompressed page in memory. Which of them is evicted first is decided depending on the workload. For I/O-bound operations, it starts with the eviction of the uncompressed pages, for CPU-bound operations, it evicts the compressed pages first. To avoid re-compressing a page for every small change, an uncompressed "modification log" is kept and only if that grows over a defined limit, the logged changes are applied and the page gets re-compressed [Ora18a].
  The new page compression, also present in both systems, is called "InnoDB page compression". The approach differs in two aspects: only uncompressed pages are kept in memory and more compression algorithms are supported (zlib [lGA06],

LZO [Obe], LZ4 [Cola], LZMA2/XZ [Tuk09], bzip2 [Sew19] and snappy [Goob]). It is also referred to as "transparent page compression" in the documentation.

- **PostgreSQL**
  The PostgreSQL DBMS does not support page compression, but there are commercial distributions that have it on their roadmap [Pos]. Instead of page compression, tablespaces can be created on file systems that offer transparent compression (like ZFS or BTRFS) [Ste13] and therefore "support" some kind of file-level compression. Other techniques supported by PostgreSQL are described in Section 2.3.4.

- **Microsoft SQL Server**
  While Microsoft SQL Server offers a feature called "page compression" [Mic16a], it actually consists of two passes of compression where only the second one is a generic, page-level compression. Details of the algorithm are not public, but the documentation states that a dictionary-based compression is used.

- **Oracle Database**
  From Oracle 9i Release 2 onwards, "block compression" (a block in Oracle terminology is comparable to a page in other DBMS) is supported for bulk load operations and from Oracle 11g onwards, it is also available for all other operations. Details of the algorithm are not public, except that it is dictionary-based. Like for MySQL and MariaDB, updates to pages do not immediately trigger a re-compression, but they are logged uncompressed until a certain threshold is exceeded [Oraa].

NoSQL Database Management Systems

NoSQL DBMS also offer various compression methods. Due to some peculiarities regarding their storage backends, these can be implemented much easier by some of them:

- **MongoDB with WiredTiger storage engine**
  The WiredTiger storage engine used in current MongoDB releases supports "block compression" [Mon18c] (like for Oracle Database, "blocks" are the same as pages) using snappy [Goob] and zlib [lGA06] compression algorithms.

- **Apache Cassandra**
  Apache Cassandra supports the compression of "chunks" (which are again comparable to pages) of a user-configurable size. As the tables are immutable/append-only, complex update mechanisms like in MySQL/MariaDB or Oracle Database are not needed [The16]. When reading the "chunks", only the uncompressed data

is passed to the higher levels of the DBMS (as opposed to the legacy page compression in MySQL/MariaDB). As compression algorithms, either one of the prepackaged algorithms (LZ4 [Cola], snappy [Goob], deflate [Deu96a] or Zstandard [CK18]) can be selected or even a user-defined implementation can be supplied.

- **CouchDB**
  CouchDB implements compression at file-level. As it uses an append-only approach like Apache Cassandra, update/re-compression mechanisms are not required. It supports snappy [Goob] and deflate [Deu96a].

While these DBMS, both relational and NoSQL, have been selected to represent different approaches, most other DBMS also support some kind of page-level or file-level compression.

2.3.4 Compression at Physical Record-level

While the algorithms described in Section 2.3.3 can help reducing a database's size, they do not take specific knowledge of the structure of the data into account (e.g. data types). Operating directly on the values knowing the data-type allows for more efficient implementations of compression – both in terms of compression ratio and runtime performance. That is why some vendors implement data type-specific compression in their systems.

One common pattern is the compression of textual data that exceeds a defined threshold. For example, in PostgreSQL that technique is called "TOAST" ("The Oversized Attribute Storage Technique") and originally resulted from limitations of the internal design of the system that would not allow to store attributes larger than the system's page size [The19c]. It uses a custom, LZ77-like [ZL77] algorithm with some optimizations from LZ4 [Cola], called "PGLZ". Oracle Database offers a similar approach, called "Advanced LOB Compression" [Chr18], for its "Large Object" types (also called "SecureFiles") that is meant to store documents, images and other files in the database. As for other Oracle compression techniques, details of the algorithm are not public. The documentation states that the compression is only applied when the data to be stored would actually benefit from the compression. Additionally, Oracle Database offers deduplication when the exact same value (e.g. image) occurs more than once in a column.

Microsoft SQL Server's "page compression" feature is, as mentioned in Section 2.3.3, actually not just a generic page compression but performs two passes of compression per page. In the first pass, for every column, prefix compression is applied to the

physical records [Mic16a]. The second pass is described in Section 2.3.3. Additionally, Microsoft SQL Server also supports a data type-specific compression feature called "row compression". Examples include storing small integer values using only one byte while the column's data type is actually a fixed-length multi-byte integer and removing trailing padding from fixed-length binary/char column values (and therefore storing them as variable-length values). Aside from that, the "row compression" feature does not include any text compression. The documentation provides a comprehensive list of how the different data types are affected [Mic16b].

Many databases, not just NoSQL ones, support storing and processing JSON [Bra14]. This support has finally been standardized in SQL:2016 (ISO/IEC TR 19075-6:2017(E) [Int17b]). While it may be easy to read and write for humans, JSON was never designed as a storage format for documents, as it is neither space-efficient nor fast to parse or to traverse. To reduce the amount of required space and to increase query performance, some vendors do not store JSON in its original, plain-text form. MongoDB converts the data to a binary representation called BSON [Mon]. Similarly, PostgreSQL offers a data type called "jsonb" [The19d], that also uses a binary representation of the data. However, both are not designed to be space-efficient, though, but to allow indexing and to be fast to process, including efficient traversal of the document. To overcome the space-efficiency issue in PostgreSQL, the ZSON extension [AKS16] can be enabled. It applies a dictionary-based compression to the document keys and string values.

On column-oriented data storage, data compression can be even more effective than on row stores as data from the same column (and thereby most likely from the same real-world domain and same data type) is stored consecutively. That increases the chance of recurring patterns, which is the basis for many data compression algorithms. Apache Kudu offers different column encoding schemes depending on the column's data type as shown in Table 2.2.

Data Type	Run length	Bitshuffle	Prefix	Dictionary	Default
integer types	✓	✓			Bitshuffle
floating point types		✓			Bitshuffle
decimal		✓			Bitshuffle
boolean	✓				Run length
string types			✓	✓	Dictionary

Table 2.2: Apache Kudu column encoding schemes, according to [The19a]

Amazon's data warehouse service Redshift, which is based on an older PostgreSQL version (8.0) with a column-oriented storage engine [Amab], provides some different

column encoding schemes [Amac] (while also supporting dictionary and run length encoding): delta-coding and "mostly encoding", which is a variation of variable-length integer coding [Amaa].

Apache Kudu and Amazon Redshift both additionally apply a general purpose compressor like Zstandard [CK18] or LZ4 [Cola] at page-level.

2.3.5 Other Applications of Data Compression in DBMS

The application of compression is not limited to the actual data storage of DBMS. Some auxiliary data structures are stored without passing the architecture presented and visualized in Figure 2.6, with the journal (also called "write ahead log") being the most prominent one. As the speed at which these can be written also has a major impact on the systems overall (write) performance, the motivation for data compression is basically the same as described in Section 2.3.1. For example, PostgreSQL [The19b], MariaDB [Mar] and MongoDB [Mon18a] all support these techniques, with MongoDB having it enabled by default (with the snappy compression algorithm [Goob]). As many systems use the journal for replication purposes, a compressed journal can increase the replication speed.

Another application of compression concerns the communication of the client with the DBMS server to reduce the amount of data that needs to be sent over the network. As most DBMS systems allow secure connections via TLS, the TLS protocol's own compression mechanism [Hol04] could be used, but that is not recommended because since the "CRIME" attack [RD12] it has to be considered insecure and was therefore removed in TLS version 1.3. MySQL supports connection compression via Zstandard [CK18] and zlib [lGA06] in recent versions, although it is disabled by default [Ora18b]. The PostgreSQL community is discussing the introduction of compression in the next version of the PostgreSQL wire protocol [Urb14]. Also, MongoDB's use of BSON in the wire protocol [Mon18b] can be seen as some sort of compression, although it was never intended to be very space-efficient.

3 Carbon Archive Compression

In Section 2.1 the Carbon archive format and framework was described. While it provides an interface for implementing compression of the string table, there was no fully-working compression implemented at the time of writing. This chapter presents how the compression framework was extended in preparation of Chapter 4, both in terms of its general structure and in terms of the supported compression algorithms.

3.1 The Carbon Compression Interface

The existing Carbon framework already contains a lean interface that allows to implement compressors (in more recent versions also called "packers") for the string table (see Section 2.1.2). It has no assumptions on how the compression algorithms themselves work, but it defines the order in which the functions are called (inversion of control) and the structure in which the compressed data as well as additional meta data generated by the compressor (like Huffman code tables) is stored (see Figure 3.1).

The procedure for serializing the string table is as follows: When writing the table, the framework will first instantiate the compressor and request it to write the *meta data section*. It does that with a specific offset, so that after the meta data section was written, it can add the *string table header* that contains the size of the meta data section that is not known before. For each entry in the string table, the compressor's method to encode a single string is called and writes the *encoded string* to the memory buffer. Again, an offset of the size of the *entry header* is kept, that is written after the encoding (compression) of the string has completed as it includes the encoded string's length. Once all strings are written, the string table header is updated once again with the total length of the string table. Storing the sizes of the entries as well as the total size of the string table allows to iterate the table very fast without decoding every entry.

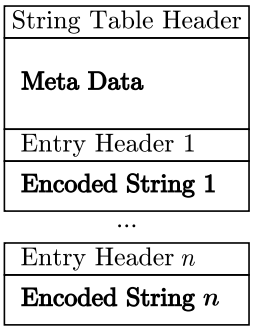


Figure 3.1: Layout of the serialized string table of a Carbon archive file. The compressor only controls the parts in bold print, the headers are automatically generated by the compression framework.

While the encoding process encodes the whole string table at once, the decoding procedure must support reading individual entries as the string table can be larger than the memory of the decoding machine (for the use cases of Carbon archives, see Section 2.1.1). Therefore, when loading a Carbon archive file, the compressor is requested to only decode the meta data section of the string table and keep it in memory. When an entry is requested by its ID, the table is searched for it by only decoding the fixed-length headers that contain the ID and once the entry has been found, the compressor is called to decode that specific entry.

This puts some restrictions on the compression algorithms that can be used: methods that only perform well when applied to larger amounts of data or that require decoding whole blocks of data are not suitable for the application in Carbon archives. For that reason, algorithms that work on lists of values (like the ones described in Section 2.2.3) instead of large single values or blocks are preferable.

To enable the implementation of the methods in the following sections of this chapter as well as the ones in Chapter 4, the interface was slightly extended:

- When instantiating a compressor, the document can be passed. This allows a more efficient resource allocation.
- To allow the compressor to prepare internal data structures, the interface has been extended to include an additional preparation step that is called between the instantiation of the compressor and writing the meta data section.
- An options framework was introduced so a compressor can define arguments it wants to accept. The command line tool (`carbon-tool`) was modified to accept the arguments via the command line (`--compressor-opt` ***`key value`***) and pass them to the compressor.

- The framework keeps track of the column a string value originated from and passes it to many of the compressors functions. The usage of these values is explained in Section 3.4.

All modifications presented in this section do not influence the file format and are therefore fully compatible with existing Carbon archives. The changes to the compression framework were kept as small as possible to allow the techniques, presented in this chapter and the following one, to be implemented while keeping the interface's lean nature.

3.2 Compression Building Blocks

This section will briefly describe the compression algorithms that have been added to the Carbon framework because they were considered suitable for the compression of the string tables according to the requirements as stated in Section 2.3.2 and Section 3.1.

3.2.1 Huffman Coding

The Carbon library already includes a Huffman encoder, but the decoding was not implemented at the time of writing, as mentioned in Section 2.1.2. As stated in Section 2.2.1, arithmetic and finite state entropy coding can beat Huffman in terms of compression ratio, but both are not designed to allow random access (at least not without sacrificing compression ratio) which makes them not suitable for the application on Carbon archives' string tables.

To increase the performance, the Huffman implementation was rewritten and now provides a more flexible interface as it is not coupled with the Carbon compression interface anymore. Instead, it can work on any string input. This allows the integration of Huffman coding into other algorithms. It also includes an adaptive form of Huffman coding, which has two advantages:

- No code table is required. While standard Huffman coding usually inspects either the whole text or some samples beforehand to estimate the symbols' probabilities of occurrence and stores the generated codes in a meta data section, adaptive Huffman coding does not need to do this. It starts with defined initial probabilities (e.g. for an alphabet of n different symbols $P(s) = \frac{1}{n}$) and updates them when the entries are being read, so no code table needs to be stored.
- Changing probabilities are reflected. When the probabilities of certain symbols change in the data (e.g. between the beginning and the end), the Huffman tree

adapts to this (and therefore the codes do). That can lead to better compression ratios.

There are also two main drawbacks of this approach:

- The runtime performance of adaptive Huffman is significantly worse than using static probabilities. The impact of rebuilding the codes depends on how often the update is performed.
- The whole message needs to be decoded up to the point that is of interest. This makes random access of entries nearly impossible, which is required for a Carbon archive's string table (see Section 3.1). The probabilities could be resetted every n values and would lead to only $\frac{n}{2}$ values to be decoded on average, but that would significantly reduce the compression ratio.

One important thing to consider that is not actually a drawback of adaptive Huffman but weakens the code table overhead argument is the fact that with larger (or more) entries encoded, the relative impact of storing the additional code table drops. For these reasons, the adaptive version is currently not in use for compressing the string table, but a non-adaptive Huffman encoder is used.

To keep the overhead introduced by the code table as low as possible, the codes are stored ordered by code length, so the length does not have to be stored for each code, but only once for each code length that occurs in the code table. The length, encoded using fixed-length 8 bit integer, is followed by tuples of the symbol (as fixed-length 8 bit character) and the code. The very first byte of the table indicates the number of entries in the table. For example, the code table containing the codes $< a, 1 >, < b, 01 >, < c, 00 >$ would be stored as

$$\underbrace{3,}_{\substack{\text{Total number of symbols}\\ \text{8 bits}}} \quad \underbrace{1,}_{\substack{\text{Code length}\\ \text{8 bits}}} \quad \underbrace{a, 1}_{\substack{\text{symbols of code length 1}\\ \text{9 bits}}} \quad \underbrace{2,}_{\substack{\text{Code length}\\ \text{8 bits}}} \quad \underbrace{b, 01, c, 00}_{\substack{\text{symbols of code length 2}\\ \text{20 bits}}}$$

resulting in a seven byte overhead. The code table supports up to 256 symbols which is sufficient because all strings are – for performance reasons – interpreted as an array of bytes without considering multi-byte character encodings.

3.2.2 Front Coding/Incremental Coding

While Huffman removes *per symbol* redundancy, front coding as described in Section 2.2.1 helps removing redundant prefixes. With the algorithm requiring the decoding of its predecessors, it does not provide *real* random access of the entries and

therefore seems to contradict the requirement from Section 3.1. But as the encoder can be configured to encode every n-th entry to not be encoded relative to its predecessor by simply setting the common length(s) to zero and encoding the whole string, a constant time access[1] can still be guaranteed while not having a big impact on the compression ratio. Another advantage over block-based compressors is that the encoder operates on entry-level. Because of that, it can be implemented within the existing compressor framework.

The implementation for the Carbon library extends the basic approach to not just encode common prefixes but it is also capable of encoding common suffixes. Therefore, the algorithm's alternative name *incremental encoding* is more suitable in this case.

As the Carbon string table does not rely on the order of the entries but stores the ID, by which they are referenced in the document, in the entry headers, the table can be sorted. That can noticeably improve the compression ratio because the values with the longest common prefixes (or suffixes if sorted starting from the values' end) are stored consecutively.

The common prefix and suffix lengths are encoded as a fixed-length 8 bit integers each, allowing them to each represent up to 255 characters. As the storage structure of the string table (see Section 3.1) is optimized to iterate forward and no reference to the previous entry is stored in the entry header, an additional pointer back to the previous entries' header has to be stored within the encoded string to allow accessing the predecessor entry. The layout is visualized in Figure 3.2.

Relative Back Ref	Common Prefix Length	Common Suffix Length	Remaining String
1 - 10 bytes	1 byte	1 byte	n bytes

Figure 3.2: Incrementally encoded entry storage layout

To save space, the (relative) back-reference is a 7 bit variable-length encoded unsigned integer. This means that while the most significant bit is set, the next byte is part of the encoded value. As a result, values of up to 127 can be encoded using only one byte while still allowing values up to 2^{64}. When the encoder is configured to only encode either common prefixes or common suffixes, the other (unused) value is of course not stored.

[1] constant in the number of entries encoded incrementally (also called *delta chunk length* of n), not in the length of the individual values

3.2.3 Prefix/Suffix Dictionary Coding

The incremental encoder will always find and remove the longest common prefixes or suffixes when the entries are sorted. But as the values can only be sorted either from the beginning or from the end, incremental encoding will only be optimal for removing either common prefixes or suffixes.

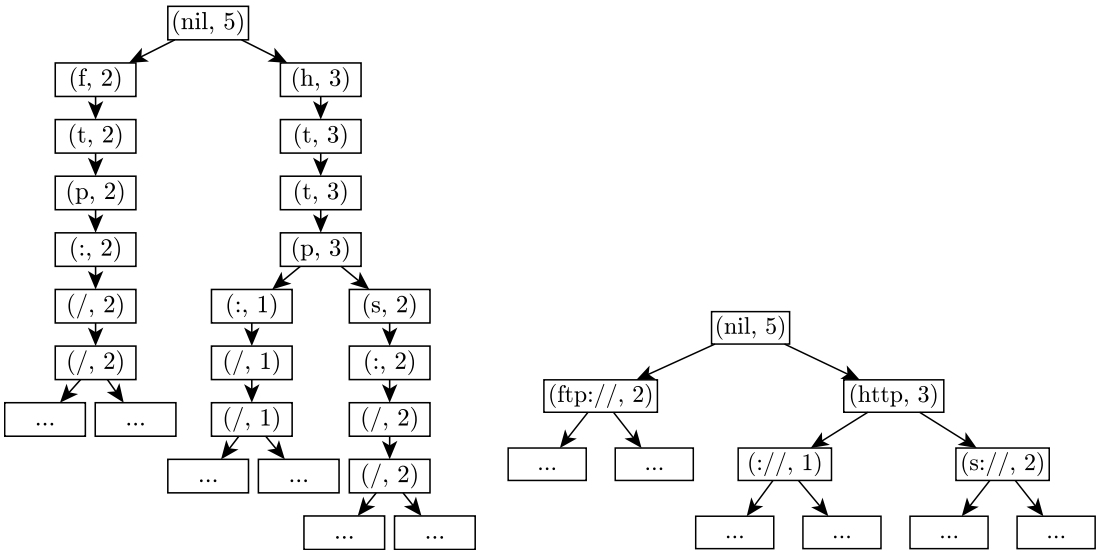
Prefix dictionary coding as explained in Section 2.2.1 does not have this problem. It stores the most common prefixes in a separate dictionary. When encoding a string, the longest matching prefix is looked up in the dictionary and the prefix is replaced by the prefix's ID in the dictionary. In contrast to incremental encoding, the prefix dictionary encoder allows *real* random access as the encoding and decoding of the prefix does not depend on other entries.

The same technique can be used to encode suffixes ("suffix dictionary coding"). Technically, the encoder could be extended to support encoding prefixes *and* suffixes, but for reasons explained in Section 3.3 that would never be used.

The dictionary is set up in the preparation step of the encoder, analyzing all strings that have to be encoded. It does so by setting up a trie [Fre60] using one node per symbol and also stores the number of visits ("count") in the node (see Figure 3.3a). To avoid running out of memory, the trie is pruned by removing all nodes with a count less than m after every n insertions, where n is set to a fixed value in the current implementation. In the end, all nodes with a count less than the *minimum support*, which is also configurable, are removed. To avoid removing too many nodes in the pruning phase, m should be smaller than the minimum support. Once all strings have been added to the trie, the nodes on a path are joined together as long as they share the same count (see Figure 3.3b).

These joined nodes are then put in a priority queue (with the count being the priority) because the dictionary size is limited to 2^{16} prefixes for implementation reasons and it should include the most frequent ones as they lead to more space savings than less often occurring prefixes. Another effect of that is, that more frequent prefixes are assigned with smaller IDs. As the IDs are encoded as 7 bit variable-length integers (just like the back-reference in the incremental encoder, Section 3.2.2), smaller IDs require less space so it is preferable to assign small IDs to the most frequently used dictionary entries.

To reduce the size of the dictionary, the prefixes are encoded recursively, such that prefixes that share a common prefix themselves are replaced by a reference to the common prefix that is stored in the dictionary anyway. In tests, this approach reduced the size of the dictionary by up to 90%. For performance reasons, the expanded form of table is kept in memory during encoding or when a Carbon archive is loaded.



(a) Trie with one node per symbol (b) Trie with nodes joined by count

Figure 3.3: Example of tries constructed from a set of URLs. While (a) allows faster insertions, the trie shown in (b) can be encoded efficiently as a dictionary.

Full Prefix	ID	Prefix ID	Remaining Prefix
`http`	1	0	`http`
`ftp://`	2	0	`ftp://`
`https://`	3	1	`s://`
`http://`	4	1	`://`

Figure 3.4: Recursively encoded prefix dictionary for URLs (the gray columns are not stored and were just added for better understanding)

As a result of the additional storing of the dictionary and its limited size, the approach is mostly less efficient for removing prefixes than the incremental encoder approach when the entries are sorted (or suffixes, when sorted from the entries' ends), as in that case the incremental encoder will always remove the longest common prefix[2] while prefix dictionary coding, especially for larger documents, might use a shorter prefix, which matches more entries.

3.2.4 Unsuitable Inputs for the Compressors

Not every input can be compressed to a shorter output – otherwise it would be possible to compress every input message to just one bit. Therefore, some inputs can be

[2]In our implementation, an additional assumption is that the common length is ≤ 255 characters long as the common prefix/suffix lengths are encoded as fixed-length 8 bit integers

considered "unsuitable" for a compressor: when the input leads to a compression ratio ≥ 1, applying the compressor is not useful. For each of the algorithms presented so far in this section, the unsuitable inputs are described in the following overview:

- **Huffman encoder**
  The only overhead the Huffman encoder produces is the code table. Theoretically, Huffman coding can create larger outputs when the predicted probabilities are not correct. But as we can analyze the complete input beforehand, we have accurate statistics available. Therefore, the Huffman coding only has to compensate for the code table and there are only two reasons why it cannot achieve that: the symbols are either near-equally distributed (which rarely happens in practice) or the input to compress is not long enough so the savings are not as high as the overhead of the code table.

- **Incremental encoder**
  The compression ratio for incremental encoding will be ≥ 1 if the common prefix (or suffix) length on average is less than $\approx (1 + \lceil log_{128}(l) \rceil) \cdot \left(1 + \frac{1}{d-1}\right)$ (with d being the maximum number of incrementally encoded entries and l the average encoded entry length). This threshold results from the per entry overhead that needs to be added to encode

  - the common prefix length: an 8 bit fixed-length integer $\Rightarrow$ 1 byte
  - the back-reference: storing the length of the encoded version of the previous string as 7 bit variable-length integers $\Rightarrow \lceil log_{128}(l) \rceil$ bytes
  - an absolute value every d entries, not saving any space, so the average prefix length removed in the remaining $d - 1$ entries needs to reduce the size at least by the total overhead added by the encoding.

  When using incremental encoding for both, prefix and suffix removal, the costs for the back-reference can be shared between the two as it only needs to be stored once.

- **Prefix/suffix dictionary encoder**
  For prefix or suffix dictionary coding, the overhead per entry is at least one byte to encode the dictionary entry to use (which can be zero indicating there is no matching entry in the dictionary), so the input needs to have an average common prefix length of at least one byte to save any space. The actual amount is higher, though, because it needs to compensate for the extra amount of space that the dictionary takes. This size is hard to predict because it depends on many properties of the entries in the dictionary: the amount of entries, their length and how

well can they be compressed using the recursive encoding.
If prefix dictionary coding is enabled although it cannot compress any prefixes, the overhead is much lower compared to the incremental encoder (1 byte vs. $\approx (1 + \lceil log_{128}(l) \rceil) \cdot \left(1 + \frac{1}{d-1}\right)$ bytes per entry).

3.3 Higher-Order Compression

The algorithms that have been implemented for the Carbon framework (see Section 3.2) all have their advantages and drawbacks as explained in the corresponding sections. But more importantly, they reduce different kinds of redundancy: the entropy encoder (Huffman) reduces the redundancy in the representation of individual symbols, the incremental encoder does so efficiently on consecutive entries and the prefix/suffix dictionary approach reduces the redundancy across the whole string table.

Original Value	Huffman (hex encoded)	Incremental	Suffix Dictionary
2019-08-15T01:23:45+0100	53216c3cd4fd77cd9c9d0	0,0,2019-08-15T01:23:45+0100	0,2019-08-15T01:23:45
2019-08-15T07:06:05+0200	53f6dfd7811dd1bac2950	12,2,7:06:05+02	1,2019-08-15T07:06:05
2019-08-15T10:22:33+0100	53216c3cd3bd2fef793a	11,2,10:22:33+01	0,2019-08-15T10:22:33
2019-08-15T12:34:56+0200	53216c3cd35fd37e1fc950	12,2,2:34:56+02	1,2019-08-15T12:34:56

Figure 3.5: Comparison of Huffman coding, incremental coding and suffix dictionary coding when applying them to a list of ISO formatted date strings. Meta data like code tables or dictionaries is left out.

The reduction of different types of redundancy is the reason why nearly all algorithms described in Section 2.2.2 combine different approaches to achieve better compression ratios, so there is little reason to assume this effect to be different for a Carbon archive's string table. To achieve that, a compressor that is able to combine the algorithms from Section 3.2 was developed. As it can be configured which of the algorithms to use (see Table 3.1), it has been called the *configurable compressor*. The compressor options can be set as described in Section 3.1 using the newly introduced compressor options framework.

The compressor first removes common prefixes and suffixes with incremental or dictionary coding, if these options are enabled. It can use each of the two algorithms to remove prefixes and suffixes, but prefix dictionary coding is used for at most one of them, because incremental encoding on sorted entries will perform better than prefix dictionary coding as explained in Section 3.2.3. Technically, the dictionary coding approach can only be applied to prefixes, but the compressor provides an option to reverse each string, and therefore common suffixes can also be removed using the prefix dictionary encoder. After removing common prefixes and suffixes, the remaining part can be compressed using Huffman coding, if enabled.

Compressor option	Values	Description
`prefix`	`none`	No compression of common prefixes
	`prefix-dict`	Use prefix dictionary coding for common prefixes
	`incremental`	Use incremental coding for common prefixes
`suffix`	`none`	No compression of common suffixes
	`incremental`	Use incremental coding for common suffixes
`huffman`	`true`	Use Huffman coding to compress the remaining text
	`false`	Do not apply Huffman coding
`reverse`	`true`	Reverses the strings to allow suffix dictionary coding using the prefix dictionary encoder
	`false`	Do not reverse the strings
`delta_chunk_length`	$n \in \mathbb{N}$	Maximum number of entries that are encoded incrementally if the incremental encoder is used

Table 3.1: List of the most important options for the configurable compressor

The choice between the different algorithms in the configurable compressor allows it to be adapted to the data it has to compress, as the compression performance of the two prefix/suffix removal algorithms depends on the input's characteristics. But also the ability to disable individual steps completely (prefix removal, suffix removal, entropy coding) in the compressor is important to increase its overall compression ratio, because disabling the application of compressors on inputs that are not suitable for them (see Section 3.2.4) ensures that the encoded output is at least not longer than the input.

Original Value	Prefix/Suffix removal	with Huffman entropy coding
`2019-08-15T01:23:45+0100`	`0,0,2019-08-15T01:23:45`	`0,0,0x99f35b8a6bbcc9e5d0`
`2019-08-15T07:06:05+0200`	`12,1,7:06:05`	`12,1,0xe8d632`
`2019-08-15T10:22:33+0100`	`11,0,10:22:33`	`11,0,0x7890ff`
`2019-08-15T12:34:56+0200`	`12,1,2:34:56`	`12,1,0x70fb8ac`

Figure 3.6: Example of combined application of algorithms (prefix: incremental encoding, suffix: suffix dictionary, entropy coder: Huffman) on a list of ISO formatted date strings (see previous example at Figure 3.5). The result is 27 bytes long instead of 96 (savings: $\approx 72\%$) without Huffman code table and the suffix dictionary, but these will grow less than linearly with the number of entries.

3.4 Key-based Column Compression

A Carbon archive usually contains more than one column, because the objects in the source JSON documents likely contain more than one property. While technically not required, it is likely that the values of one column originate from the same domain,

e.g. phone numbers or hyperlinks. Assuming the values to be from the same domain it is also likely that they have similar structure or characteristics. As the compression performance of the compressor described in Section 3.3 depends on using a suitable configuration with respect to the input data's structure and characteristics, the grouping of values by column can be exploited to use a better performing configuration for different subsets of the values instead of using one configuration for all entries.

The Carbon archives, however, consist of only one single string table. Additionally, the compression framework initially did not provide any information on the column a string originally belonged to. With changes to the internal document structure of the Carbon framework and the compressor interface (see Section 3.1), the origin of the strings is available to the compressor. To achieve a per-column compression, a proxy compressor is introduced: while implementing the normal compressor interface, it internally sets up a list of compressor instances (hereafter called "column compressors") – one per column – and acts as proxy that coordinates the instances.

In the prepare step, it first sorts all entries by their origin, so all entries belonging to the same column are grouped together, and calls the prepare function of each column compressor, passing only the subset of entries originating from the column the column compressor instance is responsible for.

After the preparation, the meta data section is written. The proxy compressor stores the number of column compressors (n) and writes n 64 bit integers representing zero as placeholders that will later be replaced by the file position at which the encoded values of each column compressor start. This is necessary to allow random access for decoding, because in that process the proxy compressor needs to determine which column compressor was used to encode an entry at a given file position. The offsets are followed by a section that stores the type of column compressor used and how many entries belong to that column. Currently, the same compressor – the one described in Section 3.3 – is used, but the approach can easily be extended to support different column compressors. Having written the proxy compressor-specific data, each column compressor is requested to write the meta data it needs.

The next step in serializing the string table is the encoding of every entry. As the entries have been sorted in the preparation step, all values belonging to one column are passed to the proxy compressor's encoding function consecutively by the compression framework. The proxy compressor forwards the encoding request to the column compressor that is responsible for encoding the values of the column the value originated from. When the first entry of a column has to be written, the current file position is used to update the offset placeholder in the meta data section indicating where the responsibility of the current column compressor starts, which is required for decoding

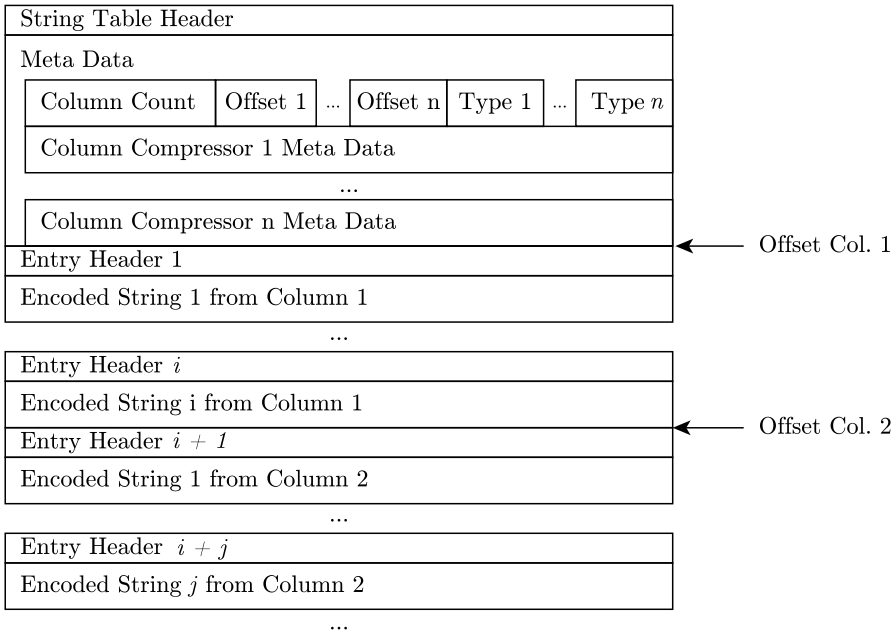


Figure 3.7: Storage layout of the string table when using the proxy compressor

as described above. The full storage layout resulting from that process is visualized in Figure 3.7.

When a Carbon archive with a string table compressed by the proxy compressor is read, the proxy compressor is instantiated. When parsing the meta data section, it creates the correct amount of column encoders of the specified types and requests each one to read their meta data. Together with the offsets for the first entry of each column, the structure is kept in memory.

Whenever a string needs to be decoded, the correct entry is searched as described in Section 3.1. Once found, the proxy compressor compares the entry headers file position with the columns' offsets and determines how many of them are less or equal than the file position. When n offsets are smaller or equal, the n-th column encoder is responsible for the decoding of the entry and the decoding request is forwarded to that column compressor.

With the approach presented in this section, it is possible to encode different entries of the Carbon archive's string table grouped by column with different compressors or configurations within a single string table.

3.5 Merging of Similar Configurations

Using one compressor per column can improve the compression as explained in Section 3.4. However, documents with many columns and only a few entries per column

create a large overhead as the meta data (e.g. Huffman code tables and prefix dictionaries) has to be stored for every column. Combining the columns containing similar values can help to improve the compression ratio, especially when encoding similar values incrementally (see Figure 3.8). Also, some columns may be from the same domain: in a table that stores orders, the two columns "order date" and "delivery date" will likely share the same optimal compressor configuration (see Section 3.3).

Value	CPL
"order date" (Chunk 1)	
2019-08-22T21:15:44+0200	0
2019-08-23T06:15:20+0200	9
2019-08-23T19:15:36+0200	11
2019-08-23T21:35:41+0200	11
"delivery date" (Chunk 2)	
2019-08-23T09:31:23+0200	0
2019-08-23T11:43:30+0200	11
2019-08-23T19:38:02+0200	12
2019-08-23T20:17:39+0200	11

(a) Incremental encoding (prefix only) separated for "order date" and "delivery date". The total common prefix length (CPL) is 65.

Value	CPL
Combined Column (Chunk 1)	
2019-08-22T21:15:44+0200	0
2019-08-23T06:15:20+0200	9
2019-08-23T09:31:23+0200	12
2019-08-23T11:43:30+0200	11
Combined Column (Chunk 2)	
2019-08-23T19:15:36+0200	0
2019-08-23T19:38:02+0200	14
2019-08-23T21:35:41+0200	11
2019-08-23T20:17:39+0200	12

(b) The values from Figure 3.8a with incrementally encoded prefixes, but both columns combined. The total common prefix length (CPL) is 69.

Figure 3.8: Example showing that sometimes data can be compressed better when the columns are combined using the incremental encoder. For better comparability, the delta chunk length (see Section 3.2.2) is set to 4.

To reduce the overhead that is introduced by splitting up the data column-wise, columns that share similar compressors are combined to one single column, so they are compressed by the same column compressor. The analysis of which columns can be merged has been integrated in the preparation step of the proxy compressor (see Section 3.4). All column compressors are compared with each other and if there is a sufficient similarity, the columns are merged. This is achieved by updating the references that are used to keep track of the origin (column) of an entry in the internal data structures that are used to manage the string values within the Carbon framework: when combining column A and column B, the references of B's entries are set to a value referencing column A. This operation is not limited to two columns, as the combined column can be merged again with another one. Theoretically, all columns can be combined into a single one.

To decide whether to merge two columns or not, their column compressor configuration's (see Section 3.3) similarity is computed. The configurations are considered similar, if all of the following conditions are satisfied:

- The prefix removal algorithm (incremental encoder, prefix dictionary, none) is the same for both configurations.
- The suffix removal algorithm (incremental encoder, none) is the same for both configurations.
- When incremental encoding is used, both configurations must sort from the same direction (by prefix or by suffix).
- Both use Huffman coding or they both do not.
- In case of Huffman coding: the difference of the average bit length of the combined column compared to the separate encoding must be below a certain threshold. After some experiments, this has been set to a fixed value of 0.25 bits.
- The configurations belong to the same merge group.

These criteria are based on a few observations that will be described in the following paragraphs.

The most obvious observation is that merging two columns with both compressors configured to not perform common prefix removal or both not performing common suffix removal, no compression ratio is sacrificed by merging as there was no compression before. Due to merging them, however, there could be common prefixes or suffixes in the resulting entry set that have not been there before, so the application of common prefix or suffix removal would be beneficial. But as this would require analyzing the combined column's entries again, this is not done in the current implementation.

The incremental encoding for a sorted column does not sacrifice any compression performance when it is merged with a second column that is also sorted by the same end of the entries (prefix or suffix), even if the data from the two columns does not share a common structure. The reason for this is, that the sorting always groups the longest common prefixes or suffixes together and for each string, the one that shares the longest common prefix/suffix will still be in the list when the data of both columns is put together. The compression performance could even improve as the example in Figure 3.8 demonstrates, because there could be longer common prefixes or suffixes in the combined entries.

When two configurations performing incremental encoding on the non-sorted end of the values (e.g. merging incrementally encoded prefixes while the entries are sorted by suffix) are merged, then the compression can become much worse. In the example shown in Figure 3.9, the different time zones result in only half the savings because the incremental encoder for the individual columns was only very effective as all values ended with the same time zone offset. To avoid these situations, there could be two different strategies:

- Forbid merging two configurations that use incremental encoding at the non-sorted end. This approach avoids the negative effects of the example shown in Figure 3.9, but the benefits of merging columns as described in the beginning of this section, especially for incremental encoders as shown in Figure 3.8, will also be lost.
- Automatically switch to prefix/suffix dictionary coding when merging two columns that use incremental encoding at the non-sorted end. In experiments, this proved to be an effective strategy, because using incremental encoding on a non-sorted end usually makes sense only when all values end with the same suffix. Therefore, the prefix/suffix dictionary would be only very small, so the main drawback of prefix/suffix dictionary coding as described in Section 3.2.3 becomes negligible.

Because of the benefits gained at low operational costs, the second strategy is used.

Merging configurations with dictionary encoding at the same end theoretically does not lead to big losses in compression performance, because in the best-case scenario the two dictionaries share a lot of common entries and therefore the size of the dictionary for the combined entry set is smaller than the sum of the two dictionary sizes. In the worst case, the resulting dictionary will be as large as both individual dictionaries combined, but the entries will still be there and can be used. In practice, the dictionary has an entry limit (see Section 3.2.3), so there can be situations where not all entries of both dictionaries can be stored. But due to sorting the extracted common prefixes/suffixes stored in the dictionary by frequency, only the least-frequently used entries would be removed in that case, not leading to significantly less savings. Also, in the experiments, the dictionaries rarely grew so large that this was really an issue. On the other hand, the savings achieved by using a combined dictionary were significant.

Whether the combination of two columns that are configured to be compressed using Huffman coding leads to less effective encoding – for Huffman coding that means a higher average bit length per symbol – depends on how similar the distributions of characters in the entries of the two columns are. The average bit length can be computed

Value	CSL
"order date" (Chunk 1)	
2019-08-22T21:15:44+0200	0
2019-08-23T06:15:20+0200	5
2019-08-23T19:15:36+0200	5
2019-08-23T21:35:41+0200	5
"delivery date" (Chunk 2)	
2019-08-23T09:31:23+1345	0
2019-08-23T11:43:30+1345	5
2019-08-23T19:38:02+1345	5
2019-08-23T20:17:39+1345	5

(a) Incremental encoding of the suffix for "order date" and "delivery date" when sorted by prefix (e.g. because prefixes are incrementally encoded, too). The total common suffix length (CSL) is 30.

Value	CSL
Combined Column (Chunk 1)	
2019-08-22T21:15:44+0200	0
2019-08-23T06:15:20+0200	5
2019-08-23T09:31:23+1345	0
2019-08-23T11:43:30+1345	5
Combined Column (Chunk 2)	
2019-08-23T19:15:36+0200	0
2019-08-23T19:38:02+1345	0
2019-08-23T20:17:39+1345	5
2019-08-23T21:35:41+0200	0

(b) The values from Figure 3.9a with incrementally encoded suffix (while sorted by prefix), but both columns combined. Due to mixed time zones, the total common suffix length (CSL) is 15.

Figure 3.9: Example showing that merging incrementally encoded columns sorted by the opposite end (prefix/suffix) can lead to much worse compression ratios. For better comparability, the delta chunk length (see Section 3.2.2) is set to 4.

using only the absolute symbol frequencies and the resulting code table. This allows the comparison between the average bit length for each individual column and the average bit length that is achieved by Huffman coding of the combined column that is predicted using the sum of the absolute symbol frequencies and the resulting code table. Therefore, a fast and exact prediction of the compression performance for the combined column is possible. A big saving that is especially important for scenarios with many columns that do not contain a lot of entries, is that the overhead for Huffman coding is reduced as only one code table needs to be stored instead of two.

To allow a column compressor's configuration to control the merging behavior, *merge groups* can be configured. A merge group is an integer value stored with the column compressor's configuration. Only columns with compressor configurations of the same merge group can be combined. The minimum value, 0, forbids merging at all, so once one of the two columns that are evaluated has a merge group of 0, the columns will not be merged, no further checks are made. Merge groups can be used to override the frameworks default behavior, but their comparison is just an additional condition that has to be satisfied – if at least one of the other conditions listed above is not met, the columns will not be combined.

4 Self-Driven Carbon Archive Compression

In Chapter 3 a configurable compressor was introduced, that can be applied with different configurations for each column through the proxy compressor (see Section 3.4). This chapter deals with the challenge of finding a suitable configuration for these compressors automatically to adapt it to each column's data.

4.1 Motivation and Big Picture

The configurable compressor as described in Section 3.3 can be used to adapt the compression algorithms to the data from the string table to achieve better compression ratios. However, configuring the compressor correctly requires the user to know about the available options and to have some basic understanding of the dataset. When using the proxy compressor from Section 3.4, the user needs to configure many compressors – one per column – which makes it a very time-consuming task.

The solution to that problem is to automatically detect a suitable compressor configuration. To do that, the proxy compressor was extended to support the optimization of the configuration based on a column's entries. Therefore, a generic *optimizer* interface was introduced to allow different optimization mechanisms to be implemented. These implementations are presented in Section 4.3.1, Section 4.3.2 and Section 4.3.3. The implementation can be selected at runtime by using the compressor options framework: the proxy compressor provides an option that can be used to specify the optimizer.

The optimizer is basically a function that takes a column's values, an optimizer-specific option set and a sampling configuration (see Section 4.2) and returns a configuration

for the configurable compressor with some additional information like a merge group (see Section 3.5) and the statistics used for the Huffman encoder if it is enabled in the configuration. The optimizer function is called in the proxy compressor's preparation step between the column-wise splitting of the data and the merging of the columns based on their compressor's configuration. This is visualized in Figure 4.1.

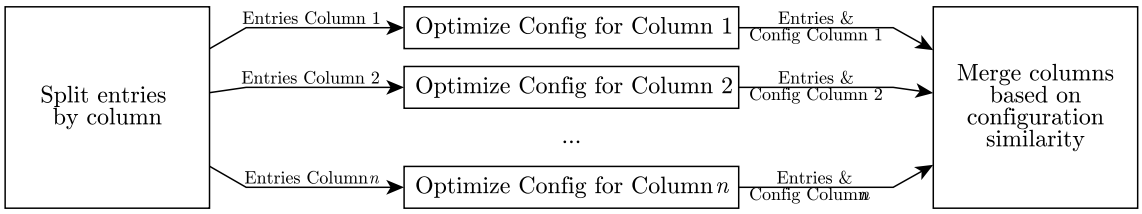


Figure 4.1: Integration of optimizers into the proxy compressor

4.2 Input Data Sampling

Irrespectively of how an optimizer decides what compressor configuration to use for a given list of strings, it always has to perform some kind of analysis on them. When analyzing all entries, the optimizer will take longer the larger the number of entries is.

2017-07-05T12:11:10
2017-07-05T12:11:11
2017-07-05T12:11:15
...few thousand left out...
2018-05-12T07:35:45
2018-05-12T07:35:49
2018-05-12T07:35:52
...few thousand left out...
2019-09-08T20:05:22
2019-09-08T20:05:25
2019-09-08T20:05:28

sampling individually →

2017-07-05T12:11:10
2018-05-12T07:35:49
2018-05-12T07:35:52
2019-09-08T20:05:22

Figure 4.2: Incremental encoder sampling example: While the original dataset has an average common prefix length of ≈ 18, for the sampled data it is only ≈ 7.7. Even when drawing more entries, this effect would still have a large impact.

To get a meaningful estimation of the entries' properties, it might not be necessary to consider all of them. Instead, using a representative subset of values can be sufficient and dramatically reduce the amount of effort which has to be spent on finding

a proper configuration. The different optimizer implementations all support sampling with exactly the same options. Sampling is enabled by default, but can be turned off.

The standard approach for sampling would be to randomly draw a fixed number of values from the full set of entries. While this is fine for the estimation of properties that are independent of the predecessors or successors of an entry, like Huffman or prefix/suffix dictionary coding, it can heavily underestimate properties like the average common prefix/suffix length for sorted incremental encoding. Figure 4.2 shows an example of this effect.

To circumvent that issue, the entries are sorted first and then n blocks of m consecutive entries instead of $n \cdot m$ individual values are randomly drawn from the full set of entries. Therefore, $m - 1$ values per block still have the predecessor they would have when the non-sampled set would be considered, such that properties which depend on their predecessors like the average common prefix/suffix length (see Figure 4.3 for an example) can be estimated much more precisely. Let $cpl(v_a, v_b)$ be the common prefix length of v_a and v_b and $v_{i,j}$ the j-th value of the i-th block. Then the optimizer can compute:

$$l_{avg\_est} = \frac{1}{n(m-1)} \sum_{i=1}^{n} \sum_{j=1}^{m-1} cpl(v_{i,j}, v_{i,j+1})$$

As a result, l_{avg_est} is a pretty accurate estimation of the average common prefix length as long as enough blocks are drawn. The same can be done for the suffixes, too. The number of blocks as well as their length can be configured through the compressor options framework. By default, 16 blocks with a length of 16 entries each are used, because they performed well in experiments: lower values led to less accurate estimations, while higher values did no significantly improve them (see Section 5.3.4).

The sampling is only applied when the total number of entries is greater than a certain threshold, which is set to $n \cdot m$ (block count · block length) by default, but it is configurable.

One drawback of sampling the entries randomly is that the resulting file may not be reproducible, e.g. when converting a JSON[Bra14] document to a Carbon archive file twice, they might use different configurations for some columns' compressors. That should happen very rarely, though, because even if different entries are selected during sampling, they should share the same structure and properties. But if the problem occurs and this behavior is unacceptable for some reason, the sampling can be made

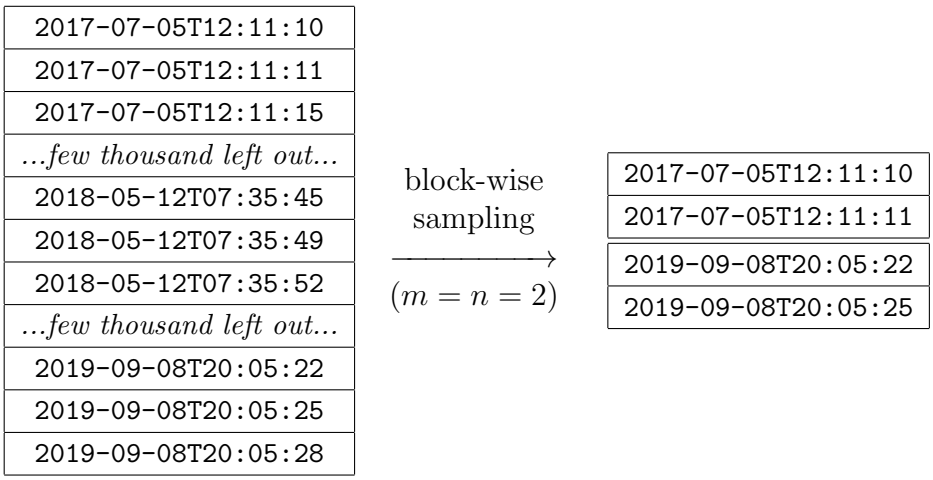


Figure 4.3: Incremental encoder block-wise sampling adapted from the example shown in Figure 4.2: $l_{avg} \approx 18$ for all entries, $l_{avg\_est} = 18$.

deterministic by using a fixed random seed. However, this is not part of the current implementation.

4.3 Compression Optimizers

To find a good configuration for the configurable compressor for a given input, an optimizer is used as described in Section 4.1. This section will present three optimizers that have been implemented and follow different approaches: optimizing by trying all possible configurations (Section 4.3.1), by using a cost model (Section 4.3.2) or by following user-defined rules (Section 4.3.3).

4.3.1 Brute Force Optimizer

The easiest-to-implement approach to find the best compressor configuration for a given set of values is the brute force approach: trying out every combination of parameters as configurations and compress the values using them all, finally taking the one that produces the best compression ratio.

The implementation considers all options defined in Table 3.1 except for the delta chunk length which is set to a fixed value, as it only allows to adjust the trade-off between runtime performance and compression ratio:

- The prefix encoder options $P = \{\text{none}, \text{incremental}, \text{prefix-dict}\}$
- The suffix encoder options $S = \{\text{none}, \text{incremental}\}$

- The binary string reversal option $R = \{\text{true}, \text{false}\}$
- The Huffman option $H = \{\text{true}, \text{false}\}$

Each configuration c is therefore a tuple from the search space $C = P \times S \times R \times H$. As the brute force optimizer evaluates all $c \in C$, it has to search through $|C| = 24$ configurations.

The evaluation of a configuration is done by creating a compressor with the configuration to evaluate and perform the compression with either the whole set of entries to analyze or just the sampled ones, if sampling is enabled. As the compressor interface is designed to always work on memory buffers, a block of memory is temporarily allocated where the compressor is requested to write its meta data and all of the encoded strings to. The number of bytes written to that chunk of memory can be interpreted as the result of the function that the optimizer tries to minimize.

Because the real compression functions are used – instead of models or approximations – the brute force optimizer returns the optimal compressor configuration (e.g. the one with the best compression ratio) for the given data when no sampling is used.

The main drawback of the brute force optimizer is its runtime performance: compressing all strings with 24 compressors can be a very compute-intensive task. This is becoming much worse if more options were added to the configurable compressor. The impact of this drawback is significantly reduced by using sampling, though. Also, because the configurations can be evaluated independently, the process could be heavily parallelized. The current implementation does not do this, because when the amount of strings to analyze – especially when using sampling – is small, then the overhead of multi-threading and the coordination of the threads is likely higher than the performance gains.

4.3.2 Cost-based Optimizer

The cost-based optimizer has been designed as a faster alternative to the brute force optimizer. Section 4.3.2.1 presents the general idea. The details of the model that has been implemented and its inputs are explained in Section 4.3.2.2 and Section 4.3.2.3.

4.3.2.1 Approach

Instead of running the compressor on the given dataset to find the best-compressing configuration, a model is used to predict the compression performance of a configuration.

The quality of a prediction when using a cost-based approach depends on how well the cost model describes the real algorithm's behavior. This approach therefore has one

major drawback: in contrast to the brute force optimizer from Section 4.3.1, it does not guarantee to find the best-compressing configuration. The advantage, however, is its much better runtime performance.

The cost model optimizer calculates some properties of the values to be compressed. These are only computed once and are then used to estimate the savings for each configuration without performing any calculations involving the input strings again.

4.3.2.2 Properties to Consider

In order to perform a reliable cost estimation, the relevant properties have to be computed based on the input values. They need to reflect what the compression algorithms would do. This section will describe them and explains, how they are computed.

- **Average common prefix & suffix length, sorted by prefix**
  The optimizer computes the average common prefix and suffix length when the input values – the sampled ones or all – are sorted by their prefix. This leads to two inputs for the cost model:

  $l_{p,p}$ ... prefix length when sorted by prefix

  $l_{p,s}$ ... suffix length when sorted by prefix

- **Average common prefix & suffix length, sorted by suffix**
  The optimizer computes the average common prefix and suffix length when the input values are sorted by suffix. Similar to the prefix-sorted ones, we get the properties

  $l_{s,p}$ ... prefix length when sorted by suffix

  $l_{s,s}$ ... suffix length when sorted by suffix

- **Costs for Huffman coding**
  As described in Section 3.2.1, the only overhead – and therefore the costs – when applying Huffman coding is the code table. To calculate the costs, the code table is generated from the estimated symbol frequencies (see below on how these are estimated) and then serialized like it would be done by the Huffman compressor. As a result, we get

  h_c ... The costs for applying Huffman coding (the code table size)

- **Savings when using Huffman coding**
  The savings when applying Huffman coding can also be computed without having

to actually apply Huffman compression to the input values. Again, this requires the estimated symbol frequencies whose retrieval is described below. Using these, the code table is set up and the average bit length (h_{abl}) that would result from applying Huffman coding is computed. This allows to calculate the estimated savings (assuming l_t is the total length of the input values):

$$h_s = \left(1 - \frac{h_{abl}}{8}\right) \cdot l_t \quad \ldots \text{ Estimated savings for applying Huffman}$$

For the last two properties listed above, the symbol frequencies need to be estimated. This could be done by simply analyzing the input data. But the probabilities might not reflect the actually encoded strings, because the common prefixes and the suffixes will be removed. This can distort the distribution of symbols, resulting in codes that are not optimal for the actually encoded text. To correct this, the optimizer assumes that the compressor will always remove the longest common prefix and the longest common suffix within the whole entry set (or the sampled one). Therefore, to estimate the entries more accurately, the optimizer first removes the longest common prefixes and suffixes and builds the Huffman codes using the remaining texts. An example of this difference is shown in Figure 4.4.

https://www.google.com
https://www.bing.com
https://www.yahoo.com

(a) Original dataset

https://www.google.com
https://www.bing.com
https://www.yahoo.com

(b) Dataset with dictionary-encoded prefix & suffix

Figure 4.4: When encoding the right hand side value set with a Huffman encoder based on the symbol frequencies of the left hand side set, the average bit length is 4.8 bits/symbol; when based on the right side set it is 3.2 bits/symbol.

The cost model's input is a tuple of the six calculated properties explained above ($< l_{p,p}, l_{p,s}, l_{s,p}, l_{s,s}, h_c, h_s >$). Additionally, the total entry count N is available to the model.

4.3.2.3 Cost Model

The cost model is a function that maps a compressor configuration to a score using the properties explained in Section 4.3.2.2. This score is a measure for the savings compared to the uncompressed data, so the goal of the optimizer is to maximize it.

To estimate a compressor configuration from the search space (as defined in Section 4.3.1), the effect of configuration options is evaluated using the cost model inputs.

The effects are estimated independently for prefix removal algorithm (s_p), suffix removal algorithm (s_s) and entropy encoder (s_e), where currently only Huffman coding is supported. The total amount of savings is the sum of these three:

$$s_{total} = s_p + s_s + s_e$$

The individual savings are estimated as:

- **Prefix/suffix not removed**
  When not applying any prefix or suffix removal, no savings can be achieved. However, there are also no costs introduced. Therefore the savings assigned for prefix/suffix removal is zero:

  $$s_p = 0 \quad \text{or} \quad s_s = 0$$

- **Incremental encoding for the sorted end**
  The application of the incremental encoder on the sorted end (e.g. incremental prefix encoding when the values are sorted by prefix) will result in the estimated common prefix/suffix length $l_{p,p}$ (for prefixes) or $l_{s,s}$ (for suffixes) minus 2.5 bytes per entry to roughly approximate the costs described in Section 3.2.4. Due to that, the estimated savings are computed as:

  $$s_p = (l_{p,p} - 2.5) \cdot N \quad \text{or} \quad s_s = (l_{s,s} - 2.5) \cdot N$$

- **Incremental encoding for the non-sorted end**
  When applying incremental encoding on the non-sorted end (e.g. prefix encoding when the values are sorted by suffix), the average common prefix/suffix length is estimated as $l_{s,p}$ (for prefixes) or $l_{p,s}$ (for suffixes). Just like in the sorted case, this leads to:

  $$s_p = (l_{s,p} - 2.5) \cdot N \quad \text{or} \quad s_s = (l_{p,s} - 2.5) \cdot N$$

- **Prefix dictionary**
  When using prefix dictionary coding, it is assumed that it performs – independent of the sort order – as well as the incremental encoder in the sorted case, but at higher fixed costs as it needs to store the dictionary, too. As a result, experiments showed that 4 bytes per entry reflect the real costs pretty well (see Section 5.2.2), leading to:

  $$s_p = (l_{p,p} - 4) \cdot N \quad \text{or} \quad s_s = (l_{s,s} - 4) \cdot N$$

- **Entropy coding using Huffman encoder**
  As the costs for applying Huffman coding (h_c) as well as the savings that will be achieved by applying it (h_s) can be predicted pretty well, the estimated savings are:

$$s_h = h_s - h_c$$

The string reversal option that is required to enable suffix dictionary coding (see Section 3.3 for a detailed explanation) of the compressor does not save any space itself and does not lead to costs, but it needs to be included in the cost model: when the strings are reversed, the prefixes become the suffixes and vice versa, so $l_{p,p}$ and $l_{s,s}$ are swapped as well as $l_{p,s}$ and $l_{s,p}$.

The search space as described in Section 4.3.1 contains 24 possible combinations that can be quickly evaluated using the formulas above. But as the decision of whether to enable the entropy encoder or not is independent from the other computations, the Huffman encoder options is simply enabled by comparing its costs and savings ($h_s > h_c \rightarrow$ enable Huffman encoding, otherwise disable it). As a result, the Huffman option is not part of the search space anymore, reducing its size to 12 combinations.

4.3.3 Rule-based Optimizer

The rule-based optimizer allows the user to precisely configure the compression behavior of the system. Section 4.3.3.1 presents the general idea, whereas Section 4.3.3.2 and Section 4.3.3.3 describe how the rules are defined and evaluated.

4.3.3.1 Approach

The brute force (Section 4.3.1) and the cost-based approach (Section 4.3.2) use different ways to decide, what configuration to choose. But both share the same basic idea: the computer analyzes the data and finds a configuration without any user interaction. For some scenarios, however, the user of the system might want to be able to override this behavior and define his or her own compressor configurations for certain columns or types of information.

To serve this need, the rule-based optimizer allows the user to specify rules, when to apply which configuration. Two strategies were considered for the implementation:

- **Specifying rules by column name**
  The configurations are assigned to certain column names. These rule sets can be evaluated very fast, because only the keys need to be compared to find a

matching configuration. The major drawback of this approach is that it is not very flexible: once a column is renamed or unknown datasets are encountered, the rules will not match anymore. There is also the possibility of false positives in rule matches, when a name is not unique or has different meanings in different contexts/domains.

- **Defining rules by value patterns**
  The rules can be matched by the values. This means, that patterns have to be defined that need to be compared with the values. The advantage compared to the first approach is, that the rules can be applied independently of the column names, so changes to the structure or unknown datasets can be handled better. As it requires more than a simple comparison of the column names, the computational effort is much higher, though.

Because the latter approach is more general and the computational resources required can be reduced through sampling (see Section 4.2) and other measures (see Section 4.3.3.3), it has been implemented instead of the name-based approach.

4.3.3.2 Rule Definitions

The proxy compressor (Section 3.4) accepts an option for the rule-based optimizer called `rule-based.filename`, allowing the user to provide a file containing rule sets.

Each rule definition consists of four components:

- **Name**
  The rule name can be chosen by the author and has no influence on the optimizer itself. It must be unique across the rule set file. The name will be logged in debug output and therefore mainly serves troubleshooting purposes when developing or optimizing rules.

- **Pattern**
  The patterns the values are compared against are expressed as POSIX regular expressions [The18]. The pattern has great influence on the performance of the optimizer: the evaluation of regular expressions can require huge computational resources in some cases. To analyze a regular expression's performance, debugging them using tools like RegexBuddy [Goy] might help.

- **Merge group**
  To control which configurations can be merged, a merge group can be configured that follows the rules from Section 3.5. When the merge group is not specified, merging is disabled.

- **Compressor options**
  A rule has to specify the compressor options for the configurable compressor. They are applied when the rule matches. The available options are explained in Section 3.3. The current implementation only supports the configurable compressor, but the approach can be easily extended to support any other compressor. It obviously only makes sense for compressors whose behavior can be changed by options, though.

The rules are defined in a plain text, line-based key-value format. Listing 4.1 shows an example of how a single rule can be defined. To define multiple rules, the definitions are simply concatenated.

When a rule named *default* is found, it is handled slightly different than the other rules: instead of matching a pattern, it is always applied when none of the other rules match.

```
1 phone.pattern=\+?[0-9\s\(\)\-/]{8,}
2 phone.merge_group=3
3 phone.option.prefix=incremental
4 phone.option.suffix=none
5 phone.option.huffman=true
6 phone.option.reverse=false
```

Listing 4.1: Example rule definition for telephone numbers

4.3.3.3 Rule Matching

The naïve approach of matching the rules to the values is simple: for all rules, the values to analyze are compared against the rule's pattern and the number of matches is stored. This can lead to three different situations, that are evaluated in the order defined below:

1. If at least one rule has matched, take the rule matching most values. If two rules match exactly the same number of values, the rule defined furthest up is used. Therefore, the order in which the rules are defined in the rule set file can matter, which allows the user to express the precedence of rules.

2. If no rule has matched any value but a default rule is defined, return the configuration defined for the default rule.

3. If no rule has matched any value and no default rule is defined, return a configuration with no compression algorithms enabled (for the current implementation, this means `prefix=none`, `suffix=none`, `huffman=false`).

The drawback of this approach is that in each case all values have to be compared against all rules. To reduce the amount of regular expression comparisons, two different *early out strategies* could be implemented:

- **Global early out**
  A minimum support criteria is introduced that can be specified by the user. It is a value greater than zero and less or equal than one. The first rule which matches at least the specified fraction of values (e.g. if the minimum support is .75 and 1000 values have to be analyzed, then at least 750 must match the rule's pattern), the rule is returned without checking the following rules. When rules that match very often are defined before rules that are less likely to match, this can save a lot of computation. The main drawback is, that it does not guarantee to find the best-matching rule: when the minimum support is set to .75 and a rule matches 78% of the values, it will be returned, even if there may be a rule defined after that one that would have matched 100% of the entries.

- **Per-rule early out**
  Just like for the approach described above, a minimum support is defined. But in this case it defines the minimum support a rule needs to get to be accepted at all: when the minimum support is set to .75 and 1000 values are evaluated, the best-matching rule is not considered a match when it only matches 700 values (which means the default rule or a configuration with all algorithms disabled is returned). This can drastically speed up the evaluation of rules, because during their evaluation, the optimizer permanently checks, if it is theoretically impossible to achieve this minimum support: when it has evaluated $> 25\%$ of the values and not a single one has matched, it does not continue the evaluation because even if this rule would be the best-matching one, it would get a support $< 75\%$ so it will be ignored anyway. When the rules are defined in a way that they should match (nearly) all values, the minimum support can be set very high (e.g. to .95), which means that rules that do not match a single value will only evaluate the first 5% of values and then abort. This approach guarantees to find the best-matching rule, as long as it matches at least the fraction of values defined by the minimum support criteria.

The current implementation uses the second approach as it finds the best-matching rules. Requiring a minimal support to accept a rule can also be desired behavior: a rule that "accidentally" matches 2 out of 1000 entries might be the best-matching rule, but it might not be really suitable, so it actually should fall back to the default rule. The second approach also adapts better to the content to be analyzed: the more entries

match the pattern (e.g. the more likely it is that a rule could be the best-matching one), the more computational effort is spent on analyzing it.

Theoretically, both approaches could be combined. But that would mainly emphasize the drawbacks of both methods: it would not always find the best-matching rule and it would reject rules below the per-rule minimum support criteria. For this reason, if an additional speedup is required, using sampling might be the better choice.

Besides these strategies whose application may influence the rules that are going to be selected, another optimization close to the per-rule early out approach is used that does not change the outcome: instead of or in addition to using a fixed minimum support, the highest matching score is used to determine when to abort the evaluation of a rule. For example, when a previous rule has matched 70% of the entries, the evaluation of the following rules is aborted when the algorithm can determine that this score cannot be achieved anymore, e.g. when $\geq 30\%$ of entries have been evaluated but no entry matched so far.

4.4 Caching Optimizer Results

When many similar datasets or datasets containing different records but from the same domain are processed (or even the same dataset is processed multiple times), the optimizers still have to decide for each dataset and each column, what compressor configuration to use. This is unnecessary when the properties of the columns do not change.

To avoid this, a configuration cache file can be specified using the `config-cache` compressor option. It will be loaded before the optimizer is called and if it does not exist, it will be created. Before calling the optimizer for a column, the cache is checked for an entry that was previously created for a column of that name. If an entry is found, the configuration will be loaded from the cache and the optimizer has not to be called and therefore the whole process is accelerated, as the time for retrieving the configuration from the cache is negligible. If no cached configuration is found for a column, the optimizer is called and the configuration it returned is added to the cache, so next time a dataset with that column name is processed (and the same cache file is used), the entry is taken from the cache.

The cache file is in a binary format, containing the column name the configuration was used for, the merge group, the configuration options as well as the symbol (character) frequencies that were calculated for the Huffman compressor component. The latter is required as it influences the calculation of the similarity of two configurations in the merging process as described in Section 3.5, but it is later recalculated when the

actual compression is performed so it matches the symbols' frequencies in the currently processed column.

While this cache can achieve a huge performance benefit, using the same cache for datasets that share column names but with a different kind of entries, the configuration loaded from the cache may not be suitable. For that reason, it is up to the user to use the same cache file only for datasets with similar properties. For that reason, configuration caching is not enabled by default.

5 Evaluation

In this final chapter, the implementations from Chapter 3 and Chapter 4 are benchmarked using a set of JSON [Bra14] files that have been selected to represent the relevant use cases for the Carbon archive files as described in Section 2.1.1.

5.1 Benchmarking Setup

In preparation of the following sections, this one describes how the implementations are benchmarked.

5.1.1 Datasets

The evaluation of the approaches presented in the previous chapters can only be realized against a concrete collection of datasets, because the performance of compression algorithms heavily depends on the data that needs to be compressed (see Section 3.2.4). It is important, though, that this selection represents the actual use cases of the system to allow conclusions on its performance in real-world scenarios.

Carbon archives are designed to handle analytical queries on columnar datasets which most likely origin from JSON files (see Section 2.1.1). Therefore, columnar JSON datasets with different properties are used for the evaluation. They are taken from different, public sources. Some of them, like (parts of) the GitHub API dump and the Microsoft Academic Graph are also considered for comparability as they have been used in the past to benchmark the Carbon archives [Pin19b].

The datasets used for the evaluation are:

- **GitHub API dump**
  The GitHub code hosting platform provides a public REST API[Ric07] to retrieve nearly all (public) information on projects hosted there. From that API, data on ≈ 25000 projects has been collected into one JSON file. It contains 46 properties per project, of which most are URLs that share a common structure (e.g. prefix and suffix). The file is ≈ 103 MiB in size.

- **Microsoft Academic Graph (MAG)** [SSS+15]
  The Microsoft Academic Graph is a dataset that contains meta data on scientific publications and their relations (e.g. citations). It consists of (relatively) long and short natural language texts (e.g. publication title and abstract) as well as URLs and UUIDs [Int96]. Only a subset of the whole Microsoft Academic Graph is used. It contains ≈ 50000 records, resulting in a file size of ≈ 88 MiB.

- **Datasets containing mainly short texts**
  Four datasets were taken from different cities' open data portals. They contain records of short natural language strings like names and localized dates, numbers as strings of digits, telephone numbers and e-mail addresses. They have been selected for the evaluation as they are good examples of very short columnar data that can be found in many databases, e.g. of the public administration.

  – Statistical data on the households of the city of Wesel in 2007[1], ≈ 550 KiB
  – List of employees of the city of Geldern[2], ≈ 35 KiB
  – Sociological data of districts of the city of Düsseldorf 2015[3], ≈ 60 KiB
  – Schools in the city of Bottrop[4], ≈ 25 KiB

  The datasets are called `short-1` to `short-4` in this chapter.

- **Datasets containing mainly long texts**
  Three datasets were taken from different cities' open data portals and one from the job offering portal of the state of North Rhine-Westphalia. They all have in common that they mainly contain longer texts (natural language).

[1] http://data.geoportal-wesel.de/OPENDATA/Statistik/Haushalte/HH_in_WP/HH_in_WP.json
[2] https://www.geldern.de/de/system/-preview-text/&src1=json-mitarbeiter
[3] https://opendata.duesseldorf.de/api/action/datastore/search.json?resource_id=c0bf3217-f60e-4397-9d80-b52ed6af807a
[4] https://www.offenesdatenportal.de/dataset/33640643-21f7-46e6-81c3-b368b2af897b/resource/69979535-2e9c-4991-bdc6-04c492eb3232/download/schulen.json

- Event calendar of the city of Kleve[5], ≈ 25 KiB
- Public sector job advertisements in North Rhine-Westphalia[6], ≈ 2 MiB
- Points of interest of the city of Moers[7], ≈ 90 KiB
- Public suggestions for the reduction of noise pollution in the city of Cologne[8], ≈ 2.5 MiB

For the rest of this chapter, these are named `long-1` to `long-4`.

As the compression only operates on the Carbon archives' string tables, a relevant property of a dataset is the amount of string data in relation to the structural information and the meta data: datasets in which the string data makes up a larger part of the total size can be better compressed than the ones where structural and meta data takes up more space. Figure 5.1 visualizes how much of each file is actually string data when importing the datasets into uncompressed Carbon archives. The amount of string data is calculated by subtracting the file size of a Carbon archive using a "compressor" that "compresses" every input string by returning an empty string from the file size of a Carbon archive using no compression.

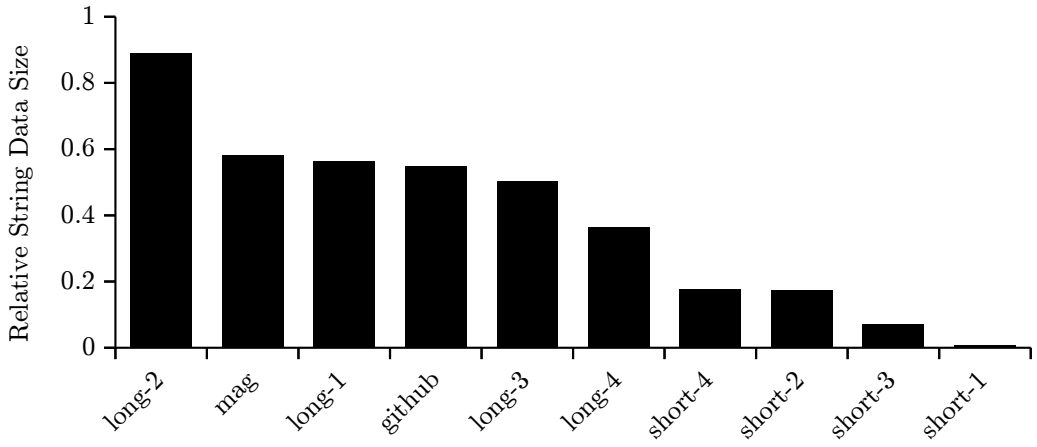


Figure 5.1: The amount of data taken up by the actual string data in relation to the total (uncompressed) Carbon archive file size

The datasets with fewer, longer strings result in a higher string data to dataset size ratio as the meta data and structural information required for long strings is the same as for shorter ones. Additionally, when the same strings occur frequently, they are only

[5] https://www.kleve.de/www/event.nsf/apijson.xsp/view-event-month?compact=false
[6] https://www.stellenmarkt.nrw.de/openNRWJobs/search.json
[7] https://www.moers.de/www/verzeichnis-13.nsf/apijson.xsp/view-list-plain
[8] http://offenedaten-koeln.de/sites/default/files/lap-koeln-2010-2011-alle-vorschlaege.json

stored once in the string table but the structural information (e.g. the references to the string) does not change in size. The latter is the main reason why the dataset "short-1" contains so little string data. But even with this small string table, the dataset is interesting for the evaluation, as any overhead introduced by compression algorithms becomes more relevant and may even increase the dataset's size. A good compression algorithm should detect and avoid this by not applying the compression.

The total size of the uncompressed datasets is ≈ 253 MiB, with ≈ 142 MiB being string data. This means, that even the best compressor could reduce the datasets size by at most ≈ 56%, which would mean that the string data is compressed to close to nothing which is practically impossible.

Therefore, to judge the compression performance of a compression algorithm itself, the reduction in size should be compared to the amount of string data. To evaluate the system as a whole, e.g. if the achieved space savings are worth the additional computational resources spent on compression, the whole compressed dataset's size must be compared to the total uncompressed dataset's size as this is the amount of storage that is relevant to the user.

5.1.2 Environment

All benchmarks in this thesis are executed on the same server to ensure that the results of different benchmarks are comparable. The machine has an Intel® Core™ i7-4770 server processor running at up to 3.4 GHz and 32 GiB of main memory at a clock speed of 1600 MHz. The server uses RAID level 1 (mirroring) with two Western Digital WDC WD2000FYYZ-0 (2×2 TB, 7200 rpm).

It is running a 64 bit Ubuntu 18.04 operating system based on a Linux kernel in version 4.15. The project's source code is built using the clang compiler [The] version 8.

The measurement of resource is performed by executing the `carbon-tool` from the Carbon framework with GNU Time [Gor]. The sum of the reported time the CPU spent in system and user mode is used as a measurement for the computational resources. For the evaluation of memory consumption, the largest amount of memory the process required – the maximum resident set size (often called `maxRSS` or `VmHWM` for "virtual memory high water mark" in the operating systems `proc` file system) – is used, as this information is the most relevant to judge how much memory a system needs at least to perform the operation that is being evaluated.

To achieve more stable results for the runtime performance analyses, the benchmarks are executed five times. The final runtime is the average of the measured values, ignoring the highest and the lowest measurement to compensate for caching effects or

short interruptions by system tasks. To reduce the influence of the operating systems filesystem cache on the I/O performance, the system's caches are flushed before each benchmark execution.

5.2 Isolated Building Block Investigation

While the effects of the implemented compressors were already discussed theoretically in Chapter 3, this section is presenting the results when evaluating them in the context of the datasets described in Section 5.1.1. The evaluated compressors are all part of the configurable compressor (see Section 3.3) and are applied to the whole string table (in contrast to the key-based application as described in Section 3.4).

5.2.1 Huffman

The Huffman compressor is an entropy coder that is – unlike the incremental or prefix dictionary encoder evaluated in Section 5.2.2 and Section 5.2.3 – not parameterized, so the results presented do not depend on any inputs except for the datasets and no parameters have to be optimized. The benchmarks are run by enabling only the Huffman compressor component of the configurable compressor (see Section 3.3) and no prefix or suffix compression.

As described in Section 3.2.4, the Huffman compressor works best when some symbols occur much more frequently than others, so datasets consisting mainly of URLs or strings of limited alphabets (like phone numbers or UUIDs) should achieve better compression ratios than others.

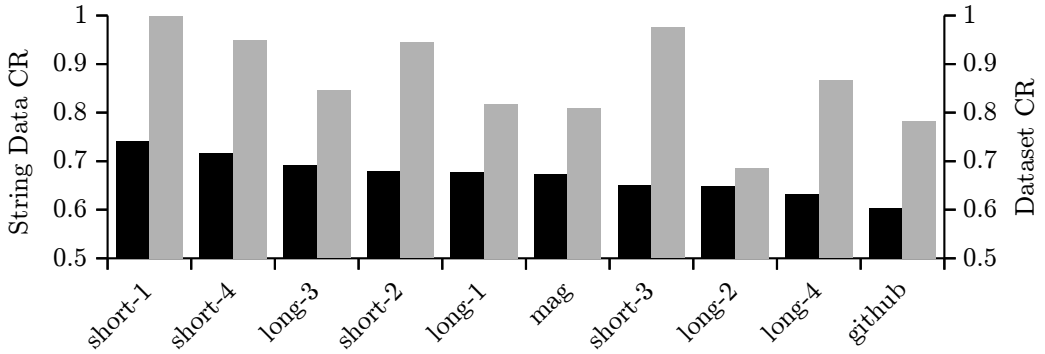


Figure 5.2: Compression ratios (CR) of the string data only (black) and the whole datasets (gray) when using the Huffman compressor, ordered by string data compression ratio

Figure 5.2 might seem to contradict this assumption, as datasets containing a lot of natural language text (the ones prefixed with "`long-`") achieve better string data compression ratios. But in fact, the compression ratio for the worst compressed datasets is

influenced negatively through the overhead Huffman coding introduces, the code tables. The smaller the input is, the larger is the relative overhead and the datasets where the compressor performs worse are mostly the ones with least string data (`short-1`: ≈ 2.5 KiB, `short-2`: ≈ 8 KiB `short-4`: ≈ 6 KiB, `long-1`: ≈ 50 KiB, `long-3`: ≈ 17 KiB). The only exception in this case is the dataset `short-3`, which only contains ≈ 3 KiB of string data, but the majority of that are digits so they are compressed very well by the Huffman encoder and the limited alphabet with short variable-length codes additionally leads to a smaller code table size.

This effect is reflected in Figure 5.3: while the datasets containing more strings of limited alphabets like phone numbers and dates (the datasets prefixed with "short-") have smaller code tables, the impact of adding ≈ 130 bytes to a small dataset like `short-1` is much larger than the overhead of ≈ 500 bytes on the `github` dataset, which contains ≈ 30000 times as much string data as `short-1`.

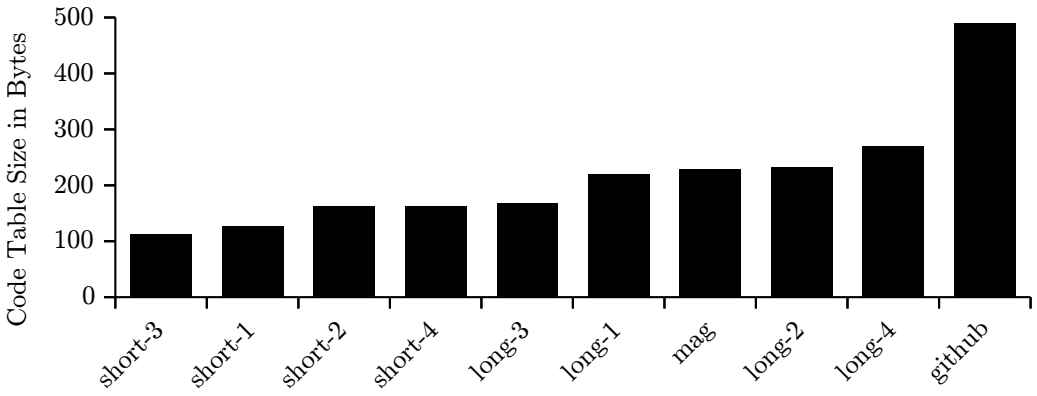


Figure 5.3: Sizes of the Huffman code tables for each dataset in bytes

It is important to note, though, that even the dataset with the worst compression ratio still achieves a reduction by 24% in size for the string data it contains. As the amount of string data in this dataset is very small in relation to the total dataset's size, this has nearly no effect, but none of the datasets becomes larger through the compression process. Regarding the whole dataset's compression ratio, the dataset achieving the best result (`long-2`) is reduced to about 69%. The pure string data of the `github` dataset is even compressed to ≈ 60% of its original size.

The compression introduces a computational overhead which grows approximately linearly with the amount of string data to be compressed, at least for the larger datasets. The import of the JSON datasets takes about 15 to 25 milliseconds per megabyte longer. For the smaller datasets, other effects like reduced I/O operations lead to the process completing slightly faster (about one percent) than without compression en-

abled. Due to additional data structures that need to be kept in the RAM, the memory consumption slightly increases by ≈ 64 KiB.

For the decompression process, the results are much less consistent between the datasets: while one dataset (`mag`) requires significantly longer to decode (≈ +14%) compared to the uncompressed scenario, the other ones complete faster (≈ −3% to −6%). The result is reproducible and is related to the different distribution of the symbols' code lengths as well as more (computationally expensive) memory re-allocations being required. The latter issue could be solved or at least reduced by fine tuning the implementation.

The same behavior can be observed for the memory consumption: for all smaller datasets, the required memory does not change noticeably (±5 KiB, for `github`: ≈ 30 KiB), except for the `mag` dataset: its memory consumption increases by more than 500 KiB compared to the processing of the uncompressed version, because of the way the memory is allocated during the decoding process.

5.2.2 Prefix Dictionary

The prefix dictionary compressor is evaluated by enabling only this component of the configurable compressor described in Section 3.3. In contrast to the Huffman compressor benchmarked in the previous section, the prefix dictionary compressor requires one parameter to be set: the minimum support required to accept a dictionary entry (see Section 3.2.3). Therefore, the influence of this parameter is analyzed first to find the optimal setting with respect to the compression ratio. In the second part of this section, the optimal configuration is further analyzed regarding its performance.

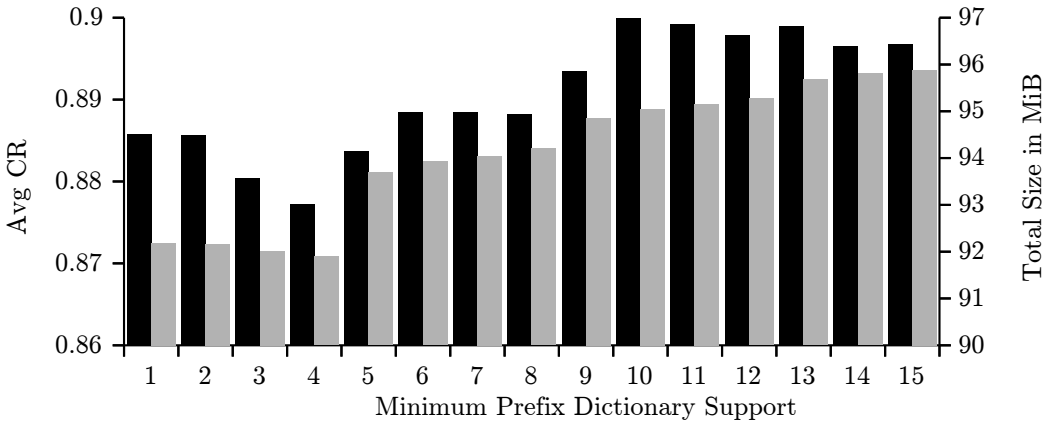


Figure 5.4: Average compression ratio (CR) of the string data (black) and the sum of the compressed string data size (gray) depending on the minimum support

The minimum support parameter defines, how many entries need to share a given prefix to include it in the dictionary. To find the best setting, all datasets are compressed with

minimum support values ranging from 1 (include all prefixes) to 15 (require a prefix to be shared by at least 15 entries). The result is visualized in Figure 5.4.

For both, the average compression ratio and the sum of the file sizes of the compressed datasets, the minimum is reached at a minimum support of 4. This is also very consistent across all datasets. The reason for this behavior is that the optimal minimum support should reflect the costs of including an entry in the dictionary. When the parameter is chosen too small, e.g. smaller than the costs, some prefixes may be included in the dictionary that require more space than they are saving. On the other hand, when using a minimum support above the costs, some prefixes may not be included although they would save more space than they take. For the rest of this chapter, the minimum support is set to 4.

Using this configuration, the compressor achieves some very good compression ratios for some datasets and no compression at all for others (see Figure 5.5).

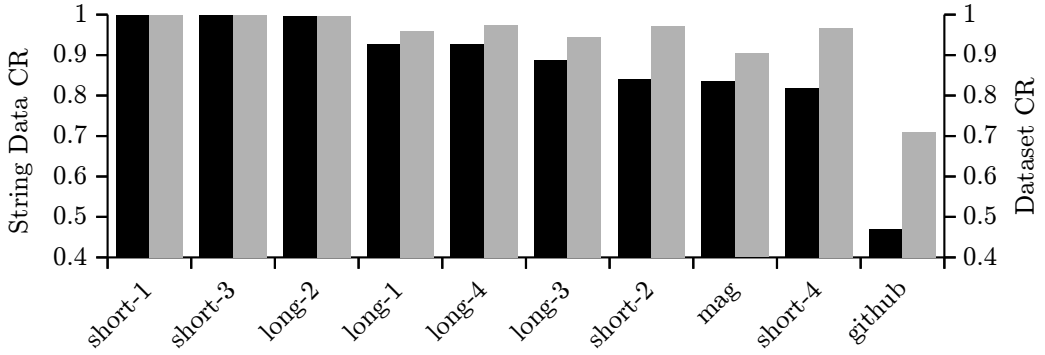


Figure 5.5: Compression ratios (CR) of the string data only (black) and the whole datasets (gray) when using the prefix dictionary compressor, ordered by string data compression ratio

For most smaller datasets, there are not enough common prefixes to achieve any compression. Also, larger datasets containing long texts do not benefit very much from this approach. In contrast, datasets containing many similar URLs or phone numbers from the same region can be compressed very well, with the `github` dump being a very good example: most of its values start with `https://api.github.com/repos/` or `https://api.github.com/users/` which is replaced by one single byte referencing a dictionary entry, leading to a string data compression ratio of $\approx 46\%$ (which results in a dataset of $\approx 71\%$ of the original size).

The computational overhead the compressor introduces during the import of the JSON files is higher for datasets with larger dictionaries, as during the compression process, the best-matching (e.g. the longest) prefix from the dictionary has to be found for

each string table entry. Especially for the `github` dataset the compressor introduces a large overhead of about 55 milliseconds per megabyte of string data, because nearly all prefixes share the same first 20 to 30 characters (protocol, host and path of the URLs, see above), so each comparison takes longer. A positive aspect of this effect is that the computational overhead is usually higher for datasets where the compression results are better, which is also reflected in the benchmark: the datasets with the largest overhead (in milliseconds per megabyte of string data) are: `github` ($\approx$ 55), `mag` ($\approx$ 30) and `long-3` ($\approx$ 20).

As only entries are kept in the dictionary that actually compress the dataset (due to the choice of the minimum support parameter), larger memory overheads are mostly observed for datasets where the compression leads to good results: the largest memory overhead can be observed for the `github` dataset. It is only $\approx$ 80 KiB, though.

When decoding the compressed datasets back to the JSON format, the runtime overhead is very low – except for the `github` ($\approx$ 2.5%) dataset it is actually negative (e.g. $\approx -8\%$ for the `mag` dataset). The reason why the decoding process is so fast is that the prefix dictionary is read and decoded once and then kept in memory, such that the decoding of every string requires only the access of a given position in an array. While the decoding itself is very fast, the compression leads to less data being read from the hard drive and less memory to be allocated and that – for most datasets – overcompensates the additional time required for decompression. The overhead observed for the `github` dataset results from the decoding of the prefix dictionary which contains many entries. As the prefix dictionaries are encoded recursively (see Section 3.2.3), they require some computation to restore the original, uncompressed form. For all datasets, except the Microsoft Academic Graph (`mag`), slightly less memory ($\approx$ 10 KiB) is required than for the processing of the uncompressed versions. For the `mag` dataset, in the current implementation a lot more memory ($\approx$ 500 KiB) is required when the compressor is used due to the same implementation issues as described in Section 3.2.1, which is not a conceptual problem, though.

5.2.3 Incremental Encoding

Just like the prefix dictionary compressor, the incremental encoder implemented as part of the configurable compressor is influenced by one important parameter, whose effect on compression and runtime performance is evaluated first. This parameter, the *delta chunk length* (d), defines the maximum number of consecutive entries that need to be evaluated to decode a single entry (see Section 3.2.2). For example, with a delta chunk length of 10, up to 9 consecutive entries can be encoded relative to their predecessor before one uncompressed string (which therefore does not depend on its predecessor) is stored.

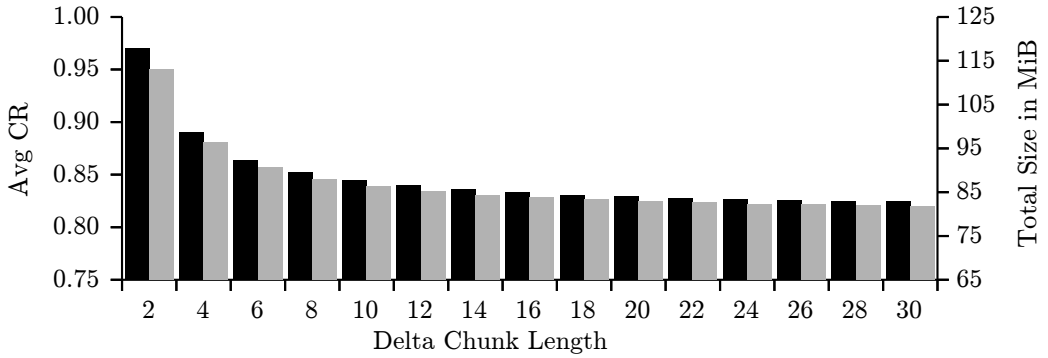


Figure 5.6: Average compression ratio (CR) of the string data (black) and the sum of the compressed string data size (gray) depending on the delta chunk length

As visualized by Figure 5.6, with larger values for the delta chunk length, the compression ratio improves. This is a consequence of less uncompressed reference strings being stored when the delta chunk length increases. As already mentioned in Section 3.2.4, the number of uncompressed strings (given that the input strings actually share a common prefix) can be estimated as $\approx \frac{n}{d-1}$ (with n being the number of input strings). The measurements as shown in Figure 5.6 support this assumption.

Increasing the delta chunk length does not negatively influence the compression time as every string has to only be compared to its predecessor, independently of the delta chunk length. The compression time does even slightly decrease for higher values as the output file becomes smaller and therefore less memory is needed and less data has to be written to the storage. As the difference between the smallest value ($d = 2$) and the highest value ($d = 30$) is less than 0.6 percent, the effect is nearly negligible, though. The same applies to the memory consumption.

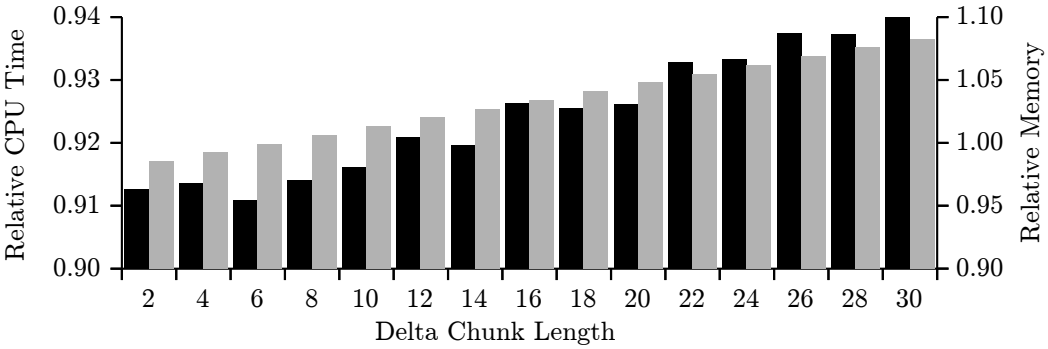


Figure 5.7: CPU time (black) and memory consumption (gray) of the incremental prefix encoder (both relative to the uncompressed datasets) depending on the delta chunk length

In contrast, for the decompression, the time and memory consumption increases linearly with the delta chunk length (see Figure 5.7), as on average $\frac{d+1}{2}$ strings have to be kept in memory and to be decoded (assuming that the values share a common prefix). Due to the smaller size of the compressed Carbon archives, the memory consumption and the decompression time are lower compared to the uncompressed archives for small delta chunk lengths. The memory consumption surpasses the uncompressed datasets' one for $d = 8$ and the decompression time the one for the uncompressed datasets for $d = 86$.

The evaluation of the delta chunk length parameter shows that there is no optimal value to use but it depends on how important the decoding speed and memory consumption is compared to the compression ratio. This decision has to be made for each system individually that uses the algorithm. This can also be seen as an advantage as it allows to tune the system to the users needs.

For that reason, statements on the compression ratio always require the delta chunk length to be specified. But one general observation is that the delta chunk length approach achieves better compression ratios (for $d \geq 6$) than the prefix dictionary method because it does not need to store a separate dictionary. For example, using $d = 30$, the `github` dataset's string data can be compressed to $\approx 37.5\%$ (prefix dictionary: $\approx 47\%$). As both approaches exploit the common prefixes of strings, the datasets that achieve the highest compression ratios are the same as for the prefix dictionary approach (see Figure 5.8).

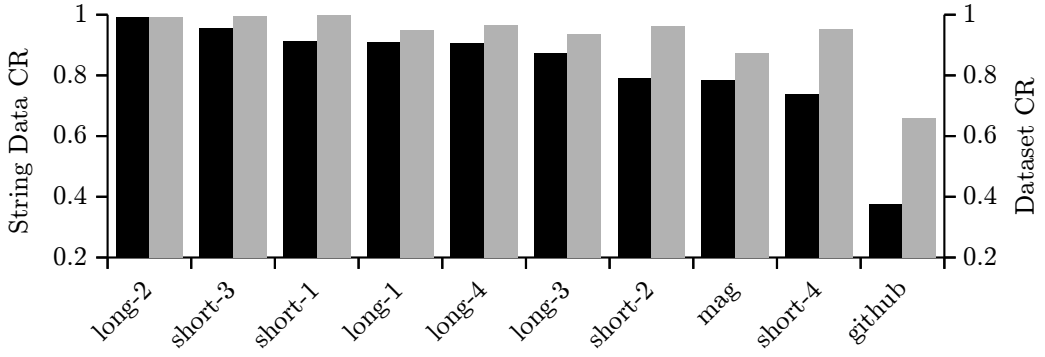


Figure 5.8: Compression ratios (CR) of the string data only (black) and the whole datasets (gray) when using the incremental prefix compressor with a delta chunk length of 30, ordered by string data compression ratio

The configurable compressor does not only include the option to compress the prefix incrementally, but also the suffix at the same time. Enabling both does only slightly influence the time required for compression or decompression (about $\pm 1\%$), because the only change is an additional memory compare/copy operation of the size of the common suffix. As the strings are sorted by their prefix, the compression effect is smaller even

if there are a lot of common suffixes and the additional overhead (at least one byte per entry) can lead to much worse compression ratios compared to the datasets with only the prefix incrementally encoded. The results for $d = 30$ visualized in Figure 5.9 show that the effect of incrementally encoding both ends of the strings ranges from an overhead of up to $\approx 18\%$ (`short-3`) to an improved compression ratio of $\approx 28.5\%$ (-8.5 percentage points) for the `github` dataset's strings.

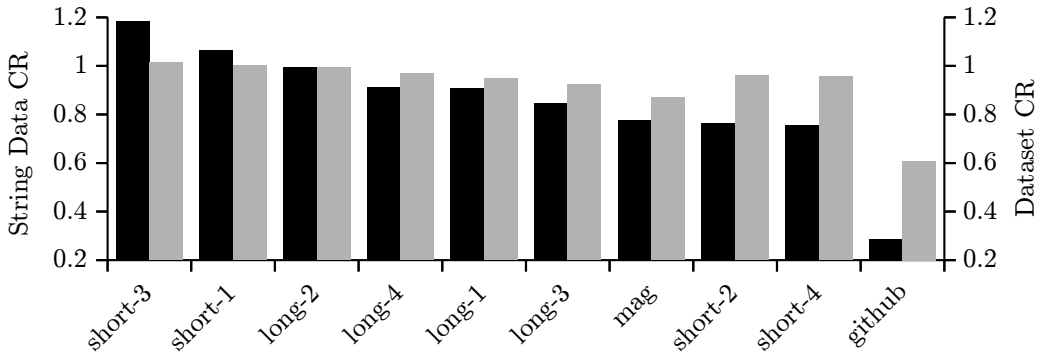


Figure 5.9: Compression ratios (CR) of the string data only (black) and the whole datasets (gray) when using the incremental prefix & suffix compressor with a delta chunk length of 30, ordered by string data compression ratio

5.2.4 Combined Algorithms

While Section 5.2.1 to Section 5.2.3 only benchmarked individual components of the configurable compressor, this section focuses on combining the approaches as described in Section 3.3.

The configurable compressor allows different combinations of the parameters to be used, but this section will focus on applying all three approaches benchmarked earlier in this chapter together, which leaves one configuration: using a prefix dictionary, encoding suffixes incrementally and apply Huffman coding to the remaining part of the strings. The entries are sorted from their end to make the incremental encoding as effective as possible. This does not influence the prefix dictionary as the order of entries does not matter for that approach. As a result of Section 5.2.2, a dictionary minimum support of 4 is used and to make the compression ratios comparable to ones shown in Figure 5.8 (Section 5.2.3), a delta chunk length of 30 is used for the incremental encoder.

This leads to significantly better compression ratios for most of the datasets compared to only using a single approach, especially for the larger ones as the overhead introduced by each approach is smaller compared to the datasets' sizes (see Figure 5.10). The `github` dataset's string data is compressed down to 19% of its original size (-9 percentage points

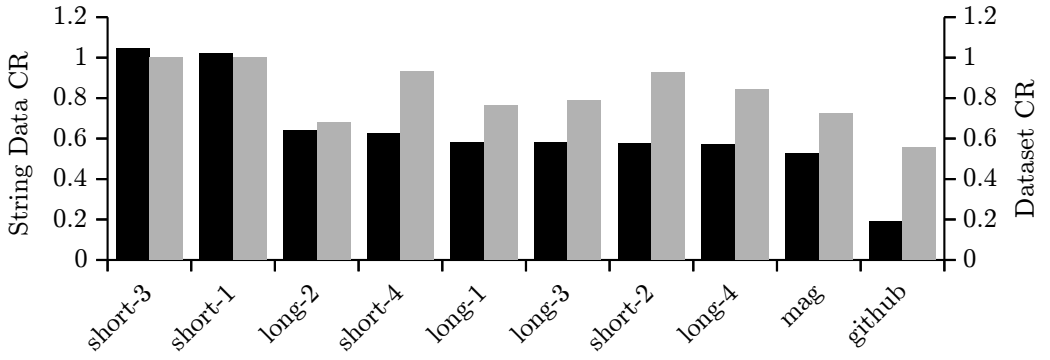


Figure 5.10: Compression ratios (CR) of the string data only (black) and the whole datasets (gray) when combining prefix dictionary (minimum support of 4), incremental suffix encoding ($d = 30$) and Huffman coding, ordered by string data compression ratio

compared to incremental prefix and suffix compression) which leads to a compression ratio of $\approx 55\%$ for the whole Carbon archive. Also, the Microsoft Academic Graph's (`mag`) string data gets far better compressed compared to its best-compressing single approach (Huffman compression), achieving a compression ratio of $\approx 53\%$ instead of $\approx 67\%$ (whole archive file: $\approx 72\%$ vs. $\approx 81\%$). The only two datasets that are increased in size when applying the compression compared to their uncompressed versions (`short-3`: $\approx +4\%$, `short-1`: $\approx +1\%$) contain very little string data and therefore the overhead of the Huffman code table and the prefix dictionary have a greater impact than the savings achieved by the compression algorithms.

The computational and the memory overhead are close to the overheads measured in Section 5.2.1 to Section 5.2.3 combined, which was to be expected as the algorithms are applied one after the other. The same can be observed for the decompression process: the sum of the (partly negative) overheads of the three approaches is close to the overhead of the combined approach.

5.3 Key-based Column Compression Optimization

The previous section, Section 5.2, evaluated the compression and runtime performance of the individual components of the configurable compressor as well as their combined application. The configurations were, however, always applied to the string table as a whole, in contrast to the key-based application as described in Section 3.4. The reason for that is, that – as explained in Section 4.1 – defining a suitable configuration for each column manually is not realistic for real-world applications. Instead, the configuration has to be determined by the system itself, using one of the configuration optimizers introduced in Section 4.3.1, Section 4.3.2 and Section 4.3.3.

5.3.1 Brute Force Optimizer

The brute force optimizer is guaranteed to find the best configuration for each individual column, so it will be evaluated first to serve as a baseline for Section 5.3.2 and Section 5.3.4.

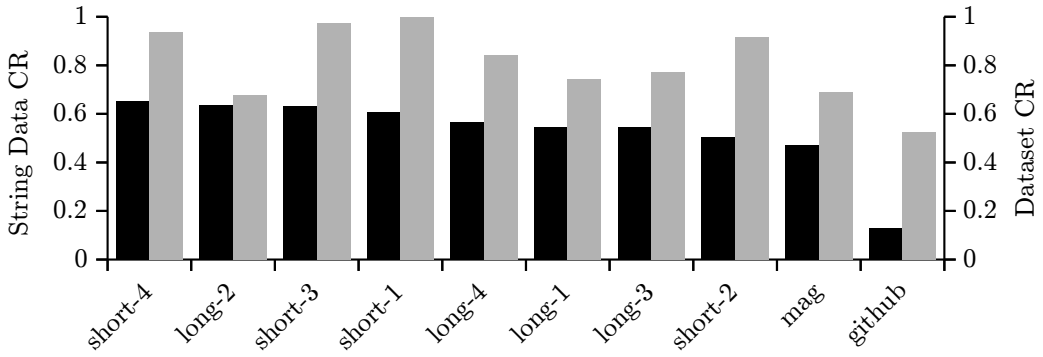


Figure 5.11: Compression ratios (CR) of the string data only (black) and the whole datasets (gray) when applying key-based compression using the brute force optimizer, ordered by string data compression ratio

Two main observations can be made based on the results visualized in Figure 5.11:

- The compression ratios are better than for each single configuration evaluated in Section 5.2, for all datasets. The `github` dataset, also being the one getting compressed better than the others in the previous benchmarks, achieves a compression ratio of $\approx 52\%$ as the string data is compressed down to $\approx 13\%$ of its original size. Even the worst string data compression ratio is still $\approx 65\%$.
- No dataset becomes larger than its non-compressed version after applying the compression. That is to be expected, because a configuration not performing any compression is also evaluated by the optimizer and in case the data cannot be compressed without creating some overhead, the configuration that does not apply any compression will be selected automatically. The only overhead introduced is the meta information required by the proxy compressor as described in Section 3.4. In case only none-compressing configurations are selected (so no space can be saved), all columns will be merged together and lead to an overhead of only 3 bytes for the whole dataset in the worst-case scenario.

Also, there is no configuration that is chosen much more often than others, which supports the assumption that different compressors should be used for different inputs and none of them is generally superior to the others. On the `short-2` dataset, for example, the optimizer chooses 6 different configurations for 12 columns.

In Section 4.3.1, it was already expected that the brute force approach would add a lot of overhead. That is confirmed by the measurements: using column-wise compression with the brute force optimizer adds an overhead of about 22% compared to the process with compression disabled. The actual compression, however, only introduces an overhead of ≈ 2%, so the optimizer itself is responsible for the remaining 20%. The largest overhead introduced by the optimizer is for the Microsoft Academic Graph (`mag`) with ≈ 150%, most likely because of the application of Huffman compression to many longer texts. The amount of memory required during the optimization phase is below the memory required when performing the compression, so the optimizer does not increase the peak memory required.

While the optimizer does not directly influence the decompression performance, the configurations it selects do so. Compared to compressing the whole dataset with a single compression configuration instead of a key-based compression (see Section 5.2.4), the decompression is accelerated by 6.5% on average. The only outliers are the `github` dataset that decompresses ≈ 7% slower and the `mag` dataset whose decompression time is reduced by ≈ 22%.

5.3.2 Cost Model Optimizer

As it was expected in advance and confirmed in Section 5.3.1, the brute force optimizer introduces a lot of overhead. To circumvent this, the cost-based optimizer was introduced in Section 4.3.2 as a faster alternative. While it does not guarantee to find the optimal configurations, it should require much less time for selecting a good one.

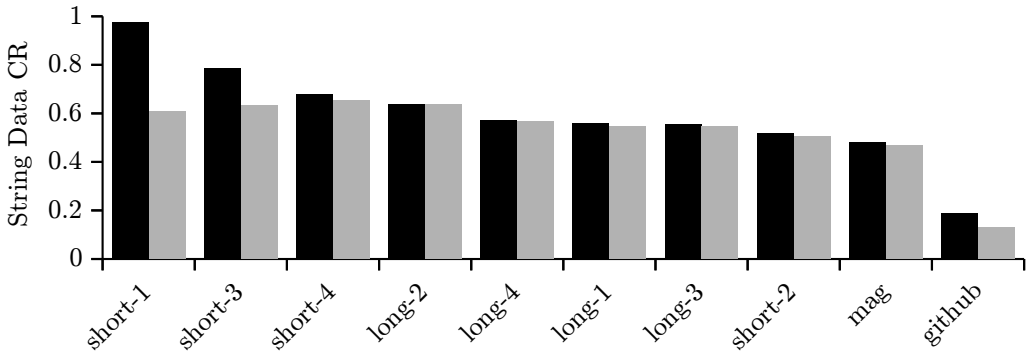


Figure 5.12: Compression ratios (CR) of the string data using the cost-based optimizer (black) and the brute force optimizer (gray) when applying key-based compression, ordered by the cost-based optimizer's string data compression ratio

In most cases, the cost model seems to be pretty accurate according to Figure 5.12. For the majority of datasets, the difference in the compression ratio between the brute force

optimizer and the cost-based one is very small or even zero. The two datasets where the largest differences can be observed, `short-1` and `short-3`, are very small ones – so while the difference seems pretty large, it is actually less than 1 KiB in absolute terms in both cases. For the `github` dataset, which is pretty large, the absolute difference in size between the two optimizers is ≈ 4.6 MiB (string data compression ratio 12.7% vs. 18.8%). However, the compression ratio is smaller for all datasets except one (`short-4`) compared to the combined compressor from Section 5.2.4 and for all datasets the compression ratio is less than one, which means no dataset becomes larger than the uncompressed version.

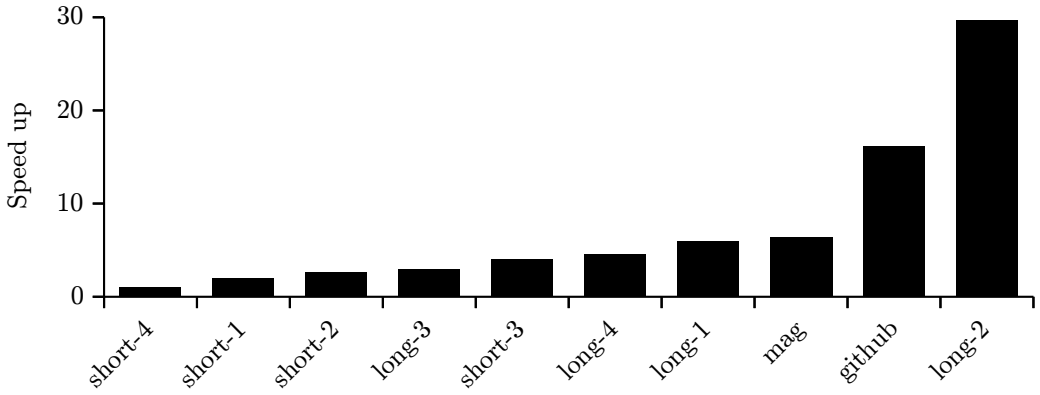


Figure 5.13: Speed up of the optimization process (without actual compression) when using the cost-based optimizer as opposed to the brute force optimizer; one means identical performance

The big advantage of the cost-based optimizer over the brute force approach can be seen in Figure 5.13: the optimizer phase is 7.5× as fast as the brute force optimizer on average, ranging from identical runtime to a 29.7× speed up. While the brute force optimizer introduced an additional overhead of ≈ 20% to the import (excluding the compression itself) as compared to the process with compression enabled, the cost-based optimizer adds only ≈ 2.7% on average.

5.3.3 Rule-based Optimizer

The brute force and cost-based optimizer do not depend on any user configuration. However, the rule-based optimizer's performance – both runtime and compression performance – fully depends on the rules defined by the user. In contrast to the parameters of the incremental encoder and the prefix dictionary compressor, the variety of rules that can be defined is much larger. It is also impossible to prove that a rule set is optimal in terms of the regular expressions used for matching.

Regarding the compression ratio, it will (theoretically) always be possible to define a rule set returning the optimal configuration by constructing the regular expression as `(entry0|entry1|...|entryN)` which will guarantee to match exactly the entry set it should.

It therefore does not make sense to benchmark the datasets with an example rule set as it would actually benchmark the rule set, not the optimizer.

5.3.4 Impact of Sampling Strategies

In Section 4.2 the idea of using only a subset of the entries for each column to determine which compression configuration to use was described. Using only a small, yet representative entry set should reduce the workload of the optimizer, resulting in a better runtime.

The sampling is based on blocks instead of individual entries and therefore depends on two parameters: the block length and the block count. Both are evaluated by using the brute force optimizer with sampling enabled, setting one of the two parameters to a fixed value and varying the other one.

With more blocks (but a fixed block length of 16), the compression ratio improves (see Figure 5.14), as entries from more parts of the dataset are more likely to represent the full set of entries correctly. While the effect is measurable, the difference in the compression ratio between the highest (32) and the lowest (4) value used is only about 0.1%. The impact on the runtime is much higher ($\approx$ 3.5 seconds vs. $\approx$ 60 seconds), as it increases close-to-exponential in the beginning, most likely due to CPU caching issues, as `perf` [Linb] reports increasing cache misses. The reason the curve flattens for higher values is that the number of entries used for the sampling process exceeds the amount of entries (per column) of many datasets, in which case all entries will be used but not more.

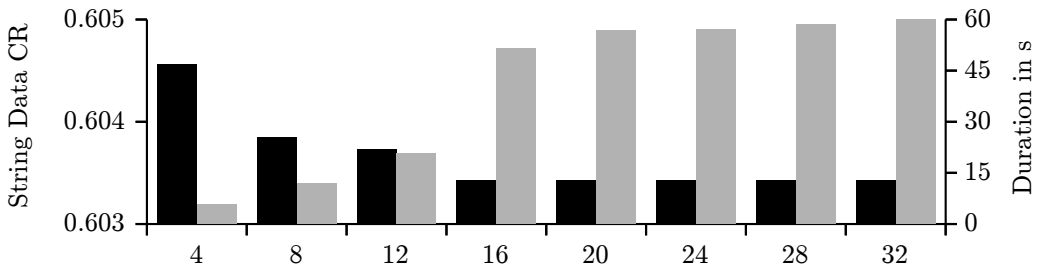


Figure 5.14: Influence of the sampling block count on the compression ratio (CR) of the string data (black) and the optimizer runtime in seconds (gray)

The influence of the block length (using a fixed block count of 16) on the compression ratio is larger, but still very small with ≈ 0.3% difference between the highest (32) and the lowest (4) block length evaluated (see Figure 5.15). The impact is higher because the common prefix and suffix lengths are better represented in longer blocks and this property is important to find the best prefix and suffix compressor for the column. The runtime of the optimizer increases from ≈ 6 seconds to about a minute. It does so almost linearly, except when using 32 entries per block. This is again likely due to CPU caching issues, according to `perf`.

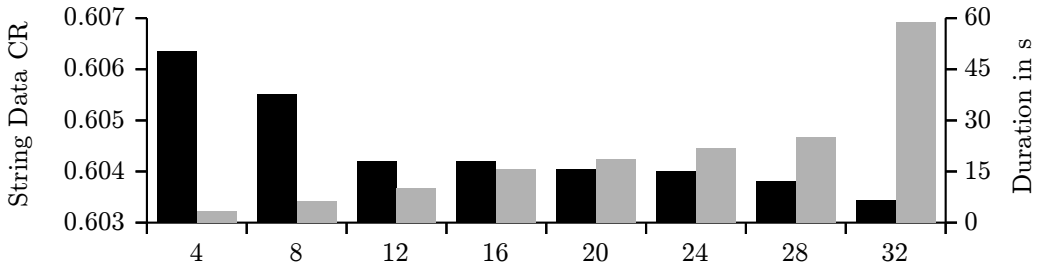


Figure 5.15: Influence of the sampling block length on the compression ratio (CR) of the string data (black) and the optimizer runtime in seconds (gray)

While the two diagrams show the results when using sampling with the brute force optimizer, the behavior of the cost-based optimizer is the same. Knowing that the influence of both parameters is very small on the compression ratio but very high on the runtime performance, small values should be used.

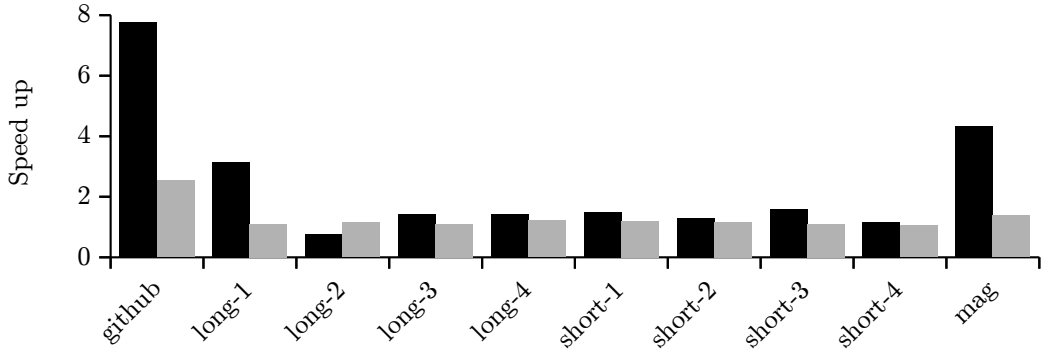


Figure 5.16: Speed up of the optimization process (without actual compression) when using sampling in the brute force optimizer (black) and the cost-based optimizer (gray) as opposed to them with sampling disabled; one means identical performance; block length: 16, block count: 16

Figure 5.16 shows that the sampling achieves a speed up for all datasets. Especially the brute force optimizer benefits from the sampling regarding runtime performance: on

average, the optimizer phase is ≈ 145% faster (taking only about 40% of the time). The larger the amount of records in the dataset is, the more effective becomes the sampling (`github`: 7.8× speed up, `mag`: 4.3×), as the amount of entries selected is not going to be higher than block length × block count (per column) and this is independent of the total number of entries. As a result, the ratio of total entries to the number of entries used with sampling enabled is higher and therefore the runtime benefit.

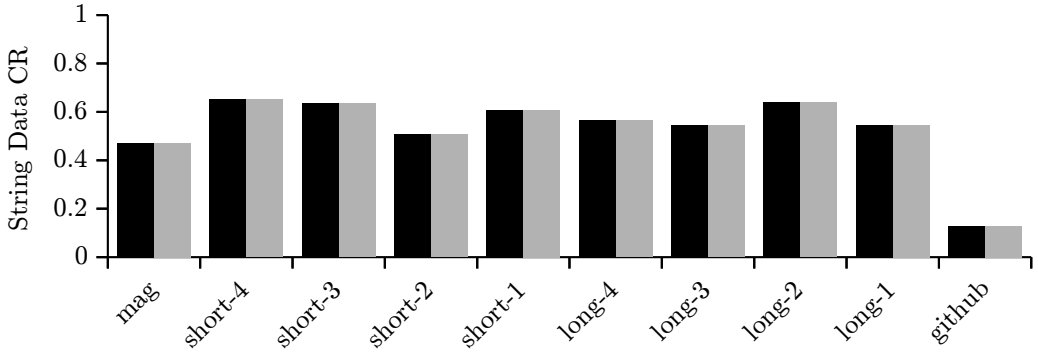


Figure 5.17: String data compression ratios (CR) when using the brute force optimizer with sampling disabled (black) and with sampling enabled (gray); block length: 16, block count: 16

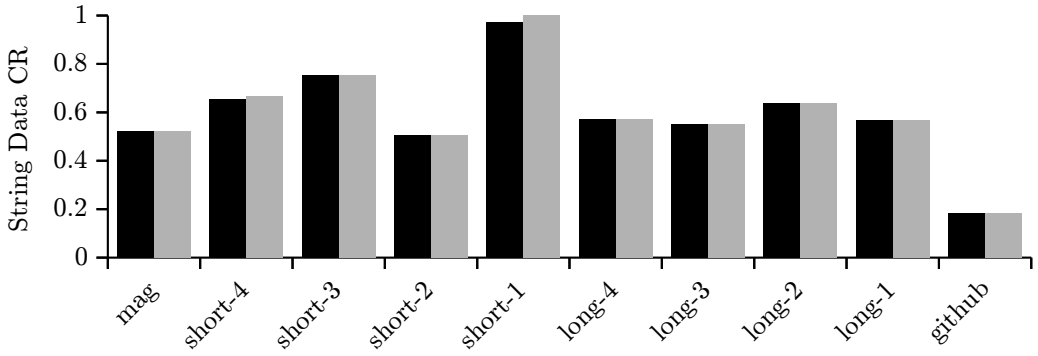


Figure 5.18: String data compression ratios (CR) when using the cost-based optimizer with sampling disabled (black) and with sampling enabled (gray); block length: 16, block count: 16

As the evaluation of the parameters has already shown, increasing the number of entries for sampling beyond a certain point does not improve the compression ratio very much. Consequently, it comes as no surprise that considering all entries in the optimizer (no sampling) does not lead to much better results compared to optimizer with sampling enabled. For the brute force optimizer in the example configuration visualized in Figure 5.17, the largest difference regarding the compression ratio for a dataset is

0.13%. For the cost-based optimizer (Figure 5.18), the average difference is ≈ 0.44%, the largest is 2.9% – on the dataset with the smallest string dictionary (`short-1`).

5.4 Real-world Comparison: Deflate and Zstandard

The algorithms discussed in Section 3.2 did not include some very common ones like the deflate algorithm [Deu96a] or Zstandard [CK18], as they do not allow random access to the individual entries and would require to decode all the data (or chunks) for reading a single entry. If used to compress individual entries, the algorithms would perform poorly and very likely introduce a lot of overhead as they are not designed to be used on very short inputs. They are therefore more suitable for the use in page-based compression as described in Section 2.3.3 if Carbon was used with an additional buffer manager. It is, however, still interesting to compare these two widespread general purpose compression algorithms to the implementation of the column compression discussed in this thesis.

To simulate the use of these algorithms on the whole string table, the strings are extracted from string dictionary and then compressed with the `zstd` and `zip` command line tool, where the latter is configured to use the deflate method. Both wrap the data in a lightweight container format, so some bytes are added on top of the actual compressors output, but the results are used for a rough comparison only.

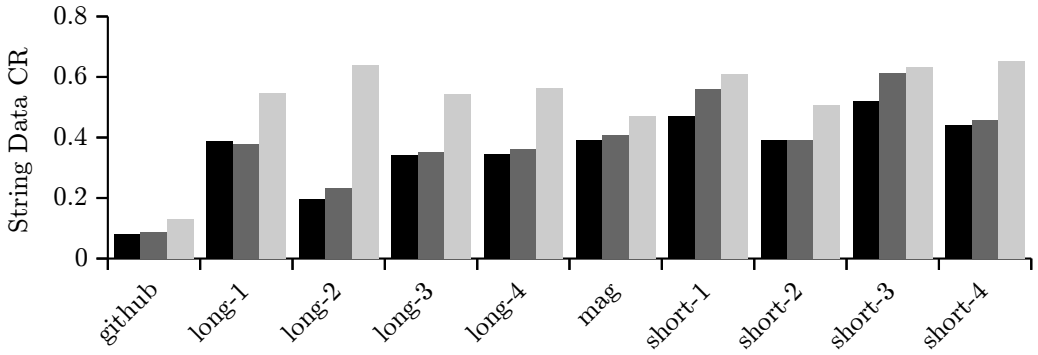


Figure 5.19: Comparison of the compression ratios (CR) for the string data achieved by Zstandard (black), deflate (dark gray) and key-based compression with the brute force optimizer (light gray)

As expected, Zstandard and deflate compress the data better than this thesis' algorithm (see Figure 5.19) as they use back-referencing over large distances instead of string directories (as in prefix dictionary) that needs to be stored separately to allow random access. Additionally, they do not just remove redundancy in the prefixes and suffixes but in all parts of the string, which pays off especially for long natural language texts. That can be observed in the diagram, too: the differences in the compression ratios

between deflate/Zstandard and our implementation are larger for the datasets that contain long natural language strings (`long-*`). For other datasets, like `github` or `mag`, the differences are much smaller.

5.5 Interpretation and Consequences

In Section 5.2 and Section 5.3, the individual parts of the implementation of column compression in the Carbon framework have been evaluated. This section interprets the results and summarizes the key findings.

One assumption made in Chapter 1 is that different inputs may be compressed best with different compression algorithms. The measured compression performances of the various algorithms in Section 5.2.1 to Section 5.2.3 for the different datasets support this assumption: which algorithm performs best, depends on the dataset, and none of the algorithms is generally superior to the other ones. The improvement of the compression ratios when applying different compressor configurations per column, e.g. using the brute force optimizer, also indicates that this assumption is correct.

The configurable compressor from Section 3.3 supports combining the algorithms because of the assumption that they could exploit different types of redundancy to reduce an input in size. This is supported by comparing the results from the individual algorithms (Section 5.2.1 to Section 5.2.3) to their combined application (Section 5.2.4) as the latter achieves better compression ratios. Additionally, the brute force optimizer in many cases returned configurations with more than one compression algorithm enabled. As that optimizer is guaranteed to find the best configuration, this is also a supporting argument for the hypothesis mentioned above.

When the brute force optimizer was introduced in Section 4.3.1, it was expected to be slow due to compressing all columns with all possible configurations. With a runtime overhead of up to 150%, this was confirmed. To speed up the optimization process, two strategies were presented in Section 4.2 and Section 4.3.2: the application of sampling and using a cost-based optimizer.

The sampling approach proved to accelerate the optimization process – especially for large datasets, where it matters most – by up to a factor of 7.8 at nearly no costs in terms of compression performance: The differences in the compression ratios achieved with and without samplings are far below one percentage point on average for the cost-based optimizer. For the brute force approach the largest difference is only ≈ 0.1 percentage point. The sampling should therefore be enabled by default.

Instead of – or in combination with – sampling, the cost-based optimizer can be used to speed up the optimization process. Even with sampling disabled, it is about 7.5× as

fast as the brute force approach on average and nearly up to 30× for one dataset. The compression performance, on the other side, is notably worse for some datasets. Because of that, speeding up the optimization process through sampling should be preferred over the cost-based approach. If the runtime performance is still not acceptable, both approaches can be combined.

In comparison to general purpose compression algorithms (Zstandard and deflate) that are not suitable for compressing a Carbon archive's string table (see Section 5.4), the compression ratios achieved by this thesis' implementation are worse. However, for datasets containing no or few long natural language text entries, the implementation can keep up pretty well.

The performance overhead introduced by the application of the actual compression algorithms is about 2% on average. As the Carbon archives are designed to be rarely written and updated, but to be read very often, that should be acceptable. However, the decompression performance is more important. The datasets compressed with key-based compression using the brute force optimizer can be converted back to JSON format $\approx$ 6.5% faster on average, but there is one outlier where the decompression time increased by $\approx$ 13%. These numbers may change in favor of compression on systems with lower I/O performance, e.g. when accessed through network storage systems, as less data has to be read and transferred. Also, for datasets that exceed the systems memory size, larger parts of the string table can be kept in memory when it is compressed. This can massively accelerate read operations. Due to issues with the tools of the Carbon framework when importing large datasets, this could not be evaluated.

6 Conclusion and Future Work

Conclusion

Compression techniques are used in databases to reduce storage costs and to increase their performance. Therefore, this approach was transferred to a column-based storage format for semi-structured data, namely the Carbon framework [Pin19a], to analyze if and how it can benefit from the application of compression techniques.

Due to the design of the framework, compression is applied to the string table of the datasets. As this table needs to provide random access, algorithms that fulfill this requirement have been selected and implemented: incremental encoding, prefix dictionary compression and Huffman coding. Each algorithm performs well in terms of the compression it achieves on certain datasets, but no algorithm performed better than an other one on all datasets that were used in the evaluation. The algorithms have also been chosen because they address different types of redundancy in the data, so they can be combined to improve the compression.

In order to achieve the best compression, both aspects have to be considered: algorithms may need to be combined and the right algorithms have to be chosen for a given input. As a consequence, a configurable compressor has been developed that allows to choose, which algorithms to apply and to combine. As a dataset may contain different types of (string) entries, the decision which algorithms to use should not be performed on dataset-level but on a more fine-grained basis, so the strings of a dataset are grouped according to the column (JSON object key) they origin from and for each column a suitable combination of compression techniques can be used.

Because configuring the compression for each column manually would require the user to spend a lot of time and also requires the user to know the different compression

algorithms very well, the framework should decide on its own, which configuration to use. It thereby provides a better user experience and potentially also an improved compression performance. From the two self-driving approaches ("optimizers") that have been implemented, the brute force optimizer provided better results compared to the cost-based one, but it also takes much longer to find a configuration. By sampling the input strings, however, this drawback can be mitigated while nearly no compression performance needs to be sacrificed. For that reason, the brute force optimizer with sampling enabled should be used by default and only in scenarios where the runtime performance is so important that the brute force approach is unacceptable, the cost-based optimizer should be considered. For repeated imports of similar datasets, the configuration cache should be used, which further mitigates the drawbacks of the brute force approach for such use cases. If the user wants to change the behavior of the optimizers for any reason, he or she can still take over control by specifying what configurations to apply using the rule-based optimizer.

The goal of saving storage space was reached with the implementation being able to compress the string data of certain datasets down to about 12%, leading to about 50% savings for the whole dataset. On average, the string data was compressed down to $\approx 53\%$, saving about 20% of storage space for the dataset. The second goal of applying compression, increasing the read performance, was mostly reached: on average, the datasets could be converted back to JSON 6.5% faster as compared to uncompressed datasets. On the other hand, there were outliers with up to 13% longer processing times. The compression may become more effective for datasets that are frequently queried and do not fit into memory, which would render the operating systems filesystem cache less useful, but due to compression, larger parts of the datasets could be kept in memory, which could help avoiding disk accesses and increase the read performance.

As the techniques presented in this thesis can reduce the amount of storage required for Carbon archives while accelerating read operations at the same time, the integration into the official Carbon framework should definitely be considered.

Future Work

While the results look promising so far, there is still room for improvement regarding different aspects of the algorithms that have been implemented:

- Although the compression is working very well for shorter strings, the compression of longer texts could be improved, e.g. by using back-referencing within longer texts or by applying dictionary compression not only to prefixes but also within the entries.

- The incremental encoder could be improved by encoding the strings incrementally based on their successor instead of their predecessor. That would require minor changes to the compression framework, but using this technique the additional reference to the previous entry could be removed and the offset encoded in the string table entry's header could be used instead. That would save 1-2 bytes per entry.

- The Huffman compressor currently uses a naïve tree-based decoding algorithm which is pretty slow. Faster and more memory efficient ones exist [HCYLYF99] and should be used to increase the read performance on compressed datasets.

- The cost model could be improved and thereby the cost-based optimizer compression ratios. To achieve that, the behavior of the compressors has to be modeled more accurately without making it too computationally expensive. In this case, it could be used as the default instead of the brute force optimizer and further reduce the key-based compression overhead.

- The runtime of the brute force optimizer could be improved by introducing an early out strategy like for the rule-based compressor. Currently, when a compressor configuration is evaluated, the strings are compressed one after the other to a chunk of memory until all strings (or the sampled ones) have been compressed. Instead of always compressing all strings, optimizer could stop once the size of the resulting memory buffer becomes larger than the smallest configuration found so far.

Bibliography

[AKS16] Aleksander Alekseev, Alexander Korotkov, and Teodor Sigaev. ZSON. https://github.com/postgrespro/zson, 2016. (cited on Page 33)

[AKSV15] Jyrki Alakuijala, Evgenii Kliuchnikov, Zoltán Szabadka, and Lode Vandevenne. Comparison of Brotli, Deflate, Zopfli, LZMA, LZHAM and Bzip2 Compression Algorithms. 2015. (cited on Page 23)

[Amaa] Amazon Web Services, Inc. Amazon Redshift: Amazon Redshift and PostgreSQL. https://docs.aws.amazon.com/redshift/latest/dg/c_redshift-and-postgres-sql.html. (cited on Page 34)

[Amab] Amazon Web Services, Inc. Amazon Redshift: Compression Encodings. https://docs.amazonaws.cn/en_us/redshift/latest/dg/c_Compression_encodings.html. (cited on Page 11 and 33)

[Amac] Amazon Web Services, Inc. Amazon Redshift: Mostly Encoding. https://docs.aws.amazon.com/redshift/latest/dg/c_MostlyN_encoding.html. (cited on Page 34)

[AMH08] Daniel J. Abadi, Samuel R. Madden, and Nabil Hachem. Column-stores vs. row-stores: how different are they really? In *Proceedings of the 2008 ACM SIGMOD international conference on Management of data - SIGMOD '08*. ACM Press, 2008. (cited on Page 10 and 16)

[Arc] Arch Linux Community. Arch Linux Wiki: pacman. https://wiki.archlinux.org/index.php/Pacman. (cited on Page 22)

[AS16] Jyrki Alakuijala and Zoltan Szabadka. Brotli Compressed Data Format. RFC 7932, July 2016. (cited on Page 23)

[BH13] Carsten Bormann and Paul E. Hoffman. Concise Binary Object Representation (CBOR). RFC 7049, October 2013. (cited on Page 18)

[Bou97] Thomas Boutell. PNG (Portable Network Graphics) Specification Version 1.0. RFC 2083, March 1997. (cited on Page 21)

[BPSM+08] Tim Bray, Jean Paoli, C. M. Sperberg-McQueen, Eve Maler, and François Yergeau. Extensible Markup Language (XML). https://www.w3.org/TR/xml/, 11 2008. (cited on Page 10)

[Bra14] Tim Bray. The JavaScript Object Notation (JSON) Data Interchange Format. RFC 7159, March 2014. (cited on Page 10, 15, 33, 53, and 65)

[Bre16] Bevin Brett. Memory Performance in a Nutshell. https://software.intel.com/en-us/articles/memory-performance-in-a-nutshell, 06 2016. (cited on Page 28)

[Bro19] David Broneske. Main-Memory Database Management Systems. http://www.dbse.ovgu.de/Lehre/ATDB/_/MainMemoryDBMS.pdf, 04 2019. (cited on Page 28)

[BSTW86] Jon Louis Bentley, Daniel D. Sleator, Robert E. Tarjan, and Victor K. Wei. A locally adaptive data compression scheme. *Communications of the ACM*, 29(4):320–330, mar 1986. (cited on Page 20)

[BW94] Michael Burrows and David J. Wheeler. A block-sorting lossless data compression algorithm. 1994. (cited on Page 20)

[Cam19] Gabriel Campero. AI Techniques for Database Management(AI4DB). http://www.dbse.ovgu.de/Lehre/ATDB/_/5_ai-2.pdf, 06 2019. (cited on Page 11 and 29)

[Chr18] Gregg Christman. Reduce the Storage Requirements of SecureFiles Data with Advanced LOB Compression. https://blogs.oracle.com/dbstorage/reduce-the-storage-requirements-of-securefiles-data-with-advanced-lob-compression, 2018. (cited on Page 32)

[CK18] Yann Collet and Murray Kucherawy. Zstandard Compression and the application/zstd Media Type. RFC 8478, October 2018. (cited on Page 21, 23, 32, 34, and 84)

[Cola] Yann Collet. LZ4 – Extremely fast compression. https://lz4.github.io/lz4/. (cited on Page 23, 31, 32, and 34)

[Colb] Yann Collet. LZ4 Block Format Description. https://github.com/lz4/lz4/blob/dev/doc/lz4_Block_format.md. (cited on Page 23)

[Col13] Yann Collet. Finite State Entropy – A new breed of entropy coder. http://fastcompression.blogspot.com/2013/12/finite-state-entropy-new-breed-of.html, December 2013. (cited on Page 20)

[Deu96a] L. Peter Deutsch. DEFLATE Compressed Data Format Specification version 1.3. RFC 1951, May 1996. (cited on Page 21, 32, and 84)

[Deu96b] L. Peter Deutsch. GZIP file format specification version 4.3. RFC 1952, May 1996. (cited on Page 21)

[Dud13] Jarek Duda. Asymmetric numeral systems: entropy coding combining speed of Huffman coding with compression rate of arithmetic coding. 2013. (cited on Page 20)

[Fac] Facebook, Inc. Zstandard – Fast real-time compression algorithm. https://github.com/facebook/zstd/. (cited on Page 24)

[Fas] FasterXML, LLC. Jackson Project Home. https://github.com/FasterXML/jackson. (cited on Page 18)

[Fas13] FasterXML, LLC. Efficient JSON-compatible binary format: "Smile". https://github.com/FasterXML/smile-format-specification/blob/master/smile-specification.md, 05 2013. (cited on Page 18)

[Fas17] FasterXML, LLC. Smile Format: design goals. https://github.com/FasterXML/smile-format-specification/blob/master/smile-design-goals.md, 2017. (cited on Page 18)

[Fre] Free Software Foundation, Inc. GNU tar 1.32: Basic Tar Format. https://www.gnu.org/software/tar/manual/html_node/Standard.html. (cited on Page 22)

[Fre60] Edward Fredkin. Trie memory. *Communications of the ACM*, 3(9):490–499, aug 1960. (cited on Page 40)

[Fur08] Sadayuki Furuhashi. MessagePack: It's like JSON. but fast and small. https://msgpack.org/, 2008. (cited on Page 18)

[Gooa] Google LLC. Brotli compression format. https://github.com/google/brotli. (cited on Page 23)

[Goob] Google LLC. Snappy, a fast compressor/decompressor. https://github.com/google/snappy. (cited on Page 23, 31, 32, and 34)

[Gor] Assaf Gordon. GNU Time. https://www.gnu.org/software/time/. (cited on Page 68)

[Goy] Jan Goyvaerts. RegexBuddy. https://www.regexbuddy.com. (cited on Page 60)

[GRS98] Jonathan Goldstein, Raghu Ramakrishnan, and Uri Shaft. Compressing Relations and Indexes. In *Proceedings of the Fourteenth International Conference on Data Engineering*, ICDE '98, page 370–379, USA, 1998. IEEE Computer Society. (cited on Page 25)

[HCYLYF99] Chen Hong-Chung, Wang Yue-Li, and Lan Yu-Feng. A memory-efficient and fast Huffman decoding algorithm. *Information Processing Letters*, 69(3):119–122, feb 1999. (cited on Page 89)

[Her16] Brule Herman. Quick Benchmark: Gzip vs Bzip2 vs LZMA vs XZ vs LZ4 vs LZO. https://catchchallenger.first-world.info/wiki/Quick_Benchmark:_Gzip_vs_Bzip2_vs_LZMA_vs_XZ_vs_LZ4_vs_LZO, 10 2016. (cited on Page 22, 23, and 28)

[Hol04] Scott Hollenbeck. Transport Layer Security Protocol Compression Methods. RFC 3749, May 2004. (cited on Page 34)

[Huf52] David Huffman. A Method for the Construction of Minimum-Redundancy Codes. *Proceedings of the IRE*, 40(9):1098–1101, sep 1952. (cited on Page 17 and 19)

[Hä05] Theo Härder. Architecture - the Layer Model and its Evolution. *Datenbank-Spektrum*, 13:45–57, 2005. (cited on Page 29 and 30)

[IBM] IBM. IBM DB2: Row compression. https://www.ibm.com/support/knowledgecenter/en/SSEPGG_9.8.0/com.ibm.db2.luw.admin.dbobj.doc/doc/c0056481.html. (cited on Page 29)

[IGN+12] S Idreos, F Groffen, N Nes, S Manegold, S Mullender, and M Kersten. Monetdb: Two decades of research in column-oriented database. *IEEE Data Engineering Bulletin*, 2012. (cited on Page 16)

[Int96] International Organization for Standardization. Information technology — Open Systems Interconnection — Remote Procedure Call (RPC). Standard ISO/IEC 11578:1996, International Organization for Standardization, Geneva, Switzerland, December 1996. (cited on Page 66)

[Int17a] International Organization for Standardization. Document management – Portable document format – Part 2: PDF 2.0. Standard ISO 32000-2:2017, International Organization for Standardization, Geneva, Switzerland, July 2017. (cited on Page 21)

[Int17b] International Organization for Standardization. SQL support for JavaScript Object Notation (JSON). Standard ISO/IEC TR 19075-6:2017(E), International Organization for Standardization, Geneva, Switzerland, March 2017. (cited on Page 33)

[Kal] Riyad Kalla. Universal Binary JSON Specification. http://ubjson.org/. (cited on Page 16 and 17)

[Knu85] Donald E. Knuth. Dynamic huffman coding. *Journal of Algorithms*, 6(2):163–180, jun 1985. (cited on Page 20)

[lGA06] Jean loup Gailly and Mark Adler. zlib Technical Details. http://www.zlib.net/zlib_tech.html, 05 2006. (cited on Page 21, 30, 31, and 34)

[Lina] Linux Kernel Organization, Inc. Kernel Wiki: BRFS – Compression. https://btrfs.wiki.kernel.org/index.php/Compression. (cited on Page 24)

[Linb] Linux Kernel Organization, Inc. Perf Wiki. https://perf.wiki.kernel.org/index.php/Main_Page. (cited on Page 81)

[LL18] Vladimir Levantovsky and Raph Levien. WOFF File Format 2.0. https://www.w3.org/TR/WOFF2/, March 2018. (cited on Page 23)

[Lou13] Phillip Lougher. Squashfs: add LZ4 compression. https://lwn.net/Articles/560015/, July 2013. (cited on Page 23)

[LR81] G. Langdon and J. Rissanen. Compression of Black-White Images with Arithmetic Coding. *IEEE Transactions on Communications*, 29(6):858–867, jun 1981. (cited on Page 20)

[Mar] MariaDB Corporation Ab. MariaDB Knowledge Base: Compressing Events to Reduce Size of the Binary Log. https://mariadb.com/kb/en/library/compressing-events-to-reduce-size-of-the-binary-log/. (cited on Page 34)

[Mar79] G. N. N. Martin. Range encoding: an algorithm for removing redundancy from a digitised message. *Proceedings of the Conference on Video and Data Recording*, July 1979. (cited on Page 20)

[Mic16a] Microsoft Corporation. Microsoft SQL Server 2017: Page Compression Implementation. https://docs.microsoft.com/en-us/sql/relational-databases/data-compression/page-compression-implementation?view=sql-server-2017, 06 2016. (cited on Page 31 and 33)

[Mic16b] Microsoft Corporation. Microsoft SQL Server 2017: Row Compression Implementation. https://docs.microsoft.com/en-us/sql/relational-databases/data-compression/row-compression-implementation?view=sql-server-2017, 2016. (cited on Page 33)

[Mon] MongoDB, Inc. JSON and BSON. https://www.mongodb.com/json-and-bson. (cited on Page 17 and 33)

[Mon09] MongoDB, Inc. BSON (Binary JSON) Serialization. http://bsonspec.org/, 2009. (cited on Page 15 and 17)

[Mon18a] MongoDB, Inc. MongoDB: Journaling. https://docs.mongodb.com/manual/core/journaling/#compression, August 2018. (cited on Page 34)

[Mon18b] MongoDB, Inc. MongoDB Wire Protocol. https://docs.mongodb.com/manual/reference/mongodb-wire-protocol/, 2018. (cited on Page 34)

[Mon18c] MongoDB, Inc. MongoDB: WiredTiger Storage Engine – Compression. https://docs.mongodb.com/manual/core/wiredtiger/#compression, 08 2018. (cited on Page 31)

[Obe] Markus Oberhumer. LZO – A real-time data compression library. http://www.oberhumer.com/opensource/lzo/. (cited on Page 31)

[Oraa] Oracle Corporation. Oracle Database Manual: Managing Tables. https://docs.oracle.com/en/database/oracle/oracle-database/18/admin/managing-tables.html. (cited on Page 31)

[Orab] Oracle Corporation. Self-Driving Database | Autonomous Database Oracle 19c. https://www.oracle.com/database/autonomous-database.html. (cited on Page 11)

[Ora18a] Oracle Corporation. MySQL 8.0 Reference Manual: 15.9.1.5 How Compression Works for InnoDB Tables. https://dev.mysql.com/doc/refman/8.0/en/innodb-compression-internals.html, 04 2018. (cited on Page 30)

[Ora18b] Oracle Corporation. MySQL 8.0 Reference Manual: 4.2.6 Connection Compression Control. https://dev.mysql.com/doc/refman/8.0/en/connection-compression-control.html, 2018. (cited on Page 34)

[Pav] Igor Pavlov. 7z format. https://www.7-zip.org/7z.html. (cited on Page 22)

[Pav15] Igor Pavlov. LZMA specification (DRAFT version). https://www.7-zip.org/a/lzma-specification.7z, 2015. (cited on Page 22)

[Pin] Marcus Pinnecke. Universal Binary JSON: Comparison to Carbon. https://protolabs.github.io/libcarbon/format-specs/alternatives/ubjson.html. (cited on Page 18)

[Pin19a] Marcus Pinnecke. A C library for creating, modifying and querying Columnar Binary JSON (Carbon) files. https://github.com/protolabs/libcarbon, 2019. (cited on Page 11, 15, and 87)

[Pin19b] Marcus Pinnecke. Backend/Database Co-Design for Rapid Webservice Prototyping. http://pinnecke.info/dl/slides/pinnecke-phd-dday.pdf, 01 2019. (cited on Page 15, 16, 17, and 65)

[Pin19c] Marcus Pinnecke. The Columnar Binary JSON Specification. Technical report, April 2019. (cited on Page 15)

[Pin19d] Marcus Pinnecke. libcarbon – Huffman Decoding. https://github.com/protolabs/libcarbon/issues/6, 03 2019. (cited on Page 11 and 17)

[PKW19] PKWARE Inc. APPNOTE.TXT - .ZIP File Format Specification. https://pkware.cachefly.net/webdocs/casestudies/APPNOTE.TXT, April 2019. (cited on Page 21)

[Pos] Postgres Professional Europe Limited. Posgtres Professional: Page-level data compression. https://postgrespro.com/roadmap/56514. (cited on Page 31)

[RD12] Juliano Rizzo and Thai Duong. The CRIME attack. https://www.ekoparty.org/archive/2012/CRIME_ekoparty2012.pdf, 2012. (cited on Page 34)

[Res15] Julian Reschke. Hypertext Transfer Protocol (HTTP) Client-Initiated Content-Encoding. RFC 7694, November 2015. (cited on Page 21)

[RHS95] Gautam Ray, Jayant R. Haritsa, and S. Seshadri. Database Compression: A Performance Enhancement Tool. In *Proceedings of 7th International Conference on Management of Data (COMAD)*, 1995. (cited on Page 28)

[Ric] Richard Yao and Matt Ahrens and Graham Perrin . OpenZFS Wiki: Performance tuning. http://open-zfs.org/wiki/Performance_tuning#LZ4_compression. (cited on Page 23)

[Ric07] Leonard Richardson. *RESTful Web Services*. O'Reilly & Associates, 1 edition, 2007. (cited on Page 10, 15, and 66)

[RL79] J. Rissanen and G. G. Langdon. Arithmetic Coding. *IBM Journal of Research and Development*, 23(2):149–162, mar 1979. (cited on Page 19)

[RPE81] Michael Rodeh, Vaughan R. Pratt, and Shimon Even. Linear Algorithm for Data Compression via String Matching. *Journal of the ACM*, 28(1):16–24, jan 1981. (cited on Page 19)

[SB18] Oksana Shadura and Brian Bockelman. Compression Update: ZSTD & ZLIB. https://indico.cern.ch/event/695984/contributions/2872933/attachments/1590457/2516802/ZSTD_and_ZLIB_Updates_-_January_20186.pdf, 2018. (cited on Page 21, 22, and 24)

[Sea16] Seagate Technology LLC. Desktop HDD Product Manual. https://www.seagate.com/www-content/product-content/barracuda-fam/desktop-hdd/barracuda-7200-14/en-us/docs/100686584v.pdf, 09 2016. (cited on Page 28)

[Sew19] Julian Seward. bzip2 and libbzip2, version 1.0.8. https://www.sourceware.org/bzip2/manual/manual.html, 07 2019. (cited on Page 31)

[SLC+19] Sahel Sharify, Alan Lu, Jin Chen, Arnamoy Bhattacharyya, Ali Hashemi, Nick Koudas, and Cristiana Amza. An Improved Dynamic Vertical Partitioning Technique for Semi-Structured Data. In *2019 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. IEEE, mar 2019. (cited on Page 11)

[SSS+15] Arnab Sinha, Zhihong Shen, Yang Song, Hao Ma, Darrin Eide, Bo-June (Paul) Hsu, and Kuansan Wang. An Overview of Microsoft Academic Service (MAS) and Applications. In *Proceedings of the 24th Inter-*

national Conference on World Wide Web - WWW '15 Companion. ACM Press, 2015. (cited on Page 15 and 66)

[Ste13] Carl Steinbach. Running PostgreSQL on Compression-enabled ZFS. https://www.citusdata.com/blog/2013/04/30/zfs-compression/, 04 2013. (cited on Page 31)

[The] The LLVM Foundation. Clang: a C language family frontend for LLVM. https://clang.llvm.org/. (cited on Page 68)

[The16] The Apache Software Foundation. Apache Cassandra: Compression. http://cassandra.apache.org/doc/latest/operating/compression.html, 2016. (cited on Page 31)

[The18] The Open Group. Base Specifications Issue 7. Standard, IEEE, 2018. (cited on Page 60)

[The19a] The Apache Software Foundation. Apache Kudu Schema Design: Column Encoding. https://kudu.apache.org/docs/schema_design.html#encoding, July 2019. (cited on Page 11 and 33)

[The19b] The PostgreSQL Global Development Group. PostgreSQL Documentation: JSON Types. https://www.postgresql.org/docs/current/datatype-json.html, 2019. (cited on Page 34)

[The19c] The PostgreSQL Global Development Group. PostgreSQL Documentation: TOAST. https://www.postgresql.org/docs/current/storage-toast.html, 2019. (cited on Page 32)

[The19d] The PostgreSQL Global Development Group. PostgreSQL Documentation: Write Ahead Log. https://www.postgresql.org/docs/current/runtime-config-wal.html, 2019. (cited on Page 33)

[Tuk09] Tukaani Project. The .xz file format Version 1.0.4. https://tukaani.org/xz/xz-file-format.txt, 08 2009. (cited on Page 22 and 31)

[Urb14] Jan Urbański. Postgres on the wire: A look at the PostgreSQL wire protocol. https://www.pgcon.org/2014/schedule/attachments/330_postgres-for-the-wire.pdf, May 2014. (cited on Page 34)

[WKHM00] Till Westmann, Donald Kossmann, Sven Helmer, and Guido Moerkotte. The implementation and performance of compressed databases. *ACM SIGMOD Record*, 29(3):55–67, sep 2000. (cited on Page 11 and 28)

[WMB99] Ian H. Witten, Alistair Moffat, and Timothy C. Bell. *Managing Gigabytes: Compressing and Indexing Documents and Images*, pages 159–161. Morgan Kaufmann, second edition, 1999. (cited on Page 26)

[WNC87] Ian H. Witten, Radford M. Neal, and John G. Cleary. Arithmetic coding for data compression. *Communications of the ACM*, 30(6):520–540, jun 1987. (cited on Page 19)

[Woo] Jason Woodward. slapt-get. https://software.jaos.org/#slapt-get. (cited on Page 22)

[XSG+15] Qiumin Xu, Huzefa Siyamwala, Mrinmoy Ghosh, Tameesh Suri, Manu Awasthi, Zvika Guz, Anahita Shayesteh, and Vijay Balakrishnan. Performance analysis of NVMe SSDs and their implication on real world databases. In *Proceedings of the 8th ACM International Systems and Storage Conference on - SYSTOR '15*. ACM Press, 2015. (cited on Page 28)

[ZHNB06] M. Zukowski, S. Heman, N. Nes, and P. Boncz. Super-Scalar RAM-CPU Cache Compression. In *22nd International Conference on Data Engineering (ICDE'06)*. IEEE, 2006. (cited on Page 25)

[ZL77] J. Ziv and A. Lempel. A universal algorithm for sequential data compression. *IEEE Transactions on Information Theory*, 23(3):337–343, may 1977. (cited on Page 19, 22, 30, and 32)

Selbstständigkeitserklärung

Hiermit erkläre ich, dass ich die vorliegende Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

Magdeburg, den 4. November 2019