Czech Technical University in Prague
Faculty of Information Technology
Department of theoretical computer science

Master's thesis

Incremental Clustering-Based Compression

Bc. Luboš Krčál

Supervisor: doc. Ing. Jan Holub, Ph.D.

6th May 2014

Acknowledgements

I would hereby like to thank to my supervisor, doc. Ing. Jan Holub, Ph.D., for introduction to this topic, relentless support and consultations during the entire time.

The biggest appreciation goes to my employers from Eccam, s.r.o., owners of the company, Ing. Libor Buš, Ph.D., and Ing. Václav Opekar, who let me work on my thesis entirely on the company premises, with endless coffee supplies.

My next thanks go to all the people who provided me with consultations or other forms of support: doc. Ing. Marcel Jiřina, Ph.D. for clustering consultations, Ing. Milan Kříž, Ing. Lukáš Cerman, Ph.D., many other colleagues from Eccam for general consultations and support, and to my parents, grandparents and uncle.

Another huge expression of my gratitude goes to Katie Wirka, M.Sc, who reviewed the entire thesis for language issues.

Last, I would like to mention Prof. Dr. Vu Duong, Joe Krachey, M.Sc., and Prof. Dieter Van Melkebeek, Ph.D., for being the most inspirational professors I have met during my master's studies.

Declaration

I hereby declare that the presented thesis is my own work and that I have cited all sources of information in accordance with the Guideline for adhering to ethical principles when elaborating an academic final thesis.

I acknowledge that my thesis is subject to the rights and obligations stipulated by the Act No. 121/2000 Coll., the Copyright Act, as amended, in particular that the Czech Technical University in Prague has the right to conclude a license agreement on the utilization of this thesis as school work under the provisions of Article 60(1) of the Act.

In Prague on 6th May 2014 .....

Czech Technical University in Prague
Faculty of Information Technology

Citation of this thesis

Krčál, Luboš. *Incremental Clustering-Based Compression.* Master's thesis. Czech Technical University in Prague, Faculty of Information Technology, 2014.

Abstract

Ever increasing data growth has given rise to many deduplication and compression systems. Most of these deduplication systems are based on several common techniques – content defined chunking, deduplication using hashing, and compression of near identical data. Although most of these systems excel with their simplicity and speed, none of those goes deeper in terms of larger scale redundancy removal. This thesis emphasizes on a proposal of a novel compression and archival system called ICBCS. Our system goes beyond standard measures for similarity detection, using extended similarity hash and incremental clustering techniques – the clustering then provides groups of sufficiently similar chunks designated for compression. Lot of effort was also put into a proof-of-concept implementation with extensive parametrization and thorough performance evaluation. ICBCS outperformed both conventional solid and separate files compression on datasets consisting of at least mildly redundant files. It has also shown that selective application of weak compressor results in better compression ratio and speed than conventional application of strong compressor.

Keywords Compression, Clustering, Deduplication, Data compression, Archival system, Similairty hash, Simhash, ICBCS, Incremental Clustering-Based Compression System

Contents

List of Figures

List of Tables

List of Algorithms

Chapter 1

Introduction

1.1 Motivation

An explosive data growth increases up both primary storage as well as backup and recovery storage costs. Many copies or small variations of the same files are being saved repeatedly to a storage device. This results in a very high redundancy, or in the worse case, duplicity in the storage system.

Recent and ongoing advancement in technology leads to ever-increasing storage capacities and decreasing prices. However there are still significant benefits in optimizing storage usage. These may be crucial in a case of any data transfer, especially for mobile devices with limited bandwidth and storage capacity.

There are many applications and settings, where data is unnecessarily repeated. For instance: e-mails often include large sections of quoted data, administrative documents have the same templates, binary file archives storing all the versions separately, telescope imagery, web pages, stock markets, etc. In many cases, most of the data is not accessed frequently and can be subject of a substantial space savings at the cost of a minor access delay.

Many deduplication schemes exist nowadays, mostly aimed at exactly these use cases. However, deduplication systems excel in dealing with exact duplicates, but fail to deal with redundancy of similar files or blocks. This is where standard data compression comes in. It is obvious though that the extreme case of reckless use of the compression on the whole dataset (solid compression) could result in a need of a long and tedious decompression to retrieve a file, or in the other extreme case of high files granularity and single-file compression in no inter-file redundancy removal.

By intelligent application of compression, we can achieve an optimal compromise: removing most of the redundancies between files, while inducing minimal decompression overhead during the file retrieval.

1.2 Problem Definition

Our goal is to design an incremental compression scheme for large collections of data – an archiver. This scheme has to allow random access to the data within the collection, so that any individual document or record can be presented to the user in its uncompressed form. Lossless compression is thereby required.

The system also has to allow new documents or records to be inserted, replaced or removed from the system. This leads to an incremental problem, where all the compression withing the system has to be dealt with on the fly, using the available information only. Extensive preprocessing of all the data is thus not possible.

Since the system operates in real-time, all the fore mentioned operations should impose a minimal time overhead, similar to a real-time deduplication or speed-oriented compression system.

1.3 The Hypothesis

The main idea is of such incremental compression system is based on a selective application of solid compression. We determine a group of records that are effectively compressible together and then compress the group. With a precise parametrization of the group sizes, records size range, clustering and grouping algorithms and similarity measures we intend to remove a significant portion of redundancy between these records.

The main problem can be divided into several subproblems. First, we have to find an optimal similarity distance that approximates the resulting compression efficiency as much as possible. Note this measure can be approximate. Second, being able to tell the similarity between single records, we need to establish an incremental clustering scheme that determines the compression groups. Third, we have to apply and setup an appropriate and fast lossless compression algorithm which can cope with ever changing set of objects to compress. Last, we must determine a scalable algorithm that ensures the cooperation of all the mentioned aspects.

1.4 Contribution

In the beginning of the thesis, large analysis of current solutions to the problem is presented. It is composed of deduplication and compression systems and compression file systems. The analysis of these systems alongside with state of the art data compression algorithms, clustering analysis algorithms and distance a similarity measures lays ground for objecting the current solutions and coming up with brand new system.

Further in the work a deduplication, compression and archival system called ICBCS is introduced. The system's design tries to overcome the shortcomings of current deduplication and compression systems by extending the redundancy removal scope to thoroughly selected clusters – sets of chunks of input files. The system assumes roles of both a compressor and a file archiver. Modular architecture allows the systems to be further adapted to various scenarios and extended easily, as well as parameterized with many different compressors, distance measures, clustering algorithms, etc.

Last, it is demonstrated that ICBCS successfully overcomes the drawbacks of standard compressors and deduplication and compression systems, especially of wide-range redundancy removal. Through extensive parametrization a reference setup of ICBCS was established and subsequently tested. The overall time overhead induced by the system was much smaller than that of high precision compressors, while providing much better compression ratios.

1.5 Organisation of the Thesis

The thesis is structured as follows: First, a state of the art of deduplication and compression systems is summarized in Chapter 2. This chapter is further divided to survey on deduplication systems (2.3), data compression (2.5), distances and similarity measures (2.6) and clustering analysis (2.7). The next Chapter 3 describes the compression and archival system proposal called ICBCS and its proof of concept implementation, followed by a its performance evaluations and extensive testing in Chapter 4. The results of the thesis are concluded in Chapter 5.

Chapter 2

State of the Art

2.1 Current Solutions

There are many techniques to reduce the redundancy of data. The core method is called *data compression*. Data compression only removes redundancy within a single object (e.g. file, document, block) and generally has a significant effectiveness on textual files, reducing the size by factors of two to six [57].

A specific case of a redundancy removal over a set of objects, where only exact duplicates are removed, is called *data deduplication*. In the most usual case, data deduplication operates over *blocks* (of fixed size) or *chunks* (blocks of dynamic size), meaning all files in the system are first split into blocks. Sometimes the definition of data deduplication is extended to include, for example, delta-compression between similar blocks [112]. Data deduplication on a block level is the most employed kind of redundancy removal over a set of objects.

Delta encoding is a method of redundancy removal of one object relatively to another object. Delta encoding is very effective on files with only minor differences. The most known application of delta encoding together with copy-on-write form of deduplication is in version control systems, such as Subversion and GIT.

There is a vast variety of applications and environments where these techniques can be used, which implies there is no universal solution that works across all the problems. This is why many solutions aim to be *scalable*, since such are more likely to adapt to more scenarios. Some of the measures of the techniques can be its *effectiveness*ICBCSthe ability to reduce the data redundancy, and *efficiency* – the amount of resources needed to do so.

According to [65], the deduplication design space can be described by three dimensions as seen in Figure 2.1. The first two dimensions are of less interest in this work. The *place dimension* determines the location of the deduplication process, as well as whether the process is run on a single node (client), on a special purpose dedicated system, or on multiple nodes (e.g. disk arrays). The *timing dimension* distinguishes between in-band (synchronous, executes before writing the data to the storage) or out-of-band (asynchronous, executes after writing to the storage) deduplication schemes. Last, the *algorithm dimensions* determines the granularity of the deduplication (whole-file hashing, fixed size block hashing, variable size block hashing) and the presence of inter-object compression.

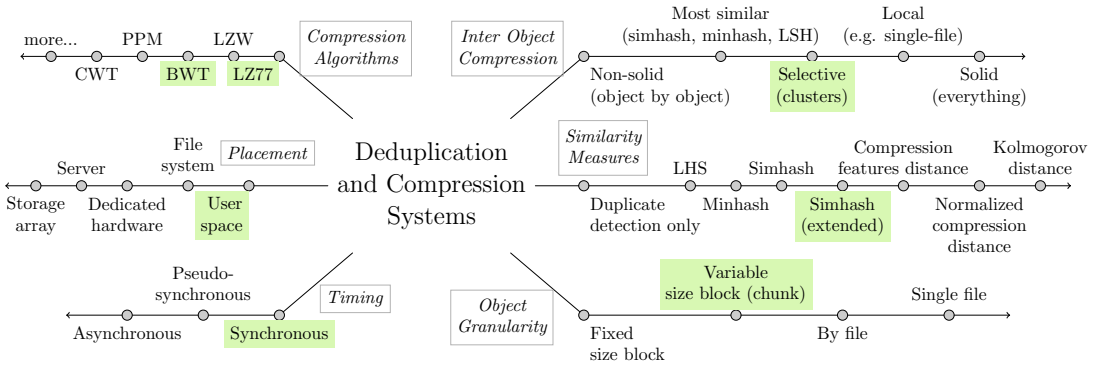


Figure 2.1: The six dimensions of compression and deduplication systems design space. Inspired by [65]. The placement of ICBCS, the system proposed in this thesis, is marked with light green.

2.1.1 Single-File Compression

In *single-file compression* (also known as *whole-file compression*) every file is compressed individually. This is the simplest form of granularity, but not the smallest. There is no inter-file redundancy removal. Also, using single-file compression does not even guarantee intra-file redundancy removal, e.g. in case of large files with a limited size window compression.

Single-file compression scales well with a large number of files and is easily applicable to transfers of individual files. It is also the most frequently used granularity in file system compressions. See Section 2.2 for details on compression file systems. Windows' `zip` works this way. The combination `gzip + tar` (`.gz.tar`) is also a single-file compression, although it is rarely used.

2.1.2 Solid Compression

The term *solid compression* refers to the process of compressing all the objects (files or blocks) together. It has a significant potential to detect redundancy across multiple or all the files, thus potentially minimizing the compression ratio, however in practice, that is not the usual case. Due to resource limitations, it tends to remove redundancy only in files that are close to each other, e.g. when compressed by stream algorithms.

Another major drawback of this approach is its poor scaling with large data sets. To access a single file, it may be necessary to decompress the entire collection.

The most known example of solid compression is a Linux-originated combination of `tar + gzip` (`.tar.gz`), PAQ compression algorithms, etc. One solution that significantly increases the span for redundancy detection is the Long Range ZIP [53].

2.1.3 Block-Based Deduplication

Every file is divided into blocks starting at the beginning of every file. These blocks cover the entire set of files, and the deduplication algorithm operates with these blocks only.

Block-based deduplication can operate either with fixed-size blocks or with variable-size blocks. In the case of *fixed-size* blocks, a signature of these blocks is usually calculated using some *strong hash function*, such as SHA1 or MD5.

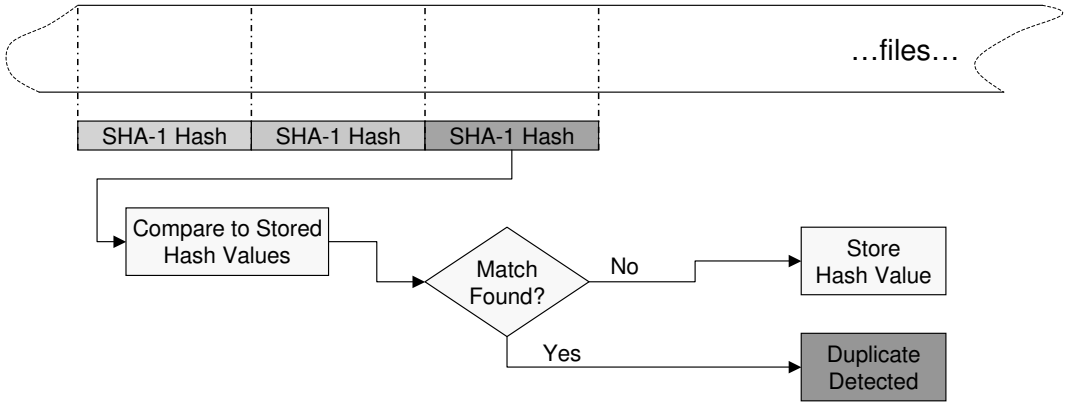


Figure 2.2: Fixed-size blocks deduplication scheme. Each fixed-size file block is hashed. Resulting hash value is compared to prior values to find duplicate blocks. Figure taken from [101].

More sophisticated approaches use *sliding window* (rolling checksum or hash) method [101, 10, 75]. *Variable-length* (content-defined) block sizes rely on fingerprinting methods to determine the block boundaries which are resistant to shifts. One of the basic methods to do it uses Rabin fingerprinting [84]. The sliding window computes the Rabin fingerprint. If the fingerprint matches a certain predetermined value (reaches a breakpoint), the current window position is marked as the block boundary. A hash of this block is then used for the deduplication. This method is very successful in identifying similar blocks irrespective of their offset within a file [112]. A simple scheme of this deduplication method is depicted in Figure 2.3.

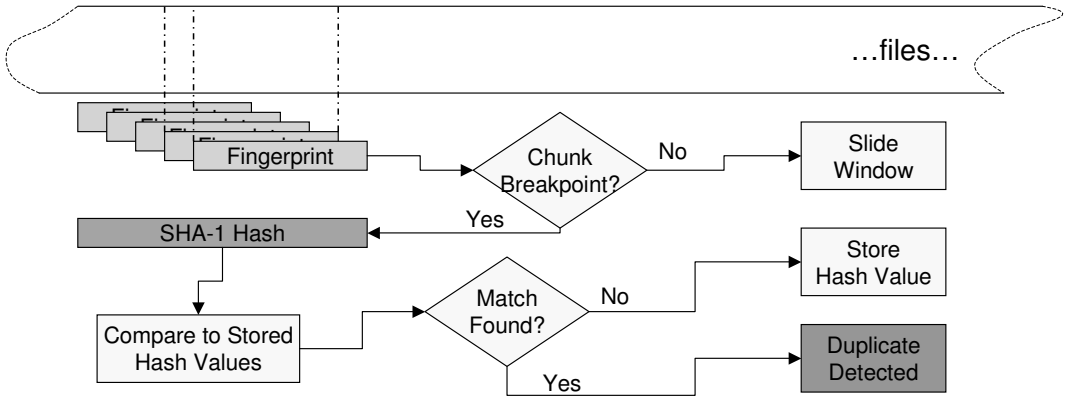


Figure 2.3: Variable-size blocks deduplication scheme. To determine the blocks, a sliding window is used to produce fingerprints. When those fingerprints match a certain value, that windows is marked as a breakpoint. Figure taken from [101].

There are further improvements of the fingerprints method. One improvement called *super-fingerprints* coalesces multiple fingerprints (chunk fingerprints) into a super-fingerprint (file fingerprint) indicating high similarity between files [11].

Note that block sizes often restrict the overall memory consumption in case of a compression applied to these blocks, e.g. many phrase-based compressions' dictionary sizes are depended on the input data size. Here, the size has an upper bound – block size. In case of a deduplication only, this is usually not a concern since the hash is computed in constant memory, often at the same time the boundaries are determined.

2.2 Deduplication and Compression File Systems

Many compression file systems have been developed and deployed. A file system is only a virtual wrapper for the compression system that potentially uses a physical device.

Several major proprietary file systems are integrated directly into operating systems, some are implementing as a module (e.g. Linux kernel module), other, usually smaller project, use the *Filesystem in Userspace – FUSE* [99] mechanism to implement a virtual file system for Linux.

Widespread file system *NTFS – New Technology File System* supports a fixed-size block compression using LZNT1 (variation of LZ77). The files are split into blocks of 64 kB. Another technique used in NTFS is that sparse files are reduced to only occupy a physical space worth of non-empty data in that file.

Ext3 for example, does not include a native transparent compression support. However there is an unofficial patch e3compr [52].

ZFS by Sun Microsystems contains both deduplication system and compression using implicitly LZJB (very fast but compression-ratio-wise ineffective variant of LZ), or alternatively gzip. ZFS is a granular file system, and as such, compression can be selectively applied to different attachment points.

BTRFS – B-tree file system is a copy-on-write file system with native compression. The compression is not applied in a block scope but rather in an *extent* scope, which is a continuous run of data [78].

Some file systems are read-only. Cramfs, for example, compresses single pages using zlib [61]. It is used in embedded devices or for initialization or installation images.

Fusecompress is a FUSE-based file system that supports LZO, gzip, bzip2, and LZMA. Fusecompress also has an experimental port on Mac [98].

Lessfs is another example of FUSE-based file system [89]. Lessfs aims at data deduplication but also supports compression using LZO, QuickLZ, Bzip and data encryption.

Opendedup is another open source deduplication filesystem that comes with a volume manager and supports multiple storages. [9].

2.3 Deduplication and Compression Systems

Numerous deduplication and compression systems and techniques have been developed so far. Each of them occupies a certain place in the design space.

Several schemes are compared in [83, 112, 65].

2.3.1 XRAY

XRAY is a compression scheme optimized for high compression ratio and fast decompression. It uses several heuristics during both the modeling and compression process,

rendering the compression times a little slower than many other solution. The compression scheme is capable of operating over a large collection of files.

XRAY consists of a three-phased approach. In the first phase, a RAY algorithm is applied, that creates a dictionary based only on a small sample of the total input [16]. This so called *training phase* is repeated several times on different samples from the whole data set.

The second phase, called *testing phase*, takes another set of training data and performs the coding on this dataset. The phrase dictionary frequencies are adjusted accordingly to this pass, minimum-redundancy codes are generated and infrequently used dictionary items are discarded.

Third phase, called *coding phase*, performs the assignment of the minimum-redundancy codes in a stream. There are several different method of phrase matching based on a windows approach that identifies overlapping phrases.

On textual files with the total training data size ranging from 0.1% to 4%, a compression ratio of 3.0 to 2.3 bpc is achieved. XRAY shows that only a small part of the total data is necessary to build a well-performing compression model. [17].

2.3.2 SILO

SILO – SImilarity-LOcality based deduplication scheme is a heavy duty system designed for large datasets and distributed processing. It exploits both data similarity and locality. Data similarity is exposed by splitting files into segments similarly to other schemes, and by grouping set of small files into a larger segment. Locality is then exposed by concatenating subsequent segments in the data stream into bigger blocks to preserve the locality-layout on the disk. [110]

2.3.3 Cluster-Based Delta Compression

As mentioned in Section 2.5.6, finding an optimal model for delta-encoding for a collection of files or blocks is not a trivial task. By having a distance measure, for example an edit distance or a delta distance, between all pairs of file in the collection, one can find an optimal model by reducing the problem to maximum weight branching in a weighted directed graph.

Such problem however has a quadratic time complexity in the size of the collection. This scales poorly in any real world scenario. A scheme called *cluster-based delta compression* approximates the branching problem using approximate clustering. This is the first approach that eliminates the drawbacks of both simple-file and solid compression.

The authors elaborate on multi-dimensional parametrization of the scheme, including similarity measures, optimum branching, hash functions, sample sizes, edge pruning rules, edge weight estimates, etc. [79].

2.3.4 REBL

REBL – Redundancy Elimination at the Block Level is a fine-tuned system that combines all of duplicate block elimination, similar blocks delta-encoding and data compression.

The data stream is divided into chunks of variable length (similar to previous methods), Rabin fingerprints and SHA hash are generated for each chunk. The SHA hash is used to find a duplicate chunk in the database.

Rabin fingerprints are then used to determine the similarity of the chunks by computing *super-fingerprints* using the Rabin fingerprints. Chunks with a certain amount of matching super-fingerprints are then compressed using delta-encoding. There are several possible algorithm that can be used to determine the structure of the delta-encoding.

The remaining chunks that are not similar to any other chunk in the database are then independently compressed. [55]

2.3.5 Pcompress

Pcompress is a high-performance, parallel deduplication tool with a modular design. Pcompress features many improvements over current deduplication solutions, many compression algorithms like LZMA, Bzip2, PPMD, LZ4, LZFX, delta compression, etc., many hashing algorithms for chunk checksums, and much more.

Pcompress even tries different deduplication schemes with less precision than standard complete hash comparison that allow for much larger scale of deduplication application, such as segmented deduplication. Another interesting feature is a heuristic for appropriate compression algorithm detection used to compress chunks.

The scale of similar chunks compression is standard among other systems – it is based on minhash and delta compression.

Pcompress is one of the most accessible, feature-packed deduplication systems currently available.

2.3.6 Other Deduplication and Compression Systems

A framework called PRESIDIO – *Progressive Redundancy Elimination of Similar and Identical Data In Objects* introduces several methods for efficient storage that are replaceable via a class-based polymorphic approach. These methods are able to predict their effectiveness for the system to decide which one to use. Another technique called *harmonic super-fingerprinting* is used to produce successively more accurate measures of the chunks. [114].

Deep Store is a large-scale system architecture based on PRESIDIO. Deep Store is a distributed system composed of storage nodes and clusters [113].

Amar Mudrankit presented in his master's thesis an integration of a fixed-size block deduplication into a Linux kernel and ext3 file system. The system consists of a virtual block layer using additional context-implied hints for the deduplication itself. [71]

Another example of a deduplication system based on similarity matching schemes, similarity signatures and hash based identities is described in [5].

Summaries of relatively new deduplication or compression schemes and solutions: Extreme Binning [8], ChunkStash [29], Data Domain deduplication file system [117].

2.4 Optimization Criteria

Since there are many deduplication systems, a unified metrics had to be formalized in order to compare these systems to each other.

One of such metrics is formalized by [65]. It consists of the following criteria:

Fold-factor – the reduction in data footprint size due to deduplication, *Reconstruction time*, *Rate of deduplication* and *Resource consumption*.

2.5 Data Compression

In this chapter, we introduce the basics of *data compression* and *information theory*. Categorization and description of data compression techniques and algorithms is provided in the following range: lossless vs lossy compression, symmetric vs asymmetric, static vs semi-static vs adaptive, uniquely decodable codes, statistical coding, arithmetic coding, dictionary-based compression and context-based compression.

A few compression algorithms will be explained more in detail, however most will only be referenced to other sources, since this thesis is not focused solely on isolated data compression techniques. Note: compression as such is a parameter in this work and so it is studied in the context of this thesis only – for large scale redundancy removal, not for perfecting compression ratio on a small dataset (corpus).

Unless otherwise specified, the definitions and other information in this chapter are taken from the following data compression related sources: [64, 91].

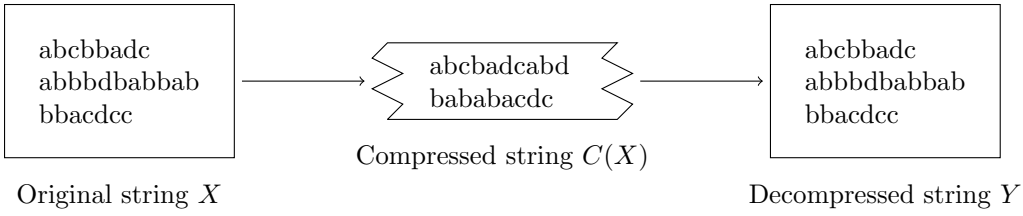


Figure 2.4: Data compression and decompression scheme. The original string X is compresses into X_C and then decompressed into Y. In case of a lossless compression, $X = Y$.

2.5.1 Preliminaries

For deeper understanding of compression and its effectiveness, we first have to define several basic terms.

An *alphabet* $\mathcal{A}$ is a finite non-empty set of symbols – characters, letters or digits. A *string* $S = \{s_1, s_2, \ldots, s_n\}$ over an alphabet $\mathcal{A}$ is a finite sequence. The length of $|S| = n$. *Source* is a generator of strings, *source unit* is then a string or a substring generated by the corresponding source. A *codeword* or a *code unit* is a binary string produced from a source unit using a code.

Definition 1 (Code)**.** Code K is a triple $K = (S, C, f)$ of a finite set of source units S, finite set of codewords C and an injective function $f : S \to C^+$ (injective means that it preserves distinctness).

Compression (translation) using the code K is the performed by $f(s) = c$, where $s \in S$ and $c \in C$. The inverse process is called *decompression* (inverse translation) $f^{-1}(c) = s$, where $s \in S$ and $c \in C$. If separate source unit produce separate codewords, we call this a *homomorphic translation*: $\forall s_1, s_2 \in S : f(s_1 s_2) = f(s_1)f(s_2)$.

A string $c \in C^+$ is *uniquely decodable* with respect to f if there is at most one source unit sequence $s \in S^+$ that translates to c - $f(s) = c$ and different codewords are decoded

to different source units

$$\forall c_1, c_2 \in C, c_1 \neq c_2 \implies f^{-1}(c_1) \neq f^{-1}(c_2)$$

A code is a *prefix code* if no codeword is a prefix of another codeword. A *block code* is a code where all its codewords have the same length. Note that both prefix and block codes are easily decodable.

Definition 2 (Compression ratio, factor)**.** *Compression ratio* is a measure of a code's effectiveness on a given data. Compression factor is its inverse.

$$\text{compression ratio} = \frac{\text{compressed length}}{\text{decompressed length}}$$

$$\text{compression factor} = \frac{\text{decompressed length}}{\text{compressed length}}$$

The most frequent units of the compression ratio are *bits per bit* (bpb) and *bits per character* (bpc). If $bpc > 1$, we call such compression a *negative compression*.

2.5.2 Categorization

If the source data can be completely reconstructed during the decompression phase, such compression is called *lossless*. Lossless compression is in some cases the only applicable compression, e.g. for text, documents, binary programs. If the overall information is reduced during the compression phase, such compression is called *lossy*. It is the most employed in scenarios, where human perception cannot effectively recognize the lost information, such as images, video or music.

Based on the computational complexity of the compression and decompression process, a compression can be divided into *symmetric compression*, where the complexity of both compression and decompression process is the same, e.g. Huffman coding, and *asymmetric compression*, where the complexities are different, e.g. LZ family compressions.

Another classification of coding is according to its *model*, further described in Section 2.5.4.

Based on the size of simultaneously processed data, compression can be divided into *block* and *stream* compression. In case of block compression, fixed subsequence of the source data is processed at a time. The redundancy is only removed within the single block. In stream compression, the entire source data is processed in a sequential fashion. The redundancy is removed from all over the source, however with uneven distribution.

Classification of compression methods can also be done by the main idea and principle: *elementary*, *statistical*, *dictionary*, *context* and *other* methods. For overview of lossless compression algorithms, see Section 2.5.5.

2.5.3 Information Theory

In 1948, Claude Shannon put together the idea of a quantitative measure of information [94]. The information based on a probability $P(A)$ of occurrence of an *event* A in an *experiment* S is called *self-information* of A:

$$I(A) = \log_2 \frac{1}{P(A)} = -\log_2 P(A)$$

Note that we will use log base of 2, since we always want to measure the information in *bits* (also known as *Shannon*). In case we used e (base of the natural logarithm) as the base, the information unit would be *nats*, and for base 10, the unit is *hartleys*.

For example, in the Shannon information theory, an uniformly random string contains more information than a well though-out thesis of the same length.

Let A and B be statistically independent events, so that $P(AB) = P(A)P(B)$, here $P(AB)$ denotes that both events occurred in the experiment. The self-information contained in two independent is:

$$I(AB) = \log_2 \frac{1}{P(A)P(B)} = \log_2 \frac{1}{P(A)} + \log_2 \frac{1}{P(B)} = I(A) + I(B)$$

Definition 3 (Entropy)**.** Let A_i be independent events from a set of all possible events in experiment $S = \cup A_i$. Then the entropy of S is

$$H(S) = -\sum P(A_i) \cdot I(A_i) = -\sum P(A_i) \log_2 P(A_i)$$

The entropy can be further extended to strings. For a source of strings S with alphabet $\mathbb{A}$ that generates a string $X = \{X_1, X_2, X_3, \dots X_n\}$, $\forall i : X_i \in \mathbb{A}$, the entropy of S for strings of length n

$$H_n(S) = \sum_{i_1=1}^{i_1=|\mathbb{A}|} \dots \sum_{i_n=1}^{i_n=|\mathbb{A}|} P(X_1 = i_1, \dots, X_n = i_n) log_2 P(X_1 = i_1, \dots X_n = i_n)$$

In case we are only interested in single characters – strings of length 1 ($P(X_i = i_1)$), we refer to such entropy as *zero-order entropy* $H(\cdot)$. In case we are interested in the k+1-st character given a directly preceding sequence of k characters ($P(X_{k+1} = i_{k+1} | X_1 = i_1, X_2 = i_2, \dots, X_{k-1} = i_{k-1}, X_k = i_k)$), we refer to the entropy as *k-th order entropy* $H_k(\cdot)$. Note that k-th order entropy and entropy of string of length k are two very different information measures.

For length of a codeword $|c(X_i)| = d_i$, the *length of encoded message* X is

$$L(X) = \sum_i d_i$$

and the average length is

$$L_{AVG}(X) = \sum_i P(X_i) \cdot d_i$$

Definition 4 (Redundancy)**.** The redundancy $R(S)$ of a source S is the difference between the codeword length d_i and the self-information $I(X_i)$ corresponding to its source word X_i

$$R(S) = L(S) - I(S) = \sum_i (d_i - I(X_i)) = \sum_i (d_i + \log_2 P(X_i))$$

The *average redundancy* is then

$$R_{AVG}(S) = L_{AVG}(S) - H(S) = \sum_i P(X_i)(d_i + \log_2 P(X_i))$$

Relative redundancy between two different sources S given (known) T can be described by *relative entropy*

$$H(S|T) = \sum_{i,j} P(X_i, Y_i) \log_2 \frac{P(Y_i)}{P(X_i, Y_i)}$$

2.5.3.1 Algorithmic Information Theory

Algorithmic information theory does not require the existence of an abstract source. It is another way of looking at information, that has unfortunately not been that useful in practice. The theory is based on *Kolmogorov complexity*, which is an uncomputable function expressing length of the shortest program that can generate a given string. In this work, Kolmogorov complexity is used in as a theoretical notion for compression-based data distances, please see Section 2.6.4 for details.

One of the practical implications of Kolmogorov complexity is a theory called MDL – *minimum description length*. The theory incorporates the existence of a *model*, further described in Section 2.5.4. In MDL, the goal is to minimize the length of descriptions of both the model and the data encoded using the model. [87]

2.5.4 Compression Models

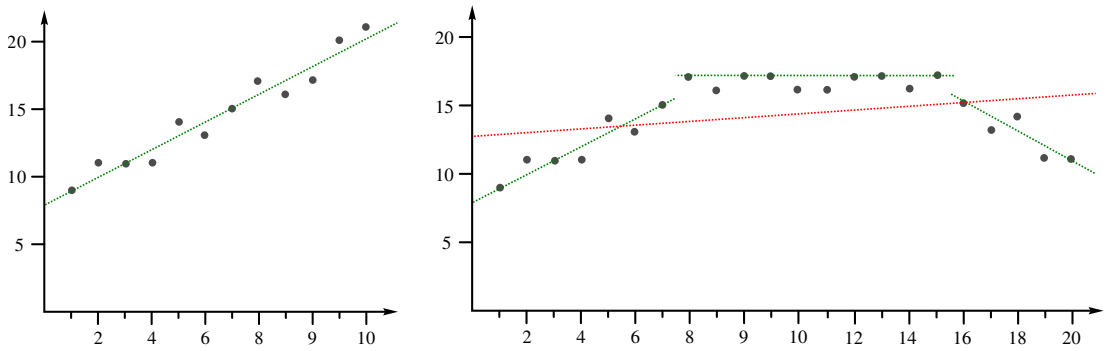
A mathematical model describing the source can be used to significantly reduce the total length of the resulting code.

Example 1. Consider the following sequence of numbers X from Figure 2.5a:

Simple binary encoding of these numbers would take 5 bits per number. However all the numbers lie approximately on a line. The linear function can be specified as $\hat{x} = n + 8$, where n is the index. This function is depicted in Figure 2.5a by a green dotted line. Let's use a model $d_X = X - \hat{X}$ to encode the numbers instead. See the following table for new values d_X to encode.

	1	2	3	4	5	6	7	8	9	10	11	12
X	9	11	11	11	14	13	15	17	16	17	20	21
d_X	0	1	0	-1	1	-1	0	1	-1	-1	1	1

The values d_X will fit in 2 bits each, which is significantly less than 5 bits if encoded as they were. We would also need some bits to encode the model, which in this case is a linear polynomial.



(a) Static linear model. [91]

(b) Adaptive linear model.

Figure 2.5: Demonstration of static and adaptive compression model on a sequence of data values. The green lines determine the currently closest linear function. The red line is a linear function minimizing the encoded values in case of a static model.

2.5.4.1 Probability and Markov Models

Assuming each character from alphabet A is independent from each other and has the same probability in a source S, we call such model the *ignorance model*. Such model can be useful in case we have absolutely no information about the source S.

Another step up in the complexity is to remove the equal probabilities. This is called a *probability model* $\mathbb{P} = \{P(a_1), P(a_2), \ldots, P(a_n)\}$, where $a_i \in A$. For a probability model, we can calculate entropy. The probability model is exploited by very efficient coding methods, such as the Huffman coding and arithmetic coding.

Removing the independence assumption results in the most interesting class of probability models – *Markov models*. Markov models are based on the knowledge of past k symbols. The theory is very similar to k-order entropy. Markov models are represented by a *discrete time Markov chains*, where the $\{X_1, \ldots, X_n\}$ are the n characters already read. If the probability estimations depend only on the last k characters, we refer to it is k-th order Markov chain. For simplicity, let $P(X_n)$ denote $P(X_n = i_n)$.

$$P(X_n|X_{n-1}, \ldots, X_{n-k}) = P(X_n|X_{n-1}, \ldots, X_{n-k}, \ldots, X_2, X_1)$$

This means that the probability estimates for the X_n character can be determined from only the last k-characters and the rest can be ignored. The $\{X_{n-1}, \ldots, X_{n-k}\}$ is the state of the process.

Markov models are suitable for standard text compression, where the following character is often determined by several preceding characters. Also it is the basis for PPM compression algorithms.

2.5.4.2 Static, Semi-adaptive and Adaptive Models

Models used as they are described and used in the previous section are called *static models*. Such models are predetermined once prior to any coding process and stay the same regardless of any input data. The initial model is estimated using some overall knowledge base, e.g. all the data in the world.

Semi-adaptive model determines the model prior to encoding. Once the encoding takes place the model cannot be changed. Most often, semi-adaptive compressors require two scans of the input data – the first determines the model, the second serves as an input to the encoding itself. The model can also be sent with the data, as mentioned in the minimum description length, see Section 2.5.3.1.

Adaptive models change the model during the encoding phase. Both compression and decompression start with the same model every time (just like static methods), but changes the model during the encoding phase. The model doesn't have to be transfered with the data, nor has the data be read twice.

Example 2. Extending the Example 1 to a semi-adaptive and an adaptive variant is quite straightforward. See Figure 2.5b.

In the semi-adaptive variant, scan the input data and determine a minimal linear error function. Such a minimizing error function can have several different forms. If we want to use fixed length code, then the maximal difference (error) should be the minimization criterion. In case we would rather use e.g. Huffman coding, then the minimizing criterion should be the entropy of the differences.

For the adaptive variant, let's assume we are able to scan the following k values prior to encoding them. In such case, we can use the same optimization criteria in the scope of the following k values only. Since the model is determined by looking ahead, we need to encode the change of the model too, otherwise the decoder would follow wrong reference function. If we are able to encode the reference function change effectively, the overall encoded length of the values will be significantly lower than with the semi-static variant.

2.5.5 Lossless Compression Algorithms

This section further describes lossless compression algorithms. Most of them are explained broadly, since the detailed analysis of these algorithms is out of the scope of this work, however principles, effectiveness and especially extraction of compression features are of high interest in this work, see Section 2.6.6.1.

The current state of lossless compression can be described by several major branches:

Efficient coding schemes include: statistical methods – *Huffman coding* [46] and *Arithmetic coding* [87], numeric codes (unary, rice, Golomb) and other codes.

The core duplicate string replacement algorithm *LZ77* [118] gave rise to numerous other algorithms such as LZSS (originally used in NTFS), *Deflate* (used in `zip`, `gzip`, `zlib`, PNG images, PDF documents, JARs), *LZMA* (originally used in `7-zip`), *LZX*, *ROLZ*, *Snappy* (used by Google, optimized LZ77 for x86-64).

Dictionary compression algorithms are based on *LZW* dynamic dictionary compression algorithm [107] (used in `compress` and GIF). Dictionary encoding is especially effective on text files.

Context sorting or block sorting algorithms are mostly based on the *Burrows-Wheeler Transform – BWT* [14]. *bzip2* is an example of a popular compressor.

Probability distribution (prediction) based models are for example bytewise, bitwise encoding (fixed orders), *Dynamic Markov Coding – DMC*, *Prediction by Partial Match – PPM*, *Context Tree Weighting – CTW*, *PAQ* and more.

A comprehensive summary of data compression techniques, algorithms, parametrization is available at [64] and a large-scale text compression benchmark is at [63].

The top algorithms for text compression compete in challenges such as the *Calgary challenge* [12] and the newer *Hutter challenge* [49]. These challenges are both won by PAQ-based algorithms.

2.5.6 Delta-Encoding

For two similar files – a reference file and a new file, *delta-encoding* generates a *delta* (a patch file) between these files. Delta encoding originally used so called string *edit operations*: copy, insert and delete. More sophisticated delta encoding algorithms are implemented by a stream matching algorithm that locates the offsets of matching chunks in the reference and the new file, emitting a sequence of edit operations that transform the reference file into the new file. A nice summary on delta compression and it's application to file synchronization is available in [97].

The most known such tools are the `diff` and `bdiff` tool for computing text file differencies. Other frequently used tools are `vcdiff`, `vdelta`, `xdelta` and `zdelta` that compute compressed representations of file differencies.

Delta encoding is only a tool to effectively encode one file based on the other. The most important aspect is how to select the reference file against which to produce the

delta in a deduplication or compression scheme of multiple objects. One of such schemes was recently described in [95]. The work also describes multi-layered delta encoding and combination of delta encoding with other compression techniques.

2.6 Distances and Similarities of Data

All data are created equal, but some are more equal than others. In this section, we describe many possible ways of expressing data equality and similarity in a more formal manner. Just as data itself, distances can be categorized based on several criteria such as input, application, online vs offline, approximation factor, availability of the original input, etc. For offline clustering problems, the most distinguishing criteria is the availability of features [23].

Feature-based similarities can be gathered from data of a specific type. Such methods usually require specific and detailed knowledge of the problem area. A feature is a single specific property of the data. For example in musical files, some of the numerical features are rhythm, pitch, harmony, etc. Text features are computed using standard information retrieval techniques like tokenization, case folding, stop-word removal, stemming and phrase detection. Features assign a unique *feature vector* to every file, which can act as an input to a clustering algorithm, see Section 2.7.1, or as a base of several linear space based distances, see Section 2.6.1.

Non-feature similarities do not assume any knowledge about the dataset. Such methods have to be general enough to capture all possible kinds of similarities. In practice, that is not the case. For example, changing an encoding of a book will result in a completely different binary image, rendering all but purely theoretical algorithms shorthanded. Currently the best non-feature similarity detection algorithms are based on compression.

Availability of the original data is a crucial criterion for any subsequent processing, especially in this work. If the original data is available for the distance function computation, we can simply compute the distance between the new item and the existing item. However in many incremental clustering, deduplication, compression and distributed system scenarios, this is not the case. To compute the distance, we must first fetch the data, which may be a time consuming task. In these scenarios, similarity hashes can be used, see Section 2.6.6.

Distance between data directly deployed in many areas, such as clustering documents by categories [23], reduction of web-crawling document space, detection of similar documents, plagiarism detection, spam detection, data extraction [66]. It is also useful in documents exchange. Several document exchange protocols were invented based on delta distances [26].

2.6.1 Distance Function, Metric and Norms

Definition 5 (Distance function)**.** A *distance function* is a function of a pair from a set of objects Ω to nonnegative real numbers $\mathbb{R}^+$ $D : \Omega \times \Omega \to \mathbb{R}^+$.

Definition 6 (Metric)**.** A distance function that satisfies the following conditions $\forall x, y, z \in \Omega$ is a *metric*, the object space Ω is then a *metric space*:

- $D(x, y) = 0 \iff x = y$

- $D(x, y) = D(y, x)$ (symmetry)
- $D(x, y) \leq D(x, z) + D(z.y)$ (triangle inequality)

The most familiar example of such metric is the Euclidean distance or the distance between geographical objects. Another example of a metric is a *discrete metric* – binary distance between categories such as book authors or clusters. $D(a, b) = 0$ if the two items a and b fall into the same category and $D(a, b) = 1$ otherwise. This is an example of a single feature described by a simple metric.

Definition 7 (p-Norm)**.** For a vector $X \in \mathbb{R}^n$, $p \in \mathbb{R}$, $p \geq 1$, *p-norm* is a function $\|X\|_p : \mathbb{R}^n \to \mathbb{R}^+$:

$$\|X\|_p = \left(\sum_{i=1}^{n} |x_i|^p \right)^{\frac{1}{p}}$$

Note: p-norm is usually defined in any vector space over a subfield of the complex numbers.

For $p = 1$, the p-norm is a *Manhattan norm* (Taxicab norm), for $p = 2$ it is the *Euclidean norm* and for $p \to \infty$ it is an *Chebysev norm* (uniform, infinity norm).

Definition 8 (Minkowski distance)**.** For two vectors $X, Y \in \mathbb{R}^n$, $X = (x_1, x_2, ..., x_n)$, $Y = (y_1, y_2, ..., y_n)$, the *Minkowski distance* is a function $M_p : \mathbb{R}^n \times \mathbb{R}^n \to \mathbb{R}^+$:

$$M_p = \|X - Y\|_p = \left(\sum_{i=1}^{n} |x_i - y_i|^p \right)^{\frac{1}{p}}$$

For $p \geq 1$ the Minkowski distance is a metric. For $p < 1$ it is not, since the triangle inequality does not hold. For vectors $(0, 0)$, $(0, 1)$, $(1, 1)$, and $p < 1$ the distance $M_p((0, 0), (0, 1)) = M_p((0.1), (1, 1) = 1$, but $M_p((0, 0), (1, 1)) = 2^{\frac{1}{p}} > 2$.

For $p = 1$ the Minkowski distance is a *Manhattan distance*, for $p = 2$ it is *Euclidean distance* and for $p \to \infty$ it is Chebysev distance.

For a distance functions or a metric, we want to exclude unrealistic degenerate distance measures like $D(x, y) = \frac{1}{2} \forall x, y : x \neq y$. One of these requirements is the *density condition*. Intuitively, it means that for every object x and $d \in \mathbb{R}^+$, there is at most a certain finite number of objects y at a distance d from x. The density conditions for x and y:

$$\sum_{x:x \neq y} 2^{-D(x,y)} \leq 1 \qquad \sum_{y:y \neq x} 2^{-D(x,y)} \leq 1 \tag{2.1}$$

Definition 9 (Admissible distance)**.** Let $\Sigma = 0, 1$, $\Omega = \Sigma^*$ be binary strings. A function $D : \Omega \times \Omega \to \mathbb{R}^+$ is an *admissible distance* if it satisfies the density condition, is symmetric and is computable.

The density condition applied to length of a code instead of the distance is known as the *Kraft's inequality*.

2.6.2 String Edit Distances

Definition 10 (String edit distance)**.** The *string edit distance* $d(X, Y)$ between two strings X and Y over an alphabet Σ is the minimal cost of edit operations required to transform X to Y.

Definition 11 (Edit operation)**.** The *edit operation* is a rule in the form $\delta(x, y) = c$, where $x, y \in \Sigma \cup \varepsilon$ are two characters from the alphabet of the two strings plus an empty string ε, and $c \in \mathbb{R}^+$ is the cost of this δ transformation.

The four basic edit operations are: [106]

- Insertion: $\delta(\varepsilon, a)$ – inserting the character a at any position
- Deletion: $\delta(a, \varepsilon)$ – deleting the character a from any position
- Substitution: $\delta(a, b)$, $a \neq b$ – replacing the character a with b
- Transposition: $\delta(ab, ba)$, $a \neq b$ – swapping two adjacent characters

If for each rule $\delta(x, y) = \delta(x, y)$, then we call the edit distance *symmetric*. If the following are also true: $\delta(x, y) \geq 0$, $\delta(x, x) = 0$, $\delta(x, y) \leq \delta(x, z) + \delta(z, y)$, then the edit distance is a metric.

All the following commonly used distances are metrics.

Definition 12 (Hamming distance)**.** *Hamming distance* allows only substitution operations. It is only defined for strings of the same length. Alternatively the distance is equal to ∞ for string of different lengths.

Definition 13 (Longest common subsequence distance)**.** *Longest common subsequence* is the longest pairing of characters between the two strings that preserves the order of these characters in both strings. `diff` algorithm is based on the longest common subsequence problem. *Longest common subsequence distance* represents the number of unpaired characters between the two string. The distance allows only insert and delete operations.

Definition 14 (Levenshtein distance)**.** *Levenshtein distance* allows insert, delete and substitute operations. It is the most commonly used edit distance in approximate string matching problems. The problem is often referred to as *string matching with k differences.*

Definition 15 (Damerau distance)**.** *Damerau distance* extends the Levenshtein distance with the transpose operation. The original idea behind this distance was to correct human misspelling errors.

Example 3. Assuming unary costs for all the edit operations, meaning all the edit costs are exactly 1. For two string X = `barborka` and Y = `brambora`:

Hamming distance between X and Y is 6, since all the characters but the first and last have been substituted.

Longest common subsequence distance is 4. Delete the first `a`, delete `k`, insert `a` after the second character, insert `m` after the third character.

`barborka -> brborka -> brbora -> brabora -> brambora`

Levenshtein distance is 3. The edit operations can be as follows: delete `k`, insert `r` after the first character, substitute the second `r` with `m`.

```
barborka -> barbora -> brarbora -> brambora
```

Damerau distance is also 3 in this case. Possible sequence of edit operations may be: delete `k`, transpose `ar` to `ra`, insert `m` after the third character.

```
barborka -> barbora -> brabora -> brambora
```

Standard dynamic programming algorithm for the computation of Levenshtein distance was published in [106]. It was later extended to work with Damerau distance.

An overview of string edit distances and the associate string-to-string correction problem is available in [76].

2.6.3 Delta Distances

Delta compression between two strings can be used to determine a distance function between those files. A subsequent compression of a sequence of strings can result in an effective way to encode such strings. A system for ideal delta compression among multiple files was described in [79] using both optimal branching and hashing techniques, see Section 2.3.3 for more details.

For an overview of delta compression methods, please see Section 2.5.6.

To use a delta compression method in the problem of string distance, we have to set up a formal notion.

Definition 16 (Delta distance)**.** For two strings X and Y, delta compression algorithm A_D, delta distance $d_A(x, y)$ represents the size of the difference produced by string Y being compressed using the delta compression algorithm A_D with a source X

$$d(X, Y) = |A_D(X, Y)|$$

2.6.4 Algorithmic Information Distances

In this section, we'll briefly describe a theoretical notion of the data distance. If we had an oracle able to compute the ideal compression of a file, we would also be able to compute an ideal compression of another file relative to the reference file. Such a theoretical tool is called the *Kolmogorov complexity* [59].

Definition 17 (Kolmogorov complexity)**.** The *Kolmogorov complexity* or *algorithmic entropy* $K(X)$ of an input string X is the length of the shortest binary program that outputs X with no input.

Note that Kolmogorov complexity is not computable – there does not exist such a compressor program. Kolmogorov complexity thus serves as the ultimate lower bound of what a real-world compressor can possibly achieve. [58]

It is easy to compute the upper bounds of $K(s)$ by compressing the string using a program and concatenating the program to the compressed string. The total length of the compressed string and the compression program.

Definition 18 (Conditional Kolmogorov complexity)**.** The *conditional Kolmogorov complexity* $K(X|Y)$ of an input string X is the length of the shortest binary program that outputs X with input Y. Kolmogorov complexity of a pair of strings is denoted by $K(X, Y)$, and of a concatenation of strings by $K(XY)$.

Lemma 1. Within an additive precision, the Kolmogorov relative complexity of a string X relative to string Y is equal to the Kolmogorov complexity of the pair of X and Y without the complexity of X alone. The complexity of a pair of strings is with a logarithmic precision equal to the Kolmogorov complexity of the concatenation of the strings. [58]

$$K(X|Y) \approx K(X,Y) - K(X) \qquad K(X,Y) \approx K(XY) \approx K(YX) \tag{2.2}$$

In practice, it is impossible to compute how far off the Kolmogorov complexity our estimate is.

Example 4. If we take a string from a uniform distribution, then very likely the Kolmogorov complexity of that string is close to the original length of this string. These may be encrypted data for example. Any standard compressor will provide a good approximation of Kolmogorov complexity of such string.

However if our string is the first 10^{20} digits of Π, then there is a program of size $< 10^5$ bits that generates the number Π. So the upper bound of Kolmogorov complexity is size of the program. However, no standard compressor is able to compress the string significantly.

Definition 19 (Information distance)**.** The length of the shortest program that computes string X with input of string Y and with input X computes the string Y is called *information distance*

$$ID(X,Y) = \max\{K(X|Y), K(Y|X)\}$$

To restrict the range of information distance, the *normalized information distance* is used.

Definition 20 (Normalized Information Distance)**.** *Normalized information distance* is the length of the shortest program that computes string X with input of string Y and vice versa relative to the sizes of Kolmogorov distances of the original input strings X and Y

$$NID(X,Y) = \frac{\max\{K(X|Y), K(Y|X)\}}{\max\{K(X), K(Y)\}}$$

Normalized information distance is universal, meaning that any admissible distance expressing similarity according to some feature that can be computed from the input data, is included in the NID.

A more in-depth analysis of Kolmogorov complexity is available in [59] and [102].

2.6.5 Compression-Dased Distances

In order to approximate the information distance, a real compressor has to be used. As mentioned in the previous section, this may fail to approximate the real Kolmogorov distance, however for the purpose of this thesis, where real compression is the goal, it does not matter.

LZ distance and *LZ metric distance* are both LZ compression inspired distances [26].

Definition 21 (LZ distance)**.** LZ distance $LZD(X,Y)$ (produce X given Y) between two string X and Y is equal to a number of characters or substrings of Y or the partially built string that are needed to produce X. Note that LZD is not a metric, since it is not symmetric.

Example 5. Let X = `aabcdefgabc` and Y = `aaaaaaaaaaaaaaa`. String X given Y can be constructed in the following way:

```
(aa) -> aa(b) -> aab(c) -> aabc(d) -> aabcd(e)
     -> aabcde(f) -> aabcdef(g) -> aabcdefg(abc)
```

but Y given X can be reconstructed in the following way:

```
(aa) -> aa(aa) -> aaaa(aaaa) -> aaaaaaaa(aaaaaaaa)
```

Thus, $LZD(X,Y) = 8$ and $LZD(X,Y) = 4$.

Definition 22 (LZ metric distance)**.** LZ metric distance $LZMD(X,Y)$ extends standard Levensthein distance with more operations to transform string X into Y: insertion, deletion and substrituion of a single character, copy of a substring, deletion of a repeated substring.

LZMD is a metric, when all the operations are reversible and have a unit cost.

2.6.5.1 Normalized Compression Distance and Variations

The information distance approximated by the real compressor C is called *compression distance* $ID(X,Y) \approx CD(X,Y)$ [23]

$$\begin{aligned} &max\{K(X|Y), K(Y|X)\} \approx max\{K(XY) - K(X), K(YX) - K(Y)\} \approx \\ &\approx min\{C(XY), C(YX)\} - min\{C(X), C(Y)\} = CD(X,Y) \end{aligned} \tag{2.3}$$

Most real compressors are symmetric or almost symmetric. Block-coding based compressors are very close to symmetry by definition and stream compressors usually only have a little deviation from symmetry. The deviations from the rules of metric can be cause by several factors. Because of string alignment issues and possible model flushing, adaptive compression algorithms tend not to be perfectly symmetric. In the LZ family, when there is no dictionary flushing and the string X uses up the full capacity, then the string Y is compressed with a wrong dictionary, resulting in $C(XY) \geq C(X) + C(Y)$.

Assuming symmetry, the compression distance can be simplified to $C(X,Y) = C(XY) - \min\{C(X), C(Y)\}$.

Definition 23 (Normalized compression distance)**.** An approximation of the normalized information distance by a compressor C is a *normalized compression distance*

$$NCD_C(X,Y) = \frac{C(XY) - \min\{C(X), C(Y)\}}{\max\{C(X), C(Y)\}}$$

The NCD has proven as a very accurate approximation of the Kolmogorov distance. This approximation is gradually improving with better compression algorithms. NCD is a metric with a logarithmic tolerance. [23]

Several other less known compression distnaces were described in [92].

The other distance functions *CLM: Chen-Li metric*, *CDM: Compression-based Dissimilarity Measure*, *CosC: Compression-based Cosine* are all variations of the NCD. The major difference of these other methods is that they only have a different normalization term. The experimental performance of these compression similarity methods is almost the same.

Several feature extractions from the compression algorithms LZ77, LZW and PPM are presented. The compression similarities are then computed from these extracted feature vectors. However these feature vectors are of very high dimensions.

2.6.5.2 Compression Dictionary Distances

A compression dictionary can be seen as a specific case of a feature space generated by a dictionary-based compression algorithm. The main idea is to omit the joint compression step that all NCD-based techniques possess. This provides a huge advantage, since the generation of the dictionary may be online a once the dictionary is generated and the original file no longer needs to be available.

For a dictionary $D(X)$ extracted by a dictionary compressor such as LZW on a string X, the $D(X)$ represents the feature vector of X.

Definition 24 (Fast compression distance)**.** For two strings X and Y and their dictionaries extracted by a dictionary compressor C, the *fast compression distance*

$$FCD_C(X,Y) = \frac{|D(X)| - |D(X) \cap D(Y)|}{|D(X)|}$$

By sorting the dictionaries alphabetically, the operation of set intersection is much more time-effective than the joint compression step by NCD methods. [20]

Another compression method *McDCSM – Model Conditioned Data Compression based Similarity Measure* uses dictionaries extracted from LZW to determine relative compress ratio based on these extracted dictionaries [19].

Definition 25 (McDCSM)**.** For two strings X and Y, the term $|(X|D(Y)|$ represents the length of a string X coded with the dictionary extracted from the string Y. The *McDCSM* is defined as

$$McDCSM(X,Y) = \frac{|(Y|(D(X) \cup D(Y))|}{|(Y|D(Y)|}$$

2.6.6 Features Extraction and Similarity Hashes

The compression dictionary comparison is an example of a feature extracted from string data. In the case of compression dictionaries, the feature is represented by a set. Such set can be transformed into distance measures, such as FCD and McDCSM described in Section 2.6.5.2. Many other set-based measures can be developed using Jaccard index [86] or Rand index and fuzzy variation of the Rand and Jaccard index [13].

2.6.6.1 Compression Features

Features in other forms may be extracted from general data using either dedicated preprocessing or as a side effect of another process, e.g. compression. Detailed theoretical description and examples of such feature vectors extracted from compressors are described in [92]. There is a high dimensional vector space $\mathbb{V}$ describing the compression process of a compression algorithm C. For every input string x, there is a unique non-negative vector $\vec{x} \in \mathbb{V}$ associated with this input string x using the compression algorithm C. The length $C(x)$ of the compressed string must correspond to a vector norm $\|\vec{x}\|$.

From *LZ77* [118], the feature vector is extracted using the substring in the context window – offset and length. Start with a zero vector $\vec{x}$ of length equal to the length of the context window p times the maximal substring length m (m can have an upper bound). Then for every output symbol of length c (pair of offset and length) increment the corresponding position in the vector by c. Note that $C(x) = \|\vec{x}\|_1$, because the vector $\vec{x}$ consists of the lengths of the output symbols only and $\|\vec{x}\|_1$ is then the total length of all the output symbols.

In case of *LZW* [107], the feature vector $\vec{x}$ corresponds to all possible strings in the dictionary. The length of a substring in the dictionary is only limited by the maximal number of entries in the dictionary $O(2^c)$, where c is a fixed length of the output code. The dimension of the feature vector for LZW compression is then $O(2^{2^c})$, which is much larger than in the previous case for LZ77. In this case, zero initialized vector with increments of c similar to LZ77 is used, and $C(x) = \|\vec{x}\|_1$ as well.

The feature vector of *PPM* [24] can be extracted in the following way: For n-order compressor, let the first n dimensions of the feature space V be determined by the n-symbol context and the last dimension is determined by the currently processed symbol. As well as for LZ77 and LZW extraction, zero initialized vector with increments of c is used, and $C(x) = \|\vec{x}\|_1$.

The fact that compression of any string can be expressed using a vector in a vector space allows us to analyze compression-based similarity measures in a vector space. Assuming adaptive, reliable compressors, the compression lengths satisfy the triangle inequality $C(xy) \leq C(x) + C(y)$ and so do norms of the corresponding vectors $\|\vec{x} + \vec{y}\| \leq \|\vec{x}\| + \|\vec{y}\|$. The compressor needs to satisfy this property, otherwise the feature vectors extracted will not satisfy it as well. The deviations from the metric are described in more detail in Section 2.6.5.1. If the compressor satisfies the metric properties, then also:

$$max\{\|\vec{x}\|, \|\vec{y}\|\} \leq \|\vec{x} + \vec{y}\| \leq \|\vec{x}\| + \|\vec{y}\|$$

To measure the distance between $\vec{x}$ and $\vec{y}$, we cannot use the Minkowski distance (see Section 2.6.1), because there is no subtraction operation defined on these non-negative vectors. However, this can be solved using a vector similarity measure

$$\|\vec{x}\| + \|\vec{y}\| - \|\vec{x} + \vec{y}\|$$

When x and y are very similar, this term approaches $max\{\|\vec{x}\|, \|\vec{y}\|\}$ and when they are very dissimilar, the term approaches 0. This implies the normalization factor for NCD. In terms of high-dimensional vectors, the NCD can be expressed as

$$NCD(x, y) = 1 - \frac{\|\vec{x}\| + \|\vec{y}\| - \|\vec{x} + \vec{y}\|}{max\{\|\vec{x}\|, \|\vec{y}\|\}}$$

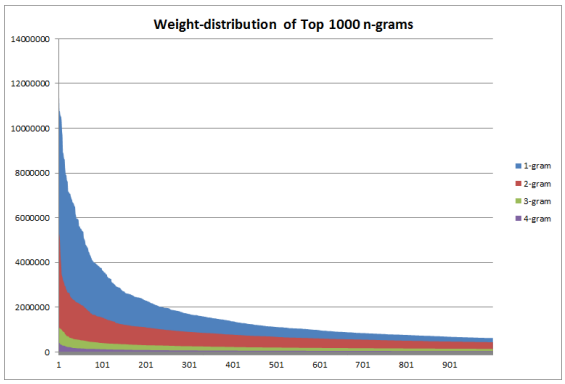


Figure 2.6: Weight for the 1000 most popular 1-4 word phrases in 9 million tweets from twitter.com. The weight partially suppresses multiple posts by single user. Without the suppression, the distribution would be even steeper. Figure taken from [70].

2.6.7 N-Grams

N-grams is the simplest and the most frequently used form of feature extracted from strings. N-grams have two major granularities: *character* and *word*. Character n-grams are based of fixed-length characters, such as 1 byte, while word n-grams are based on substrings separated with a delimiter.

N-grams are frequently used for classification of textual files. Examples of such classification are a language recognition, author recognition, etc.

Definition 26 (n-gram)**.** A character-based *n-gram* is a string X is a substring of X of length n. In other words, it is a n-character consecutive sequence that occurs somewhere in X. For $n = 1$, the set of 1-grams is the set of all the unique characters in X. The n-grams are called *bi-grams* for $n = 2$, *tri-grams* for $n = 3$ and *quad-grams* for $n = 4$.

Definition 27 (Order, Weight and Rank of n-gram)**.** *Order of n-gram* N is n. *Weight of n-grams* N is the number of occurrences of N in X. *Rank of n-gram* is its position in an ordered set (by weight) of all all n-grams of X.

From now on, we only assume character based n-grams.

Example 6. For a string `SWISS_MISS`, the bi-grams, tri-grams and quad-grams are the following sets:

bi-grams	`IS`(2x), `SS`(2x), `SW`, `WI`, `S_`, `_M`, `MI`
tri-grams	`ISS`(2x), `SWI`, `WIS`, `SS_`, `S_M`, `_MI`, `MIS`
quad-grams	`SWIS`, `WISS`, `ISS_`, `SS_M`, `S_MI`, `_MIS`, `MISS`

In the case of single order n-grams, the problem of the beginning and end of a string is solved by adding a special character, so called *padding*. In that case, the number of n-grams in a string X is always $|X| + 1$. In mixed order n-grams, this is usually not needed.

One of the first n-gram distances has been introduced in [18] for word-based n-grams. It is based on the *Zipf's law*. Zipf's law directly implies that a small amount of most frequent words account for the highest overall coverage of the text, see Figure 2.6. By extracting only the top T n-grams from both strings, we get an *ordered list* on n-grams. The weight are then discarded, only the order is preserved, yielding a *ranked list*. Every n-gram in either list is then assigned a mismatch value based on the difference of its position in both the ranked lists. In the case the n-gram is not present in one of the lists, it is given a mismatch value of T. The sum of these mismatch values is then the total

distance between the two strings. Many other methods to compare ordered or ranked lists can be used to determine the overall distance.

N-grams are not always used as multisets (or weighted sets), but can be used as a simple set. In that case we refer to the problem as *binary bag of n-grams* or *binary bag of words* [92]. Such sets can then be used with the Jaccard or Rand index mentioned in Section 2.6.6. Another measures of similarity called *shingle intersection* and *shingle containment* were introduced in [79].

N-grams are frequently used for prediction of user input, as in the case for people with disabilities. A nice large-scale analysis are available at the *Google Ngram Viewer* [36] or at the *Twitter talk analysis project* [70].

2.6.8 Similarity Hashing

Traditional *hash functions* such as SHA-1 or MD5 map a long string, resp. vector into a short one. If you change a single byte in the string, the resulting hash will differ significantly, meaning its Hamming distance to the original hash will be high. The *similarity hash function* acts in the opposite way. Similar strings have similar hashed – the Hamming distance is small.

The core example of a similarity hash function is *simhash* [22]. The simhash has also been applied to the *hamming distance problem* [66], which is similar to nearest neighbor search described in Section 2.7.6. Main idea behind simhash is to add together hashes of length f of all the single features from the string into a similarity hash that is then transformed into a binary hash of the same length f.

Example 7. To get a simhash of size 64 bits from a string $X =$ `SWISS_MISS`: Have a zero-initialized vector of size 64. We will compute hashed using a standard hashing algorithm for all tri-grams of the string, which are `ISS(2x), SWI, WIS, SS_, S_M, _MI, MIS`.

```
hash("ISS") = 1110100011011010000010111101110110000010001101110011110100011000
hash("SWI") = 1000010100011111100100110001011100100000100001010000101011100000
hash("WIS") = 1100001110100001110010010010111000110100001001101100011011101100
hash("SS ") = 1000010100011111100100110001011100100000110000100001001111100011
hash("S M") = 1000010100011111100100110001011100100011101011011100001011111011
hash(" MI") = 0000101111101110010011110100011101110011111011011100101010001101
hash("MIS") = 0101100010100011101111100000101101111001111000110101101110111010
```

The hashes are then added to the simhash vector. Zeroes on i-th positions decrement the corresponding value on i-th position of the simhash, ones increment the value. The simhash itself is then extracted from the simhash value. For positions, where the value was positive, there is 1, otherwise there is 0.

```
simhash(X)  = 1000000100011111100010110001011100100000101001110000000110101000
```

The simhash approach of [22] has been extensively compared to [10] in [44], where the two were also combined together in a contenxt of finding near-duplicate web pages.

2.6.9 Min-Wise Independent Hashing

Min-wise independed hashing – *minhash* first introduced in [11] deals with dimension reduction using multiple independent hashing functions. Let every object be represented by a binary vector v, corresponding to a presence of a n-gram in the objects. Note that

v only represents a set, not a multiset or n-grams related to the object. Let there be n simple hash functions (linear will do well) that are used to randomly permute the vector v. The minhash h of length n is then constructed. The i-th element of the minhash h is then the minimal index of the i-th hash (in the permuted vector) with value 1 (meaning there is the corresponding n-gram present). Note that the minhash's vector size is n, but the binary size depends on the number of indices in the vector v (with possible cut-offs).

Minhash estimated the Jaccard similarity. A single hash function probes the Jaccard similarity on a single pair from the sets. If the minhash exhausted all the permutations, it would correspond exactly to Jaccard similarity. However the fixed set of hash functions ensures that there is the same rule for choosing pairs to sample in all the set comparisons.

For nice examples, see [85].

Other approaches to similarity hashing are: *Sdhash*, that (as opposed to previously described techniques) select statistically improbable features, that are unique for each object [88]. *Sketching* is another method based on hashing object n-grams multiple times to make a sketch of the object. Sketching is described in [67, Chapter 19].

2.6.10 Locality Sensitive Hashing

Many previously mentioned methods required computation of all the pairwise distances. Even if the distance computation is performed very quickly (e.g. Hamming distance between hashes), the quadratic complexity still scales the computation time too high. If our goal is to compute the pairwise distance between all objects, there is no way around the problem.

However, quite often, we are looking for either a set of nearest objects to a reference object – this is called the *nearest neighbors problem* and its approximate variant. Both of these have been well described in [35].

Another problem is so called *near neighbor search* solved by techniques called *Locality Sensitive Hashing* originally introduced in [47]. The near-neighbor search tries to find all objects within a certain distance from a reference object. LSH is a technique using hashing for dimension reduction. It is a probabilistic approach of mapping input objects into a much smaller number of *buckets*. The hashing is performed multiple times with different similarity hashing functions or the hashes are divided into sub-hashes – these elementary units are called*bands*. Bands are then used for separate bucket assignments. Similar items are mapped into the same buckets with a high probability. When looking for similar objects, only objects in the same buckets as the reference object are considered. These objects are called *candidate objects*. Based on the total number of *bands*, the change of a false negative (missed similar object) decreases significantly. Deeper analysis with examples, application of minhash in LSH is described in [85].

Locality Sensitive Hashing has been successfully applied in many areas, such as the nearest neighbors search, near-duplicate detection, image similarity identification. The most interesting application for this thesis is its application to agglomerative hierarchical clustering. [51]

2.7 Clustering Analysis State of the Art

Measurements in a wide range of fields are generated continuously and in very high data rates. Examples of such are sensor networks, web logs, network data traffic. *Data*

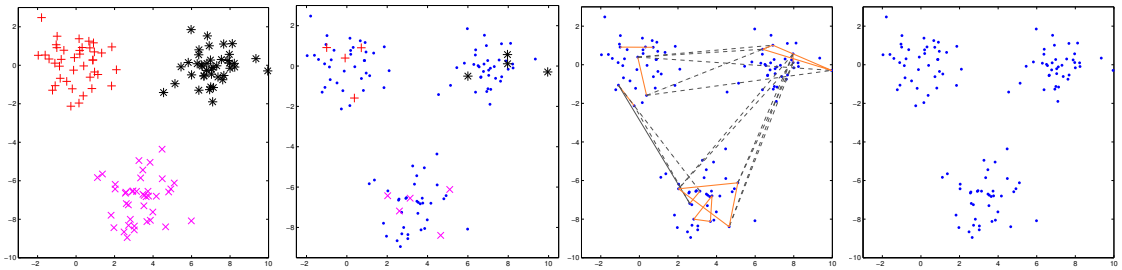
analysis is a process to understand and summarize such data. Based on a goal, data analysis can be divided into two groups: *exploratory*, where the investigator does not have a pre-specified model, but wants to understand the general characteristics of the data; second groups is *confirmatory*, where the investigator wants to confirm a hypothesis given the available data.

Cluster analysis is a method to discover the natural grouping of a set of objects, patterns, or points. Unlike *classification* that has class labels associated with the data objects and is a *supervised learning*, clustering does not have any labels and is an *unsupervised learning*. A mix between supervised and unsupervised is so called *semi-supervised learning*, where the labels are only present with a subset of the data, however all of the data is used in the learning process. In semi-supervised learning, *constraints* can be used to create links between the data: *must-link* and *cannot-link*.

A recent situation in machine learning has shifted the concept of cluster analysis from *offline* (batch) to *online* (incremental) processing due to large amount of data produced every day. The concept of is further described in Section 2.7.5.

This section is a summary on modern cluster analysis problems, algorithms and concepts. Detailed description of cluster analysis algorithms required in this work is described in Section 2.7.2.

Excellent sources on data mining techniques are the following books: Data Mining: Concepts and Techniques [40], Principles of Data Mining [41] and The Elements of Statistical Learning [43]. Another resourceful summarization article aimed mostly at partitional clustering algorithms is [48] and [111].



(a) Supervised (b) Partially labelled (c) Partially constrained (d) Unsupervised

Figure 2.7: Learning problems: (a) Labeled – classes are denoted by different symbols. (b) Partially labeled – unlabeled data are denoted by dots. (c) Partially constrained – lines represent must/cannot-links. (d) Unlabeled. Figure taken from [56].

2.7.1 Input Data

A typical offline clustering algorithm may have two different input types:

- *Pattern matrix* – $n \cdot d$ matrix for n objects, of which each is described in d-dimensional space by d values. Such vectors of size d are called a *feature vector* and the space is called *feature space*.

- *Similarity matrix* – $n \cdot n$ matrix for n objects, where for each of them the similarity to other objects in the set is described by a value in the matrix. The closer the objects, the higher the corresponding value in the matrix. A *dissimilarity matrix*

or a *distance matrix* on the other hand describes how far apart the objects are. On a normalized scale $D = 1 - S$, where D is a distance and S is a similarity.

Transition between these types of data representations is possible. To get the distances from a pattern matrix, is is possible to simply apply an distance function. The most known distance function on metric space is a *norm*, namely the second norm, the *euclidean distance*.

Getting a pattern matrix from a distance matrix is a minimization problem of an error (or a *strain*) called MDS – *multidimensional scaling*. More about MDS in Section 2.7.7.

There are other input types for clustering algorithms that are referred to as *heterogeneous data*:

- *Rank data* – cluster analysis is based on data from user questionnaires, ranking and voting. Such empirically measured data are often incomplete and with a varying number of available data fields. [15]

- *Data stream* – is a continuous flow of dynamic data, that has to be processed on the fly in a single pass and cannot be stored on a disk to perform a batch processing later. There is an usually high volume and potentially unbounded size of the data, and a sequential access to the data only. Data stream requires the clustering algorithm to adapt to changes in the data distribution, to be able to merge or split clusters, and discard old clusters. Further details are in Section 2.7.5.

- *Graph data* – data represented in a graph is natural for many application. Graph in this case refers to a single object, e.g. a chemical compound. It does not refer to the whole data input, in which case a weighted graph would be easily transfered to a similarity problem, and an unweighted graph would result in a link-constrained setup. Graph clustering is mostly based on extraction of features of the graph, such as frequent subgraphs, shortest paths, cycles etc. [103]

- *Relational data* – the problem is given as a large graph with links of diverse types. The goal is to partition this large graph based on the links structure and node attributes. The key issue is to define an appropriate clustering criterion. [100]

2.7.2 Clustering Algorithms Disambiguation

Clustering algorithms are usually divided at the principle level into two groups: *partitional* and *hierarchical*. Partitional algorithms find all the clusters simultaneously as a partition of the input data. The oldest and most known example of a partitional clustering algorithm is k-Means [62] and k-medoid. Partitional algorithms can easily use vector data (pattern matrix). They can be seen as optimization problems over d-dimensional space.

Hierarchical algorithms on the other hand create a hierarchical structure. In *agglomerative* more, hierarchical clustering starts with all the data objects as their own clusters and merges consecutively the most similar clusters until there is just one cluster left – or any other pre-determined amount of clusters. The opposite is *divisive* mode, where the algorithms start with a single cluster composed of all the objects and recursively divides the cluster.

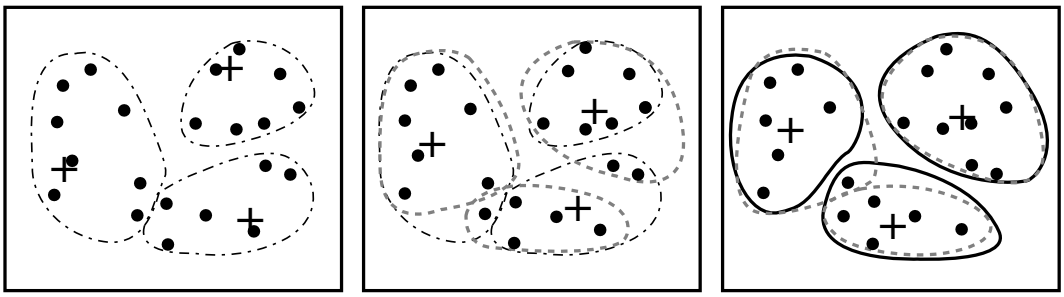


Figure 2.8: Three iterations of clustering by k-Means. The cluster centers are marked by +. Figure taken from [40].

A measure of dissimilarity between clusters is required in agglomerative hierarchical algorithms. The most known types are: *single linkage* – takes the minimal distance pair between the two clusters, the best known implementation is called SLINK [96]; *complete linkage* – takes the greatest distance pair; and *average linkage* – takes the average of all the pairs. An visualization of the three in dendrograms is on Figure 2.9.

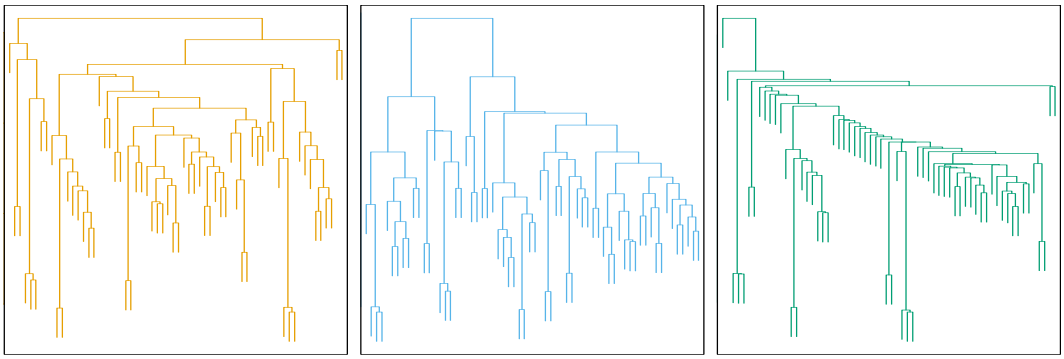


Figure 2.9: Dendrograms from agglomerative hierarchical clustering. Figure taken from [43].

The original k-Means algorithms has been extended numerous times. In k-Means, all the objects are assigned to exactly one cluster. In *fuzzy c-Means*, objects are assigned to multiple clusters with a variable weight of the assignment. Another extension is X-Means, that dynamically finds k – the right amount of clusters by optimizing a criterion such as Akaike Information Criterion (AIC) or Bayesian Information Criterion (BIC) [82]. The most known alteration is the k-medoid, where the median value of the objects from the cluster is used to represent the cluster [81].

Most clustering algorithms operate on a feature space (with an input of a pattern matrix). A specific class of probabilistic clustering algorithms assumes the data are generated from a mixture distribution [69].

For data represented as a weighted graph, there is a *spectral clustering*. The goals is to divide the graph in two subgraphs so that the graph is split over the minimal sum of the edge weights. The original solution to this problem used minimum cut algorithms. Current spectral clustering algorithms also consider cluster size and many other criteria. In-depth description and summary of spectral clustering algorithms is available in [105].

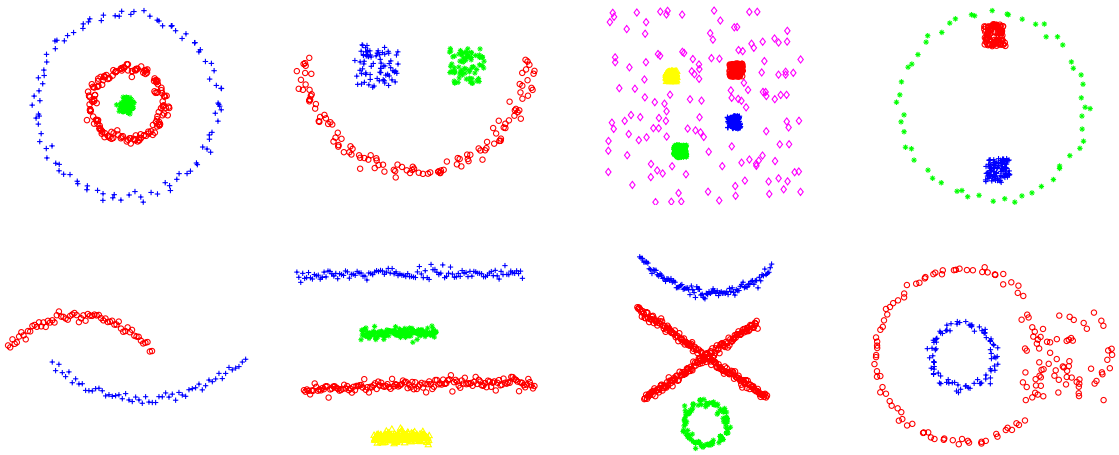


Figure 2.10: Example results of spectral clustering algorithm. Figure taken from [115].

Another approach for clustering similarity data is *message passing* [33]. All the objects are equally considered to be the clusters. The magnitude of the messages sent to other nodes varies based on a search of a minimum of an error function. The messages are sent in iterations, looking for an exemplar neighbor, based on which a clustering can be determined.

Correlation clustering was first proposed to solve a graph partitioning problem on a weighted full graph with binary weights. [6] The approximate algorithms were then extended to work on real-valued weights.

2.7.3 Clustering High-Dimension Data

Most clustering algorithms are not capable of effectively processing data of high dimensions (10 or more, thousands, or even tens of thousands). Usually only a small subset of the dimensions contains relevant data to certain clusters, but the remaining irrelevant data may cause noise.

Two major approaches to altering the feature space are *feature transformation* and *feature selection*.

Feature transformation techniques transform the data into less dimensions, while preserving the original relative distances between objects. Example of such techniques are the *principal component analysis* and *singular value decomposition*.

Feature subset selection determines a subset of features that are the most relevant for the clustering and discards the rest.

Summary of high-dimension data clustering is available in [54].

2.7.4 Clustering Large Data Sets

Handling large data sets requires specialized kinds of clustering algorithms. These can be classified into the following categories according to [48]:

- *Approximate nearest neighbor search* (ANN) – Many clustering algorithms need to decide the cluster membership of each point based on nearest neighbor search. An approximate search can yield a sublinear complexity for this task at a minimal loss of accuracy. More on ANN in Section 2.7.6. [72, 2]

- *Summarization* – The idea is to summarize a large data set into a smaller subset and then apply the clustering algorithm on this smaller subset. Example of such algorithm include the BIRCH, see Section 2.7.5 for details, and coreset for k-means and k-median. [42]

- *Distributed computing* – Each step of the data clustering is divided into a number of independent procedures. These are then computed in parallel. A survey on distributed data mining algorithms, including clustering was given in [80]. An example solution is described in [30].

- *Sampling-based* – Partitioning the dataset into clusters can be based on a subset of the whole data set. An example of such algorithm is CURE [39].

- *Incremental clustering* – See Section 2.7.5.

2.7.5 Incremental Clustering

Incremental clustering, sometimes called *data stream clustering*, is a specific case of clustering problem, where the data to be clustered arrive continuously. Such streaming model has derived from scenarios, where the entire data were too large to fit in memory. In data stream clustering, the data is expected to be finite. The only difference to batch clustering is in the stream availability of data. Conversely, in *online clustering*, the data stream is considered to be infinite. So in online clustering, there must be a way to periodically get the current clustering.

One of the most famous examples is the COBWEB [32]. It keeps a classification tree as its hierarchical clustering model. Then the algorithms places new points in a top-down fashion using a category utility function.

Another incremental hierarchical clustering algorithm that works in a bottom-up fashion is described in [108].

SLINK [96] is the most time-wise effective implementation of single linkage hierarchical clustering. It works incrementally, building several linear indexes.

BIRCH [116] also uses a hierarchical clustering, but the hierarchy is built on so called clustering features. A clustering feature statistically summarizes the underlying data.

DBSCAN [31] searches for its nearest neighbors when placing a new point. If there are sufficiently enough points under a minimal distance of the new point, such point is then added into the respective cluster of the nearest nodes. A generalization of DBSCAN called OPTICS [3] work with a varying density of clusters.

CURE [39] uses yet another approach to clustering. It lies between BIRCH and SLINK, as it uses hierarchical clustering, but instead of representing the cluster with once center as BIRCH, or considering all points as SLINK, it chooses only several representatives of the cluster that are then moved closed to the center.

Algorithms (some of those described only in referred overviews) such as DBSCAN, OPTICS and DENCLUE, STING, CLIQUE, Wave-Cluster and OPTIGRID do not optimize the k-means objective. An overview article of incremental data stream algorithms [34, 38, 21].

2.7.6 Nearest Neighbors and Approximate Nearest Neighbors

The NN – *nearest neighbors*, sometimes referred to as *nearest neighbors search*, is a problem of finding the k nearest neighbors. NN search for K nearest neighbors is often abbreviated as KNN. A variation of NN is ANN – *approximate nearest neighbors*, or ϵ-NN, which in an approximation algorithm.

An effective algorithm using randomized KD-trees is introduced in [72]. Another solutions use for example Approximate Principal Direction Trees [68] or PCA trees [104].

A comparison of ANN algorithms is available for example in [60], however it is a little older than the two previously mentioned approaches.

2.7.7 Ordination Methods

Ordination refers to a transformation of objects from high-dimensional space to a lower-dimensional space (order), so that objects that are closer to each other in the original space are also closer in the target space. For example in a full n-gram space, simhashing 2.6.8 is used to transform the exponentially dimensional space of n-grams to an artificial space of fixed number of dimensions.

Some of the ordination methods are discussed in this work. While it is appropriate to list the representatives of ordination methods, they are not directly applied to our problem.

Multidimensional scaling (MDS) is the most common ordination method. In its most typical form, it transforms a distance matrix into a low dimensional space, usually for visualization. There are many more variations of MDS [27] such as an incremental version of MDS [1, 109].

Principal Component Analysis (PCA) transforms the vector space into a new orthogonal space, where the data is linearly uncorrelated in the new dimensions (principal components) [50]. A brother of PCA that applies to categorical instead of continuous data is called Correspondence Analysis [37].

Chapter 3

ICBCS – Incremental Clustering-Based Compression System

The ICBCS – Clustering-Oriented COmpression System is completely described in this chapter, together with a brief description of its implementation and several optimizations used to speed up the whole system.

Combining properties of both a lossless compressor and an archiver, the ICBCS creates an ultimate deduplication, compression and archival system. It was designed with high extensibility in mind, and various parts of the system can be replaced or altered.

ICBCS has evolved during the development process significantly. Various clustering techniques were tested and replaced, as some of those failed to live up to the performance expectations. The most notable difference was between distance based clusterings and vector space based clusterings described previously in the survey, Section 2.7.1.

This chapter attempts to describe the system as thoroughly as possible, however not all implementation details are provided. The description is sometimes rather theoretical and omits certain amount of detail.

Complete description of the ICBCS is written in Section 3.2. Deduplication layer is described in Section 3.3, the two major approaches to clustering: SLINK and incremental clustering are described in Section 3.5, resp. 3.6. Compression and grouping into compression groups is described in Section 3.7. Next Sections 3.8 and 3.9 describe the compressor, resp. archiver capabilities of the system. Following are several implementation notes in Section 3.10 and optimization notes in 3.11.

3.1 Objecting conventional approaches

3.1.1 Objecting Solid and Single-file compression

See Section 2.1.2 for description of solid compression and Section 2.1.1 for description of single file compression and their respective applications.

For the design of ICBCS, it is important to realize precisely where the drawbacks of solid and single-file (non-solid) compression lie. Every currently wide-spread compression

algorithm only works with a limited scope, beyond which no or little redundancy is removed.

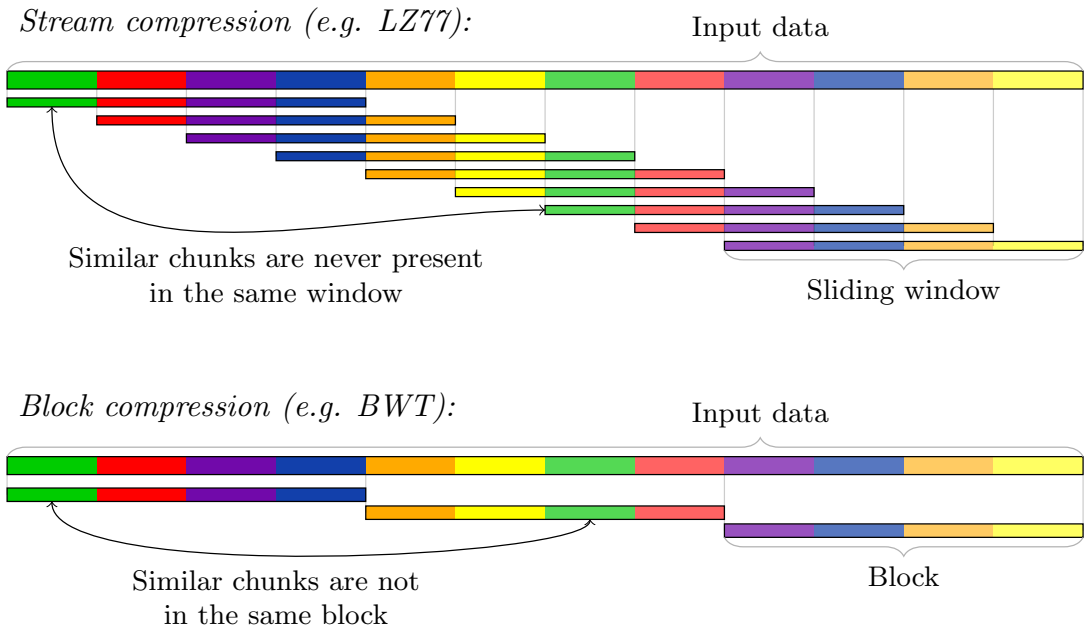


Figure 3.1: Example of a compression failure due to insufficient context of conventional stream and block compression methods. The input data resembles that of a `dual-calgary` dataset, where there are two copies of variable data. These copies are far apart enough for the compression algorithms to miss the redundancy.

In case of the solid compression, the whole dataset is compressed in a given order (the order of files in a `tar` for example). This is very advantageous if many small files are tared together as those are all compressed together and the redundancy is removed even among those files. However if those similar files are far apart, the context of the compressor has changed enough and the redundancy is not removed between these files.

The same applies for a single-file compression. If the file is long enough, and parts far apart of it are similar, the compressor will fail to compress those effectively. It only has two distinguishing scenarios compared to solid compression. The context of the compressor is reset with every file, meaning a superior compression ratio compared to solid compression can be achieved, just because there is no context overflow and suboptimal compression of another file with the compression context created on the previous file, but highly inefficient for the current file. The opposite case lies in the previously mentioned fact that single-file compression will not remove redundancy between files at all, even if there would be a good scenario of consequent similar files.

ICBCS tries to overcome the disadvantages of solid and single-file compression by being smart about how to put files together to achieve the best compression rate possible. It actually does not put whole files together, but only small chunks that are extracted from input files. Doing this ensures similar files are always close to each other and that the compression of those is as much effective as possible.

3.1.2 Objecting Binary Simhash and Minhash

Many systems for similar documents search use a set of techniques: Similarity hashing (Section 2.6.8), min-wise independent hashing (Section 2.6.9) and locality sensitive hashing (Section 2.6.10).

The same techniques are also used in extended deduplication systems, where files detected as similar are then compressed together. This is mostly done with a delta compression. See Section 2.3 for overview of such deduplication and compression systems.

All these systems suffer from a common drawback: simhash and minhash are not precise enough. Both of those are binary vectors, and even if more of those are used to describe a chunk, the overall span of distances is too small to create a sufficiently diverse vector space for an effective clustering of all the data.

ICBCS tries to overcome this problem by extending the simhash into a larger vector space and clustering the entire data. This makes all the chunks to contribute to the redundancy removal, not only the most similar subsets.

3.2 System Design

The system consist of several layers and/or components. These layers should be replaceable to a certain extent, as the system was designed to be extensible and easily modifiable.

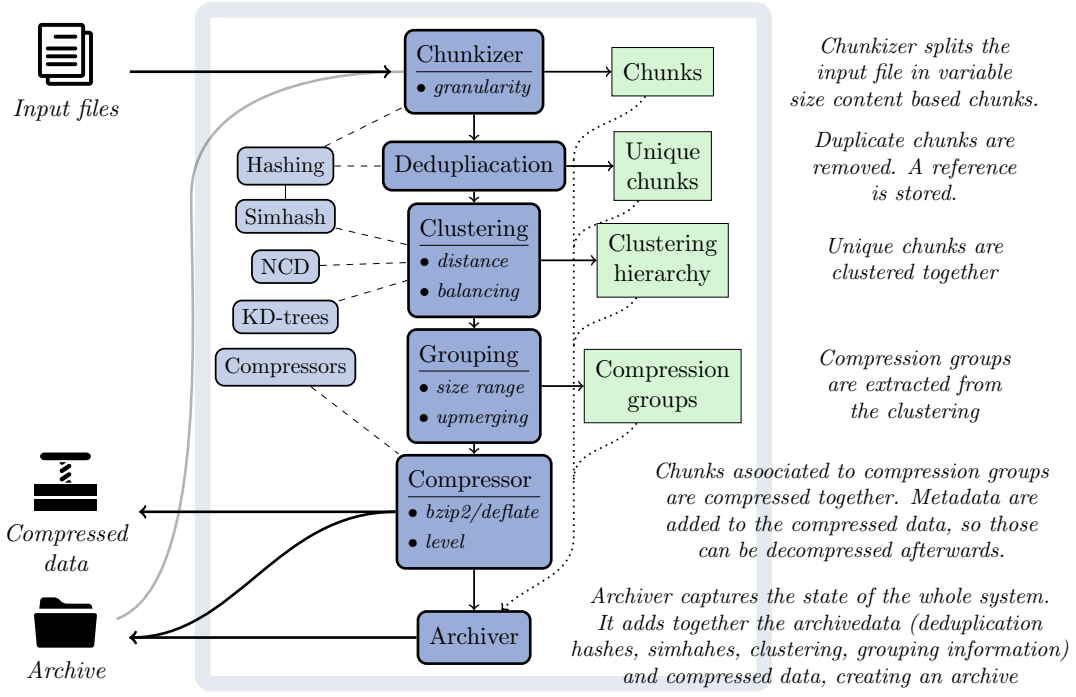


Figure 3.2: Scheme of the entire ICBCS system.

The first layer, *deduplication layer*, consists of the *Rabin chunkizer* that together with the *deduplication storage* take care of the deduplication process. This layer is described in Section 3.3 is a lite version of block based deduplication systems described previously

in Section 2.1.3. The deduplicating storage acts as an abstraction for the successive layers.

Second layer is the *similarity layer* described in Section 3.4. It provides an interface for distances between individual chunks. In case of simhash distances, it also implements a KNN – k nearest neighbors search algorithm mentioned and further referenced in Section 2.7.4. KNN is used to search for the most similar chunks already in the system, which are then used in the clustering layer for bottom-up clustering. NCD-based similarity layer does not allow for effective KNN search.

Third layer is the *clustering layer* described in Section 3.6. The layer does the essential incremental clustering of all unique chunks that made it though the deduplication layer. The layer itself takes care of all the balancing, distance and representatives updates. This layer is also responsible for proper assignment of chunks (leaf clusters) to compression groups. Lite version of the clustering layer that does batch clustering using NCD is described in Section 3.5.

The last layers is the compression layer described in Section 3.7. The compression layer uses a compressor to compress chunks within all compression groups. It also compresses the metadata and archivedata.

The whole system is responsible for coordination of these layers. It also keeps track of metadata and archivedata, the former is necessary for successful decompression of the entire dataset, and the latter promotes the compression system into an archival system by keeping track of archivedata – different files in the system, the current clustering and simhashes of all the chunks, etc. The archivedata is not necessary for the system to operate as an archiver, however without the archivedata, such system would be very ineffective, since complete decompression, recomputation of all the simahshes and clusterings would be necessary.

Figure 3.2 shows all the components of the ICBCS system.

3.3 Rabin Chunkizer and Deduplication

The method of content-defined chunking for deduplication was first used in [75]. It relies on Rabin fingerprinting by random polynomials [84] that uses a rolling hash similar to one of the possible rolling hashes to be used in Rabin-Karp multiple pattern string searching algorithm.

Since the technique is not new and well documented elsewhere, we will restrict the description to a bare minimum with the specific modifications of ICBCS.

The Rabin fingerprinting works with a limited size rolling window, for which the hash is computed. A rolling hash is capable of recalculating the window hash by hashing in the new incoming value and hashing out the last outgoing value. Let's call this hash a *window hash*. The scheme is depicted in Figure 2.3.

When the window hash $H(w_i)$ matches a predefined value called the *window pattern* P, a new chunk is delimited with its end in the point of windows hash match. The minimal size of a chunk W_{min} is then skipped – no chunk can be delimited here. If on the other hand no match is found and the maximal size of a chunk W_{max} is reached, the chunk is delimited at the maximal size. If a chunk of size $w < W_{min}$ is encountered at the end of the input, this one is appended to the previous chunks, regardless of the W_{max} constraint.

Figure 3.3: Comparison of fixed size (block) and variable size (chunks) deduplication. The former is especially vulnerable to inserted and removed characters, since all the block contents are shifter due to the change. The latter depends on a content of the data, so these operations do not pose a problem.

Through the entire input, another hash is computed – so called *chunk hash* H_{ch}. This one does not skip though blocks of minimal size. The chunk hash is used as a chunk identified in the deduplication storage.

The mask itself is constructed based on the desired chunk size. For uniform input and ideally spreading hash function, the distribution of H_w is also uniform over all windows w_i of string S:

$$\forall w_i \in |S| : Pr[H(w_i) = P] = \frac{1}{2^{|P|}}$$

that is, for any window w_i over the string S, the probability that the windows hash $H(w_i)$ matches the pattern P is inversely proportional to size of all the possible patterns of that length. Using any single pattern only exponential sizes in the form 2^x in expectancy can be achieved. The pattern can however be used for for so called boosting of average chunk size, further described in Section 3.3.3.

The pattern is then generated using the previous formula of matching probability. For example, to match $4KB$ in expectation, the pattern must be matched for 12 bits. Examples of such patterns for 32 bit hashing function are:

```
0100 0111 0010 **** **** **** **** ****
**11 **10 **00 **11 **11 **10 **** ****
```

The don't care symbols are achieved using AND masking of the window and a pattern template.

Chunk spread parameter determines the minimal W_{min} and maximal W_{max} window sizes. It is given to ICBCS as a left and right shift of the average chunk size. More information on that topic in the measurements of chunkizer in Section 4.4.

3.3.1 Deduplication Storage

The deduplicating storage works in a manner of hash comparison and subsequent full comparison. This is somewhat more complicated in archival manner, where the subsequent comparison is delayed, see Section 3.9 for details about that. Note that a full block/chunk hash is used in this case, not a window hash.

Both block and chunk deduplication follow the same duplicate detection algorithm, however over a different blocks, resp. chunks. See Figure 2.2 for a scheme of block deduplication and Figure 2.3 for variable size chunk deduplication.

Measurements and performance tests of both the Rabin chunkzier and deduplication are given in Section 4.4.

3.3.2 Performance and Optimizations

There are two major performance concerns within the chunkizer. First is an effective computation of the window hash by the vectorization of the hash computation over the window further ahead than by a single byte. This can be easily achieved since most of the operations are adding and multiplying.

Another optimization can be achieved by skipping the minimal chunk size. This is always done for window hash, however block hash is not skipped due to its negative impact on block hash and increased number of false-positives in duplicate detection.

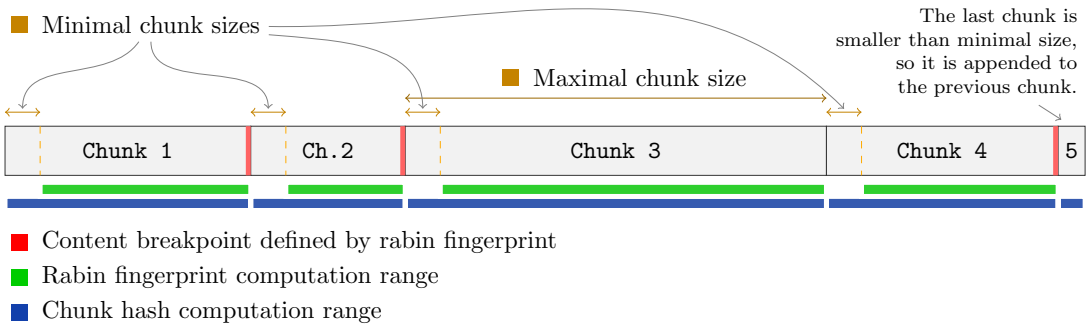


Figure 3.4: Chunk computation example. Due to the minimal chunk size, the Rabin fingerprint is only computed in the range starting with the minimal chunk size. Upon reaching the maximal chunk size, the chunk is cut, regardless of whether content defined breakpoint was reached.

3.3.3 Boosting Average Chunk Size

The chunk size distribution (ignoring the min and max size chunks) always converges to geometric distribution – negative binomial distribution with number of failure $r = 1$. This can be mitigated and pushed closer to negative binomial distribution with $r > 1$ by running several trials of the window hashing function $H()$. If say, we had two patterns P_1 and P_2, then by union bound:

$$\forall w_i \in |S| : P[H(w_i) = P_1 | H(w_i) = P_2] = 2 \cdot \frac{1}{2^{|P|}} = \frac{1}{2^{|P|-1}}$$

which implies if we use two different patterns, their length must be one bit shorter to achieve the same match probability rates.

However, having two different patterns allows us to select the one closer to out desired average chunk size. The same operation can be applied for any number 2^x of patterns of length $|P| - x$.

This optimization helps to reduce the disadvantages in the compression system cause by a large number of max width chunks, such as lower deduplication ratio, worse compression ratio and a higher processing time (assuming multiple pattern matching is properly vectorized and doesn't add a significant overhead).

Multiple patterns can also be simulated via different AND masks applied to the current window hash and a pattern template.

3.4 Extended Simhash

Previously used binary simhash is insufficient in terms of small vector space resulting in small precision distance evaluation. The discussion on this is in Section 3.1.2.

The main idea of *extended simhash* (further refered to simply as *simhash*) is to add more variability to the vector positions by extending those to real values instead of binary. The hashed features then contribute to the simhash in a similar manner is they do to the binary simhash. The contribution function of zero byte can be either noop or decrement, both ways are plausible and the final computation only differs in the normalization of the simhash. See Algorithm 3.1 on simhash computation over n-grams of string.

Algorithm 3.1: Simhash computation, where `sizeof` hash is the same as desired width of the simhash – *bijective* simhash construction, which is the simplest form with no merging.

```
Data: chunk (string), n-gram-size (int), width (int) /*hash size*/
Result: simhash (int vector)
Function Simhash (chunk)
    simhash ← vector(width, 0) ;
    for sub ∈ substrings (of length n-gram-size) of chunk do
        hash ← FNVhash(sub) // hash of single feature – sub ;
        for i ∈ {0..width − 1} do
            if hash[i] then // i-th bit of hash is set
                simhash[i]++;
            else
                simhash[i]−−;
    for i ∈ {0..width − 1} do // Normalization
        simhash[i] ← simhash[i]\length(chunk);
    return simhash
```

The overall value span of simhash is a subject to parameterization, although in the reference ICBCS solution, a 20 bit integer range was used. For the reasoning, please see Section 3.4.4.

The simhash can also be constructed from multiple sources, mutiple hashes or hashes of non-matching width, see Section 3.4.3 for more information.

Since the simhash occupies a much larger memory than in its binary form, reductions in width were implemented and tested. The simhash widths available in ICBCS are: 4, 8, 16, 32 and 64. The performance evaluation is in Section 4.3.4. The simhash width also has a significant impact on the archival properties of ICBCS, causing increased size of archivedata.

Simhash normalization is a process of transforming the values of simhash from simhash buffers for every position into the desired range of values. The process can vary a lot based on the destination number format, preprocessing time capacity, etc. In the simplest form – linear fit, the normalization looks like this:

$$\text{simhash}[i] = \text{simhashBuffer}[i] \cdot \frac{\texttt{SIMHASH MAX VALUE}}{|chunk| \cdot \frac{\texttt{HASH WIDTH}}{\texttt{SIMHASH WIDTH}}}$$

where the `SIMHASH MAX VALUE` speficies the desired simhash value range, and $\frac{\texttt{HASH WIDTH}}{\texttt{SIMHASH WIDTH}}$ is the overlay factor of the hash mapping, further specified in Section 3.4.3.

Unlike NCD, simhash is a vector in a vector space and thus allows for an effective simhash combination using a weighted vector average operation. This simhash combination is the core of simhash-based clustering.

Measurements Section 4.3 summarized the overall performance and various attributes of the simhash.

3.4.1 Feature Sources

Feature stands for a characteristic attribute of the input data. It is described in the context of clustering in Section 2.7.1 and it is one of the subjects of discussion in 2.6. The most typical features in data similarity scenarios are n-grams – see Section 2.6.7. Another very specific feature sources are the data compressors themselves. So called compression features are described in 2.6.6.1.

In ICBCS, only n-gram based features are used in simhash construction. Originally, a variable amount of n-grams was merged into a single simhash. The simhash was generated from these various n-grams with different weights for different n-gram size, for example: bi-gram: 0.2, tri-gram: 0.3, quad-gram: 0.2, 5-gram: 0.2, 6-gram: 0.1, or with different predefined position in the simhash vector, as described in Section 3.4.3.

Note that compressed strings cannot be effectively used for n-grams. Two slightly different uncompressed string can end up being completely different when compressed, especially whenever Huffman coding or other probability based codings are used.

Performance evaluation over different n-gram sizes was done in Section 4.3.3.

3.4.2 Hashing Functions

The hashing functions used to hash features should provide both fast speed and good dispersion. Some features only consist of several bytes of data, and some of those are not necessarily uniformly random.

The first hash function tested in ICBCS was the Rabin-Karp rolling hash (same hash function is used in the Rabin chunkizer). Although such hashing would provide ultimate speed performance, the dispersion was bad. Other rolling hash functions were used: hashing by cyclic polynomials and hashing by irreducible polynomials.

Summary of hashing functions used for the purpose of hashing n-grams was done in [25].

Rolling hashes, despite their superior speed, provided usually inferior results compared to the Fowler–Noll–Vo hash function [77]. The FNV is not a rolling hash function, meaning for n-grams, it has to restart the computation again for every n-gram position. See Section 4.3.1 for details on FNV and corresponding tests on simhash statistical qualities.

3.4.3 Merging Multiple Feature Sources

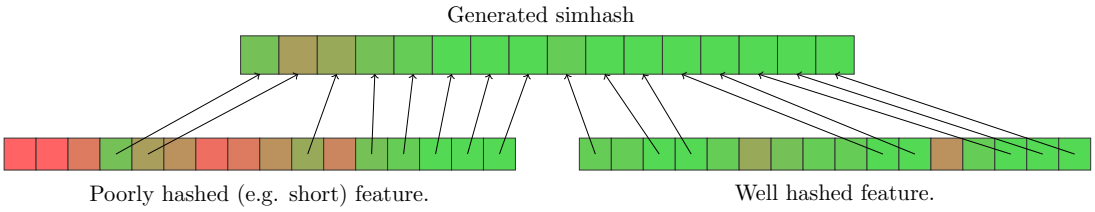


Figure 3.5: Injective hash merging. Some features (usually short) generate simhashes of poor statistical properties. Choosing the most variable hash positions allows for the final simhash to be both statistically variable and equally composed of the two features. Note that the positions cannot be change on the go, and have to be specified in the archivedata.

If the width of a simhash is the same as the size of hash generated by the hashing function used to hash features, no merging is needed. This is referred to as *bijective* simhash construction.

In ICBCS' reference setup, the default width on simhash is 32. However on 64 bit system, the FNV generates a 64 bit hash. In addition, if we have 2 feature sources, that is 128 bits of hashes we need to somehow map into the 32 positions of the simhash vector.

Note that of those 128 bits of hashes, some of those may be statistically more significant than others. If the hash function is not good enough, or the hashed feature is too short, the resulting hashed feature will be far from uniform distribution even for uniformly distributed data. This suggests that an *injective merging* needs to be utilized, where only the statistically most valuable hashed bits make it into the simhash.

Another method called *surjective merging* uses all of the hashed bits in the produced simhash. The form of the surjection further characterizes the merging.

Weighted surjective merging assigns each source with a weight attribute. Note that for binary weight, it is in fact injective merging. The weights can be determined by the statistical significance of the source hashed bits.

Modulus surjective merging assigns the hashed bit into the corresponding position of the simhash modulo the simhash's width. This is the merging used in ICBCS.

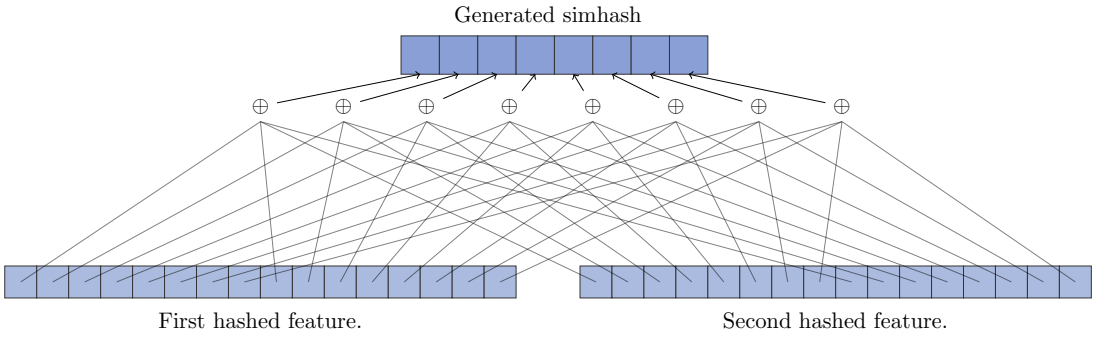


Figure 3.6: Modulus surjective hash merging used in ICBCS. All the individual hashes contribute equally to the final simhash. The merging operator can either be + or $\oplus$, where the former preserves more statistical variance, but requires larger simhash buffer as in Algorithm 3.1.

This has to be, however, prespeficied in the system before simhashes are generated. Since such solution is not adaptable, experimentation with the expected data and the hashing function has to be conducted.

Measurements on simhash statistical attributes is conducted in Section 4.3.1.

3.4.4 Integral vs Floating Point Representation

Due to performance reasons, floating point simhash was dropped from ICBCS during the optimizations further described in 3.11.2.

For floating point simhash, the distance computation is not limited by floating point precision drawbacks, namely the rounding error. If such error occurs in some simhash position, the distance is still well determined by the remaining positions. In case rounding error would have occurred in all positions, such simhashes would be then possibly misplaced in a group of similar simhashes, but due to the overall vector space, this is very unlikely to happen. Also, an impact of such imprecision is assumed to have minimal impact on the final compression scenario, although this has not been verified neither theoretically nor experimentally.

Integral simhashes on the other hand are much more likely to become victim of the limited precision. Especially for long chunk lengths, the simhash tends to converge to the mean, as was shown in Section 4.3.2. This can be mitigated by a logarithmic interpolation, where the values closer to mean would be spread more than those further away. However this would impose performance drag for both simhash generation, distance computation and combination.

Combining integral hashes nonetheless creates a problem. This is due to an optimization, where the division operation (of the weighted average) is not applied to the entire vector. More about this optimization and the possible issue in Section 3.11.2.

Performance evaluation remarks on integral vs floating point can be read in Section 4.3.6.

3.5 SLINK and NCD-SLINK Clustering

Of all the hierarchical clustering algorithms only the minimal linkage, often referred to as *single linkage*, is of interest to us. For compression, we try to achieve the min-wise pairing of the chunks, meaning the the most similar chunks are the closes to each other in the clustering.

This was not obvious in the beginning, as max-wise (*complete linkage*) clustering also provided reasonable results, but only in a small scale. In large scale, the system always degenerated fast. The same was true for average-wise clustering (*average linkage*), nonetheless with delayed degeneration.

Summary of hierarchical clustering algorithms is in Section 2.7.2.

The SLINK algorithm [96] is still the best algorithm for agglomerative hierarchical single linkage clustering, achieving a stable quadratic time complexity and a linear memory complexity.

SLINK together with NCD resulted in the best clustering achieved. The problem with SLINK its quadratic complexity, and the necessity to compute all pairwise NCDs. NCD computation itself is very time consuming.

That is the reason we tried implementation of NCD-based incremental hierarchical clustering based on a top-down approach (Section 3.6.1) and representatives (similar to representatives balancing in Section 3.6.5).

While the first SLINK method provided the best compression ratio results, it was unusable due to its time complexity. The representatives-based methods on the other hand failed to compress the data properly, due to missing min-wise similarities. More about the performance of NCD based clusterings, see Section 4.5.

The SLINK implementation successfully remains as an oracle of the capabilities of clustering to effective compression.

Note: An idea that has never been implemented or tested is to do incremental multidimensional scaling ([1, 109]) using NCD. The initial estimate for the multidimensional scaling would be based on simhash.

3.6 Incremental Clustering and Balancing

The incremental clustering algorithm used in ICBCS is not exactly any of other currently known algorithms. The clustering algorithm was tailored specifically for the purpose of subsequent grouping and compression.

The bottom-up version is inspired by agglomerative hierarchical clustering [108, 90] described before in 2.7.2. Conversely, the top-down version is inspired by divisive hierarchical clustering.

BIRCH clustering algorithm [116] uses a concept of clustering features to describe a clustering node withing a hierarchy. These clustering features are purely statistically based. It has inspired the ICBCS' combination of simhashes to represent the clustering nodes with a simhash alternative. There are statistical attributes at the ICBCS clustering nodes based on the extended simhash. Contrary to simhash, no clustering features can be extracted for NCD based distances.

For both top-down clustering using NCD and balancing of the clustering tree in any clustering mode, a method using representatives is used. These representatives are either leaf clusters or cluster nodes deeper in the hierarchy that describe a certain clustering

node. Clustering using representatives (per cluster) was first introduced in CURE [39]. This technique is further described in Section 3.6.5.

The bottom-up version is also inspired by DBSCAN [31], where KNN – k nearest neighbors are used to assign an incoming point to a cluster. In ICBCS, only 1NN is used. This is to comply with the SLINK property that goes well with the compression results of the system.

Unlike in DBSCAN, the sizes on final clusters seed to be rescaled, joined, split and altered. The hierarchical clustering allows us to do all this scaling easily with minimal overhead. The grouping into clustering groups is described later in Section 3.7.

We need single linkage properties in our clustering, but simhashes implicitly generate average link when combined together into clustering features similar to BIRCH. Deep and balancing and balancing with representatives partially overcome this.

Unlike in BIRCH, we also do not need a height-balanced tree. If the clustering actually ignores the depth balancing, the compression properties are hindered only minimally. However, it is shown in the measurements chapter (Section 4.6.1), that balancing the clustering tree to some degree is profitable especially for faster execution speed. Moreover, to a certain extent, balancing the tree can also guarantee a logarithmic complexity of a new chunk addition.

The hierarchical clustering of ICBCS works as a binary tree. There is no reason not to use higher order trees, however for implementation ease (especially balancing and representatives), the binary version was chosen. The tree's leaf nodes represent individual chunks from the Rabin chunkizer, sometimes these are referred to as *chunk clusters*. Every cluster remembers its total size (the number of chunk clusters in the subtree) and the distance between the left and right subclusters. Optionally, the cluster also has a set of representatives assigned to it.

The *distance* is a function of two clustering nodes. Ideal distance gives the true single-linkage (minimal) distance between these two clusters. However, this is only true for full depth or full representatives simhash or full representatives NCD.

$$dist(C_1, C_2) = min\{dist(\ell_1, \ell_2) : \ell_1 \in leaves(C_1), \ell_2 \in leaves(C_2)\}$$

where the $dist(\ell_1, \ell_2)$ is a distance between corresponding chunks of the leaves (chunk clusters). This can either be a pre-specified Minkovski norm of the simhashes (see Section 2.6.1) or the NCD 2.6.5.1.

Note that for simhash $dist(C_1, C_2)$ can be evaluated directly, because every inner node has a simhash assigned as a weighted combination. Such distance however represents the average-linkage (average distance) between the clusters. This is the default distance for ICBCS.

Several improvements of the distance measures exploit the clustering structure to achieve better clustering that is closer to a single linkage. These are described in Sections 3.6.4, 3.6.5 and 3.6.6.

3.6.1 Top-down clustering (NCD or Simhash)

To place a new chunk cluster, the top down clustering algorithm starts in the root and recursively searches for the closest subcluster to place the new chunk.

The position of placement is determined by the relative distance of the newly placed chunk cluster and the distance between clusters currently examined – the left and the

Algorithm 3.2: Top-down clustering. Single chunk cluster is placed into the clustering, starting from the top, recursively descending to the nearest subclusters.

```
Data: newCluster (chunk cluster to be placed), currCluster (defaults to root)
Result: newCluster is placed into the clustering
Function PlaceClusterTopDown (newCluster, currCluster)
    if currCluster.size = 1 then // Single node clustering
        return CreateParent (newCluster, currCluster);
    distBetween ← Distance (currCluster.left, currCluster.right);
    distToLeft ← Distance (newCluster, currCluster.left);
    distToRight ← Distance (newCluster, currCluster.right);
    if distToLeft < distBetween then // descend left
        currCluster.left ← PlaceClusterTopDown (newCluster, currCluster.left);
        currCluster ← UpdateCluster (currCluster);
        return BalanceTowardsRoot (currCluster);
    else if distToRight < distBetween then // descend right
        currCluster.right ← PlaceClusterTopDown (newCluster,
        currCluster.right);
        currCluster ← UpdateCluster (currCluster);
        return BalanceTowardsRoot (currCluster);
    else // current cluster has better integrity, place next to it
        if distToLeft < distToRight then
            parent ← CreateParent (newCluster, currCluster);
        else
            parent ← CreateParent (currCluster, newCluster);
        // Go back to top, balancing the clustering
        return BalanceTowardsRoot (parent);
```

right cluster. One of the four scenarios can happen: If the new chunk cluster is further to both the left and the right examined cluster, the placement recursively descends to the closer one. Else the chunk is closer to one of the clusters, so a new inner node is created and the closer of the left and the right cluster and the new chunk clusters are attached as children of that new inner node. The new inner node then replaces the closer left or right in the hierarchy.

This is better described in the Algorithm 3.2.

Once the new chunk cluster is placed, the algorithm proceeds back to the top, updating the clusters on the way with new size, distance between their children and optionally the list of representatives and the simhash as a weighted combination of its children.

3.6.2 Bottom-up clustering (Simhash)

Bottom-up clustering first finds the closest chunk clusters from the current hierarchy. This is achieved using a KD-tree (K-Dimensional tree). Simhashes of all chunk clusters are indexed in this KD tree. (Actually, multiple KD trees are used, more on this in Section 3.11.3.) Starting at the closest chunk's parent, the distance between the new chunk cluster we are placing and the distance between the nearest cluster and it's sibling

Algorithm 3.3: Bottom-up clustering. Single chunk cluster is placed into the clustering. Using randomized KD trees, the nearest chunk cluster already in the clustering is found. The new chunk cluster is then placed starting the the nearest cluster, ascending towards the root.

```
Data: newCluster (chunk cluster to be placed)
Result: newCluster is placed into the clustering
Function PlaceClusterBottomUp (newCluster)
    currCluster = FindNearestLeafCluster (newCluster);
    distToNearest = Distance (newCluster, currCluster);
    while do // Go up the clustering
        if currCluster is root then
            return CreateParent (currCluster, newCluster);
        parent = currCluster.parent;
        distBetweenParent = Distance (parent.left, parent.right);
        if distToNearest < distBetweenParent then // Place the new cluster
            if currNearest is left child of parent then
                parent.left = CreateParent (currCluster, newCluster);
            else
                parent.right = CreateParent (currCluster, newCluster);
            currCluster = parent;
            break;
    // Go back to top, balancing the clustering
    return BalanceTowardsRoot (currCluster);
```

is compared. The same operation of insertion as in top-down clustering is performed if the distance of the new chunk is closer than the distance between the closest chunk cluster and it's sibling, otherwise the algorithm proceeds to the parent node.

Once the new chunk cluster is in place, the algorithm proceeds towards the root, same as with top-down clustering described in Algorithm 3.2.

The entire bottom-up clustering process is shown in the Algorithm 3.3. An example is also depicted in Figure 3.7

3.6.3 Balancing the Clustering

In the described hierarchical clustering system, a disbalance may occur, where the distance between children of a certain node is smaller than the distance between one or both children of the children. This is best described by the Figure 3.8. The disbalance of cluster c can be described by the following formula:

$$\begin{aligned}\text{disbalance}(c) &= max(0, dist(c_{L_L}, c_{L_R}) - dist(c_L, c_R)) \\ &\quad + max(0, dist(c_{R_L}, c_{R_R}) - dist(c_L, c_R))\end{aligned}$$

where the $dist()$ is a distance function between clusters, C_{L_R} refers to the right child of a left child of C, etc.

When a disbalanced pair of cluster and its parent is found, a balancing operation is performed. This operation is similar to heap balancing, or a rotate operation in Red-

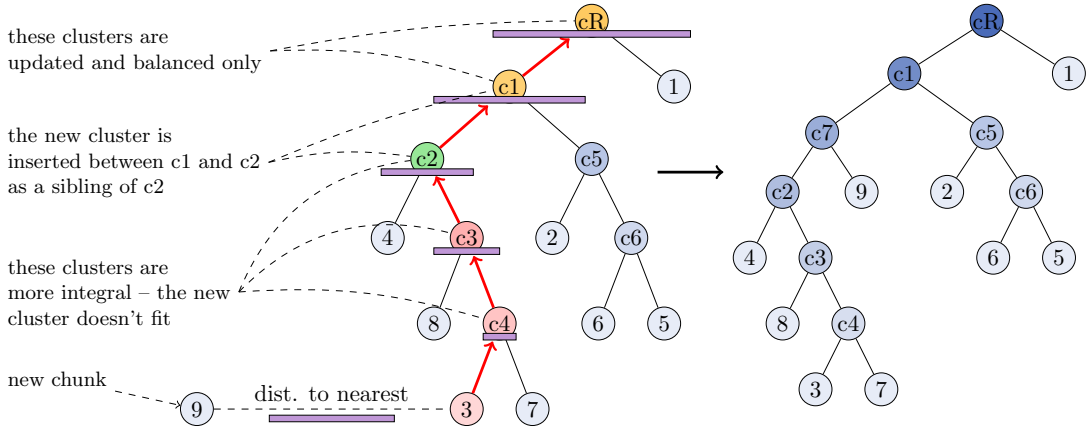


Figure 3.7: Example of bottom-up clustering. The purple bars denote intra-cluster distance, i.e. the distance between left and right subcluster. For details, see Algorithm 3.3.

Black trees. The disbalanced child is swapped with it's parent and gives to it's parent one of its child nodes – one that is closer to the former parent's other child. See Figure 3.8 for an example of the rotation.

The balancing algorithm can have several precision levels. The simplest, most shallow balancing is called *single balancing*. On the way from the newly placed chunk to the root, balancing is performed at most once in every node. This can of course leave disbalanced nodes behind, just as shown in Figure 3.8.

A parameter called *tolerance delta* can also be given to ICBCS. According to this parameter, the balancing operation is called recursively on both the currently balanced node (in case both left and right children are disbalanced) and on the rotated parent (that is now a child of a new parent). Overall tolerance t is accumulated for these recursive calls. The tolerance t is added to the distance disbalance formula, causing the disbalance to become ignored based of the recursive depth and disbalance of the node:

$$\begin{aligned}\text{disbalance}(c,t) \;&= max(0, dist(c_{L_L}, c_{L_R}) - dist(c_L, c_R) - t) \\ &\quad + max(0, dist(c_{R_L}, c_{R_R}) - dist(c_L, c_R) - t)\end{aligned}$$

For full description of the balancing algorithm, please see Algorithm 3.4.

The recursive (tolerance based) balancing may also run into an infinite loop, because when the subtrees are swapped and the distance invariant may swap with it. This does not occur for single-linkage and strict distance comparison. However note that combined simhash (unless of full depth or full representatives are used) is in fact an average-linkage and runs into the same problem. It is advisable not to use $tolerance = 0$ in any case.

The measurements in Section 4.6.1 show there is no reason to use excessive balancing anyway.

Balancing the hierarchical clustering by adding the size of the subtree to the distance function can result in implicitly more balanced clustering, however at the possible cost of separation of similar clusters due to being outweighted by small-size cluster. Primarily applicable in top-down clustering. This is just an idea that has not been implemented nor tested.

Algorithm 3.4: Balancing a single cluster. Variation of this function called `BalanceToRoot` simply traverses the clustering towards the root, balancing nodes on the way.

```
Data: newCluster (chunk cluster to be placed), deepBalancing (bool), tolerance
      (float), Δtolerance (float)
Result: curr is balanced, possibly its descendants too
Function BalanceCluster (curr, tolerance)
    distBetween ← Distance(curr.left, curr.left);
    distLeftBetween ← Distance(curr.left.left, curr.left.right);
    distRightBetween ← Distance(curr.right.left, curr.right.right);
    while distBetween < (distLeftBetween - tolerance)
          or distBetween < (distRightBetween - tolerance) do // disbalanced
        if distLeftBetween < distRightBetween then
            // curr.left and curr are disbalanced
            distLLtoR ← Distance (curr.left.left, curr.right);
            distLRtoR ← Distance (curr.left.right, curr.right);
            if distLLtoR < distLRtoR then // curr.left.left moves under curr
                curr ← RotateRight (curr.left, curr, curr.left.left /* curr.left.left
                swapped under curr */);
            else // curr.left.right moves under curr
                curr ← RotateRight (curr.left, curr, curr.left.right /*
                curr.left.right swapped under curr */);
            if deep then BalanceCluster (current, tolerance);
        else
            // curr.right and curr are disbalanced
            rotate left is analogical...;
        if not deep then break;
    tolerance ← tolerance ·(1 + Δtolerance) + Δtolerance;
```

3.6.4 Deep Distance (Simhash)

For simhash distance, it is possible to evaluate the distance between clusters using their children instead on the clusters themselves. By specifying the depth, the *deep distance* is then the minimal distance between all child nodes of the specified depth h (or of lesser depth for chunk nodes).

$$deep\_dist(C_1, C_2, h) = min\{dist(c_1, c_2) : c_1 \in desc(C_1, h), c_2 \in desc(C_2, h)\}$$

where $desc(C_1, h)$ are descendants of C_1 of depth up to h. The deep distance is evaluated in Section 4.6.2.

3.6.5 Representative Distance (NCD and Simhash)

Both NCD and simhash can also use representatives. The main idea is the same as with deep distance, representatives are used instead though. The set of representatives however has to be calculated. This can only be done from the representatives of it's

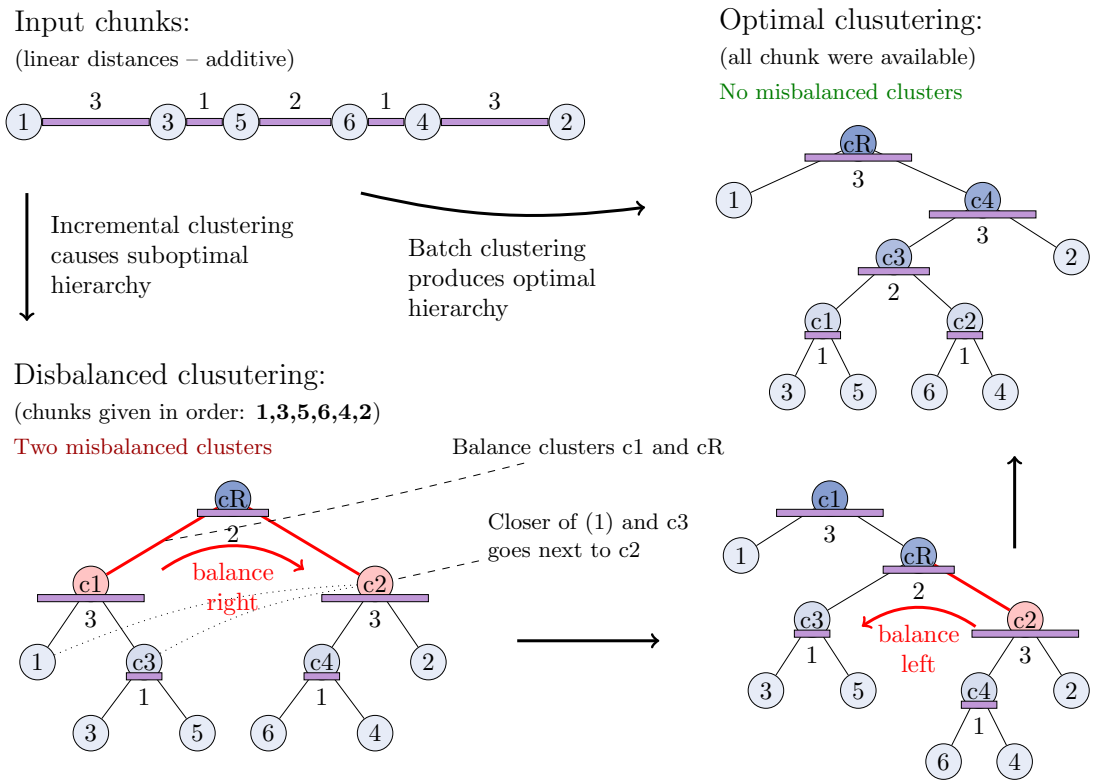


Figure 3.8: Disbalanced clustering example due to inconvenient order of incoming chunks. Two balancing operations take care of the misbalanced clusters.

children (or any depth). Note that the complexity of selecting the representatives grows exponentially with the number of desired representatives and the depth of candidates. In ICBCS, only direct descendants were used, i.e. depth = 1.

Determining representatives from the children of the cluster leads to so called *facility location problem*, where k-centers (representatives, facilities) are searched. There are several ways of doing so. For each cluster from the candidates, an average distance to all clusters is computed. The representatives are then selected from the candidates via a criterion. This is open for experimentation, solution in used in ICBCS is called *1min-max*, where one cluster with minimal distance to others is selected (this is a center), and then the rest of representatives is selected from candidates with maximal average distance to others (the border).

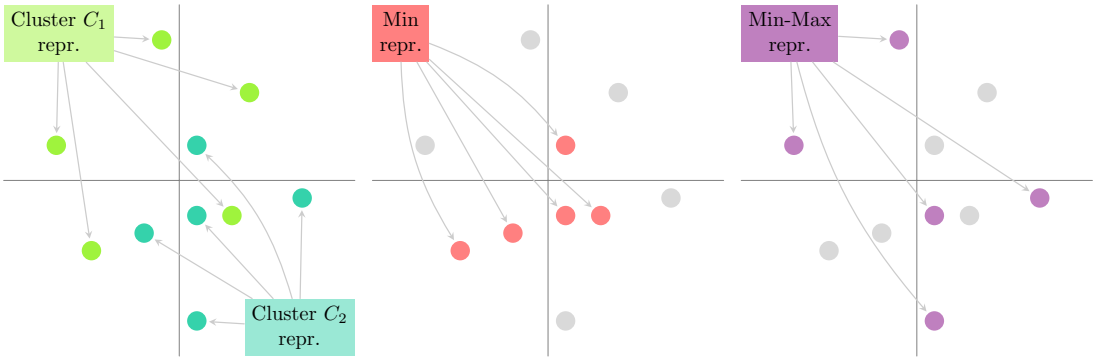
The *representative distance* is then similar to the deep distance, using minimal distance between representatives instead of the descendants.

For an example of representatives selection, look in Figure 3.9.

The representatives were tested for top-down NCD clustering in Section 4.5.

3.6.6 Deep Representative Distance (Simhash)

The *deep representative distance* refers to a combination of both deep and representative distance. It uses the representatives of descendant at a specified level.



Representatives of C_1 and C_2 | Min representatives of $C_{1,2}$ | Min-Max representatives of $C_{1,2}$

Figure 3.9: Example of representatives selection in 2D space. Representatives of cluster C_1 and C_2 (left) are merged using Min-wise representatives selection (center) and Min-Max-wise selection (right).

This distance has been implemented, tested briefly, but is not present in the evaluation.

3.7 Grouping and Compression

At any point of the clustering process, it is possible to determine compression groups from the current clustering hierarchy. This is usually done in the end of the clustering process in the compressor mode, or in the end of adding, editing or removing documents in the archiver mode.

A minimal compression group size parameter CG_SIZE_{max} is specified by the user. The hierarchical tree is traversed using DFS and subtrees of size

$$size(C) < \text{CG\_SIZE}_{max} = 2 \cdot \text{CG\_SIZE}_{min}$$

are assigned to a compression group. This traversal ensures that all compression groups CG_n are of size

$$\forall n : size(CG_n) \in [1, \text{CG\_SIZE}_{max}]$$

The compression groups assignment is depicted in Figure 3.10.

The subtrees' chunk clusters (leaves) are the chunks to be compressed within the compression group. Note that the order of chunk withing a compression group matters, as similar chunks need to be closer to each other for better compression results. This is easily achieved using preorder or postorder traversal. The order of compression groups does not matter.

See Section 4.8.2 for evaluation of different performance groups sizes.

3.7.1 Compression Algorithms and Levels

The set of compression groups is then compressed using a user specified compressor and compression leven. The only two compressors implemented are the `Bzip2` and `Deflate`, as representatives of a block based and stream based compression.

The compression algorithms and levels are evaluated in Section 4.8.

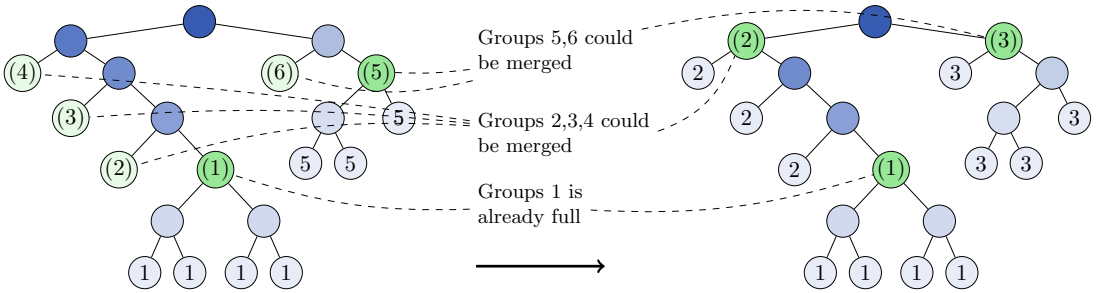


Figure 3.10: Compression groups with and without upmerging. On the left, no upmerging was used, total of 6 groups were created. On the right, with upmerging, only 3 compression groups were created.

3.7.2 Compression Groups Upmerging

Since the naive DFS compression groups assignment can result in compression groups of very small size, these groups can be merged together to achieve better compression results.

With *compression group upmerging*, the DFS traversal does not create a compression group unless a minimal group size $\cdot \text{CG\_SIZE}_{min}$ requirement is met or the root is reached. The parameter for minimal group size is deduced from the maximal size as: $\text{CG\_SIZE}_{max} = 2 \cdot \text{CG\_SIZE}_{min}$. This upmerging ensures that all compression groups CG_n are of size

$$\forall n : size(CG_n) \in [\text{CG\_SIZE}_{min}, \text{CG\_SIZE}_{max}]$$

The relation between minimal and maximal size as a multiple of two works well for binary trees.

The comparison of upmerging vs normal compression group assignment is written in Section 4.8.2.2.

3.8 Compressor Capabilities

The ICBCS can act as a compressor. Given a file as in input, it performs the chunking, deduplication, clustering, grouping and compression. The set of compressed groups is forms the compressed data.

For a decompression to be possible, there has to be additional information called the *metadata*, that is used to fully decompress the file.

Metadata not only allows a decompression, but also preserved the information about chunks and duplicates. Although a complete reconstruction of the whole deduplication state, clustering and grouping is possible, it would take a lot of unnecessary computation. Archivedata described in Section 3.9.1 serves that purpose.

3.8.1 Metadata

The metadata is a lite preamble to the actual compressed data. It contains the following information: file signature (magic number), number of compression groups and number of chunks, total size of original data, offsets of compression groups to compressed data block and offsets of chunks in the original data.

An example of 32 bit implementation of metadata used for measurements in Section 4.7:

```
0000: 0000 0123           // magic number
0004: 0000 1C28 002D 40AC // number of compression groups
                                  and chunks
0014: 0000 0000 06C1 229F // original data size
001C: 0003 DBB2 0013 6002 // offests of 1. group
                                  to comp. and orig. data
      .... .... .... ....
1C3C: 00E3 2421           // offset of 1. chunk
      .... ....
```

The magic numbers tells the decompressor that the file is in ICBCS format. Note that since Bzip2 or Deflate magic numbers are not stripped from the compressed groups, so there doesn't have to be any information about the compressor.

Number of compression groups and number of chunks are used to delimit the following lists of compression group offests and chunk offsets.

The compression group offsets to compressed data are designated to delimit the compression groups in the compressed data block. These can be effectively replaced by explicit ordering and size of the chunk, however implicit ordering requires only one information to be stored – the offset. Ordering the list by these offsets then create the full index of clustering groups.

The compression group offsets to original data don't have to be used for full decompression, however they provide the information in which compression group a chunk is placed, allowing for more effective decompression, where the compression groups can be decompressed selectively based on the chunks they contain. Without this information, chunk assignment to compression groups would only be possible after full decompression of all the groups.

Offsets of chunks are also given in the implicit order of chunks, so the same method or ordering by offets can be used to create the index of chunks by the original data position. Duplicate chunks share the same offset.

3.9 Archival Capabilities

The ICBCS also has archival abilities. It can retrieve particular files, update files, add new file and remove existing files from the archive. These are called the CRUD operations (Create, Read, Update, Delete).

This is achieved using and extended metadata called *archivedata*.

None of the archival capabilities were implemented in ICBCS.

3.9.1 Archivedata

Archivedata serializes information about the clustering and grouping, namely the clustering hierarchy, compression groups, simhashes of chunks and deduplication hashes of chunks. The archivedata also contains a set of files in the archive and for each such file, an ordered list of chunks it consists of.

The format and scale of information in archivedata has to been finalized for the ICBCS. It still a partially open problem, especially to find the right balance between information saved in archivedata and the cost of archive CRUD oeprations.

The forementioned archivedata format was used for preliminary measurements in Section 4.7:

3.9.2 Retrieving Documents

Retrieving a document is a process of finding all the compression groups of chunks the file consists of. These compression groups are then decompressed and the original file is reconstructed. Note that the order of the retrieval of the parts does not matter, however it is advisable the follow the order of chunks to be more likely to achieve a sequential output of the file.

3.9.3 Adding, Editing and Removing Documents

In case of altering operations, the clustering gets possibly updated. Let's also consider the edit operation to be a sequence of deletion and addition.

The clustering update operation all follow the same patterns or adding or removing a chunk cluster and then updating the structure on ascending to the root. The major concern here is to recompute the compression groups. This is also dependent on whether compression group upmerging is enabled. In any case these operations either edit, split or merge two compression groups, while scheduling these groups for recompression.

Delayed recompression refers to a method of recompressing the changed group only after a completion of all scheduled operations, e.g. after adding all the files to the archive.

Delayed duplicate verification refers to the same method, however for duplicate verification. In case hashes match, we need to compare the uncompressed chunks. To get the archived chunk, it would have to be decompressed, compared and compressed again. Instead, save the incoming chunk as it is and add it to the list of duplicate verification files. The first time the cluster containing the potential duplicate is reclustered (decompressed), we do the verification.

3.10 Implementation Notes

ICBCS was implemented in `C++11`, using mostly `STL` for a lot of internal data structures and basic algorithms. The std c++ library used was that of `GCC 4.8.2`.

CLI interface was implemented using `boost::program_options`. The complete CLI is listed in Appendix B.

Compressors that were implemented in ICBCS both used the newest version of their libraries: Deflate [61] and BZip2 [93]. For randomized KD-tree the FLANN [74] library was used. And for rolling hashes a small implementation called `rollinghashcpp` [28] was used.

Several other libraries were used for intermediate implementations, however these were dropped in the final version. These libraries include: Eigen (linear algebra), CompLearn (compression based data mining from [23]) and ANN: A Library for Approximate Nearest Neighbor Searching.

The build system `CMake 2.8`. Several useful options for dependent library setup and profling option using gprof are documented in the CMakeList file.

A source code snapshot of the version that was used for measurements and writing this thesis is provides on the attached disk. The ICBCS source repository has not been made public by the time of writing the thesis.

3.11 Performance and Optimizations

Major performance-wise optimization were conducted on ICBCS. The drawback of this system (any basically any deduplication system) is the need for complete, or almost complete scan of the input data and excessive computation of hashes. Thereby the most effort was targeted at the optimizations of computation and manipulation with hashes. The best performance gain were due to vectorization via SSE (Streaming SIMD Extensions) available to our hardware configuration (Section 4.2).

Many optimization directly implied by GCC's `-O4` are not described, except for cases where the user made code optimization leads to more effective compiler optimizations.

Profiling output can be easily generated by specifying `PROFILING=ON` in CMake and running the `./generate-calltree.sh`. A png image of a call tree will be generated.

3.11.1 FNV Hash Vectorization

The first performance drag was caused by the FNV hash, described in Section 3.4.2. Since rolling hashes have generally failed to create good simhash vector space.

The idea is to compute n FNV hashes at the same time. The n is based on the size of n-gram, e.g. for 4-gram, 4 hashes are computed. The implementation is based on a cyclic array and vectorized XOR and multiply operations. This is also the reason why there was no significant decrease of computational speed for higher sizes of n-grams in Section 4.3.3.

The FVN computation could of course be extended to a wider vectorization. As of the current implementation, FNV computation is still the performance bottle neck.

3.11.2 Simhash Optimizations

Another computation heavy part of ICBCS is its simhash vector system, described in Setion 3.4. These vectors not only have to be computed (once per each chunk), need to be combined during clustering updating, but mostly need to be compared all the time.

Simhash data structure is implemented using `std::valarray<uin32_t>`. Valarray was chosen due to its minimal overhead and ease of vectorization.

The computation of a simhash uses relies on a maximal vectorization. For this to take effect a width of a simhash has to be known. However this is an input parameter of ICBCS. Therefore a maximal loop unrolling withing a switch for all simhash widths was used. A switch is generally smaller than that of a virtual method call (on a tested platform). There are also no multiplication operations in simhash creation.

Just as the computation of a simhash, the distance evaluation between two simhashes was optimized in a similar manner. The distance evaluation is the most often function called in ICBCS. Full vectorization of the $Minkovskinorm1$ distance is used, with a subsequent sum over all the positions.

Simhash combination optimization was a little more difficult task. If simhash implementation used floats and its datatype, such combination would be a simple weighted vector average:

```
float lweight = weigthLhs/(weigthLhs+weigthRhs),
      rweight = weigthRhs/(weigthLhs+weigthRhs);
*out = (lhs*lweight + rhs*rweight);
```

however simhash intepretation uses integers, as floating point multiplications are too slow. Using the weights (`lweight` and `rweight`) as integers would however yield zeroed results after the cast to integers.

For this reason, a *weight offset* is used, to promote weights into a higher order, where cast to integers results in only a little biased values.

```
int lweight = (int)((weigthLhs*(1<<woffset))
                        /(weigthLhs+weigthRhs)),
    rweight = (int)((weigthRhs*(1<<woffset))
                        /(weigthLhs+weigthRhs));
*out = (lhs*lweight + rhs*rweight) >> woffset;
```

On 32 bit platforms, the multiplication of the simhash with a desired weight can cause an easy overflow. The `woffset` thus has to be regulated based on the weight. On 64 bit platforms, this is not a problem. In ICBCS run on 64 bit platform, 20 bits are used to store the value, 8 bits for weight offset, the rest easily accommodates high weigh values.

With integer-based simhash, vectorization and no division, all simhash operations are extremely fast and no longer pose a performance bottleneck.

3.11.3 Randomized KD Trees

The KNN problem or its approximate variation ANN is easily solved using KD-trees for lower dimensions. See survey in Section 2.7.4, 2.7.6.

This was however mitigated using several randomized KD-trees and a limited number of leaf searches. The parameters could be a subject to further parametrization, nonetheless for ICBCS testing, these were fixed to 4 KD-trees and 32 leaf searches as defined by the FLANN library used: [73].

Chapter 4

Evaluation

This chapter covers all of the parameterization and measurements done on ICBSC. There are many parameters in the system, but some of those play more significant roles in the resulting performance of the system. For a complete list of input parameters, please see the ICBCS usage in Appendix B and for a complete overview and explanation of measured attributes, please see Appendix C.

The process of many experiments resulted in a *reference setup*. This setup was then used for further parameterization and measurements presented in this chapter. Note that due to a very high number of parameters of ICBCS, it would have been extremely difficult to find an optimal solution using exhaustive parameterization.

The ICBCS reference setup	
Compressor	deflate
Compression level	5
Average chunk size	4 KB
Chunk size spread	1–16 KB
Deduplication	ON
Distance	simhash
Simhash width	32
Simhash n-gram	4
Compression group size	64
Balancing	no balancing
Upmerging	ON

For visual aids, many plots are available to the reader. These usually fall into several categories based on their purpose. Line plots are used in numerical type single variable plotting – e.g. simhash width, compression group size, compression level, etc. Bar plots are used for categorical variable, e.g. balancing method, or type of distance measure. To display multidimensional numerical variables, 2D color plots are used, e.g. for simhash histograms and chunk size and spread plotting. Tables are used to display multiple variables of interest where desired, but usually on a smaller scope (e.g. just one dataset). Overall, the visualization should be always be explained either in the text or in the caption.

`dataset`	*description*	*redundancy*	*size*	*habitat*
small	Small text file	≈ 90%	55 KB	artificial
calgary	Calgary corpus	≈ 70%	3.2 MB	corpus
canterbury	Conterbury corpus	≈ 70%	2.7 MB	corpus
dual	Two compressed PDFs	≈ 50%	1.9 MB	real
dual-cal	Two Calgary corpora	≈ 85%	6.3 MB	corpus
random2	Random 2 MB data	≈ 0%	2 MB	artificial
prague	Prague corpus	≈ 50%	56 MB	corpus
em	Confidential documents of FIT CTU	≈ 98%	21 MB	real
reqview	Reqview documents	≈ 60%	19 MB	real
athens	ATHENS reports	≈ 5%	132 MB	real
random	Random 64 MB data	≈ 0%	64 MB	artificial
linux-kernels	Linux kernel sources (3.0, 3.1, 3.2)	≈ 90%	1.3 GB	real

Table 4.1: Summary of datasets

The chapter first goes through the datasets used in most of the measurements (Section 4.1), then briefly describes the testing platform (Section 4.2). Next, the simhash is thoroughly analyzed (Section 4.3), followed by the deduplication layer (Section 3.3), a comparison of simhash and NCD (Section 4.5), clustering quality (Section 4.6), metadata and archivedata overhead (Section 4.7), as well as underlying compression algorithms (Section 4.8). Finally, memory requirements are briefly mentioned (Section 4.9) followed by overall performance summary of ICBCS (Section 4.10).

4.1 Datasets

The selection of datasets is quite different from a selection of datasets in most compression algorithm papers. The major difference lies in the fact that conventional corpora consist of files of different types. The variability of these files and low similarity between these files make them bad candidates for an effective large-scale compression system such as ICBCS. Although the power of ICBCS cannot be effectively demonstrated on variable file sets, several of those were selected, since corpora stand for reference platforms.

Other than that, other interesting datasets were selected that demonstrate various properties of ICBCS. An overview of datasets is in Table 4.1.

4.1.1 Small single files

The `small` dataset is only a small text file with few repeating characters and words. Its main purpose is to demonstrate the degeneration of ICBCS to simple compression algorithms in case of small files. Additionally, this dataset demonstrates that reordering

of chunks within a single compression group can have a positive effect on compression ratio – when using Deflate.

4.1.2 Corpora

Three corpora were selected for the tests, the `calgary` [4], `canterbury` [7] and `prague` [45] corpus. Since these are extremely variable data with only a little explicit duplicates, the compression effectiveness of ICBCS is not well-demonstrated on those. The usage of these corpora is also not conventional, because the whole single corpus is used as an input itself, not the files it contains.

4.1.3 Duplicate and Highly Redundant Data

The opposite of corpora is duplicate data. The `dual` datasets consist of exactly two identical compressed PDF files. The `dual-calgary` dataset consists of two `calgary` corpora. The major difference between these two is that there are no intra-file redundancies in the `dual` dataset. The PDF files do not allow for any further compression, however appropriate mapping of these files on top of each other results in perfect deduplication. The `dual-calgary` ,on the other hand, still contains a lot of intra-file redundancies and thus allows for further compression within the compression clusters.

The dataset `em` consists of 9 unzipped `docx` documents. The documents are all version of application forms from the Faculty of Information Technology at the Czech Technical University (CTU). Please note that this dataset is confidential (also top redundant) and so it is not attached to the thesis.

4.1.4 Similar and Moderately Redundant Data

The dataset `reqview` consists of about 50 documents generated using the requirement managing software – ReqView. The documents are similar to each other though parts are encrypted and effective compression is not viable there.

Three complete Linux kernel sources (version 3.0.101, 3.1.10, 3.2.56) were packed into the `linux-kernels` dataset. This dataset not only exhibits a very high amount of duplicates, but lots of the files are merely different version of each other.

4.1.5 Random, Compressed and Image Data

This last category of datasets consists of uncompressible files. The datasets `random2` and `random` are randomly generated files of size 2 MB, resp. 64 MB. These were produced using `dd if=/dev/urandom of=random bs=1M count=64`. `random2` is used few scenarios, where `random` would result in too long execution time.

The last dataset `athens` consists of all ATHENS student reports from November 2013.

4.2 Testing Environment and Hardware

All tests were run on a laptop, however, we have created an environment, where the interference to ICBCS caused by other processes is minimized using several techniques described further in this section.

The hardware used had the following configuration: Intel Core i5-2520M CPU @ 2.50GHz x 4 (2 physical cores, Hyper-threading), SSE4.1 instruction set, 8 GB of physical memory. The operating system is 64b Linux – Fedora 19 (3.10.10-200.fc19.x86_64). As a compiler, we used GCC 4.8.2.

To ensure maximal CPU-wise performance and constant efficiency, the kernel was started with `isolcpu=1,3` (1,3 are virtual cores that correspond to physical core 1) – this disables two cores, so that the kernel task scheduler will not utilize them. Then, every ICBCS test was run with assigned affinity using `taskset 0xA` on this physical core.

Due to the frequency scaling of this CPU, a performance governor was set for this CPU. This requires a `cpufreq` kernel module. A `performance` governor was then set for CPUs 1 and 3. This governor setup is very likely also available via a GUI application.

Since CPU exclusivity is not enough for our process, for time-oriented tests, several runs (8x) are made and the minimal time value is then selected. For time measurement, C++11s std::chrono and boost::chrono libraries were used to measure wall time, user time, system time and thread time. For all plots and tables used, the sum of system and user time is used.

4.3 Simhash Distribution

The simhash creates a vector space in for the compression system. In order for the distance measures not to degenerate and fall victim to limited floating point or integral precision, we want the vectors to occupy as much of the space as possible. This is closely relevant to hash diffusion, that describes high hamming distance between hashes of two similar inputs.

The hashes produced by the used FNV hashing function have a width of 64 bits, it is necessary to provide enough input to the hash for it to uniformly spread the input values into these 64 bits as much as possible. If the desired simhash vector size is less than 64, uniformity can be achieved by hash injecting methods described in Section 3.5, where lesser amount of input is sufficient.

Extensive comparison of n-gram hashing function is out of the scope of this work. This section's main purpose is to introduce an insight into how the hashes affect the simhash vector space and subsequently the whole compression system.

4.3.1 Simhash Distribution Tests

To represent data without any induced bias, the following tests were run on `random` dataset. To represent real world data, the `prague` corpus is used. Both dataset are of similar size.

The former symbolizes random data, where the randomness is diluted with increasing chunk size, because computation of simhash for a chunk uses a sum (or an average, based on normalization factor) of all n-gram hashes within a chunk. For high chunk sizes, the resulting simhash distributions will always converge to the geometric distribution. This makes sense because random data cause the chunks to have the same set of n-grams with increasing size, and we only have a single hash to describe the data.

The latter dataset symbolizes real-world data that are sufficiently diverse but also exhibit high locality. It is very likely that data in a single chunk will be similar within

the chunk, but not within the whole dataset. This results in higher diversity of the simhashes than in case of the random data.

Note: The 2D histogram plots represent the distribution of simhash values (over the entire dataset). Single column of the plot represents single simhash vector position. If we are concerned about the 32 byte long chunks, this means that the simhash can have $32 - (ngram - 1)$ different values. Each column has 16 buckets, so in the case of 128 byte chunks and 6-grams, each bucket represents value span of 7.7.

The first measurement was done with only a single input byte (1-gram) for a desired vector size of 32 bits. It is obvious that one random byte cannot generate 32 independent bits of information and the expected result is for the simhash distribution to be extremely biased. The form of this bias is determined by the hash function used, e.g. the FNV's offset basis and prime. Figure 4.1 shows the resulting histograms for 32 bit hash. Several of the hash values have zero variance – the corresponding bit is always the same.

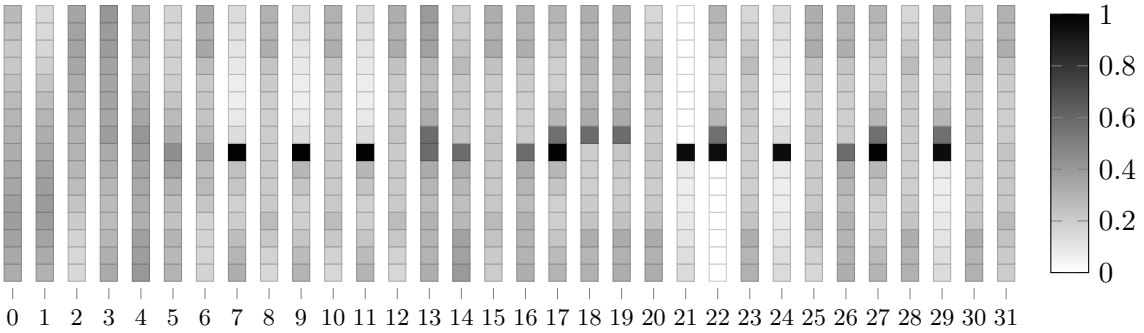


Figure 4.1: Simhash distribution over all simhashes generated from the `prague` corpus. N'th column indicates the distribution of values (histogram) in the n'th simhash vector positions. The values in single cells correspond to the amount of simhash values in that range. Simhash vector positions with uneven distribution are degenerate and provide poor distance properties within the vector space. FNV hash, 1-gram, fixed size of a chunk – 32 bytes, histograms not zoomed – show the entire values range.

Second example illustrates that to fully use the span of 32 bit hash, at least 5-grams have to be used. See Figure 4.2. Simhash histograms for ngram values of 2 to 4 are omitted – they represent a transition from Figure 4.1 to Figure 4.2 where the amount of misbehaving simhash positions is decreasing. For the sole purpose of hashing, n-gram of 5 is then sufficient, however it doesn't say anything about it's appropriateness for compression purpose. Since much more than simple n-grams may be used for simhash computation, it is necessary that these features provide at least 5 bytes of information in order to be effectively hashed into 32 bit hash using FNV. See Section 3.5 for details of how to combine hashes from individual sources.

For the previous two examples, fixed size of 32 bytes per chunk was used. This is very high granularity that is never used in the final setup. It was chosen merely to demonstrate the behavior of the hash function on different input sizes. Also, fixed chunk size does not allow for extensive locality. Similar data that would otherwise be placed in the same chunk are separated, resulting in artificially higher variance.

Next example illustrates a real world setup on the `prague` corpus. The data are quite variable and so are the resulting simhashes. Figure 4.3 demonstrates the simhash distribution with a size span of 256 B to 1 KB and 5-grams. Note that even though every

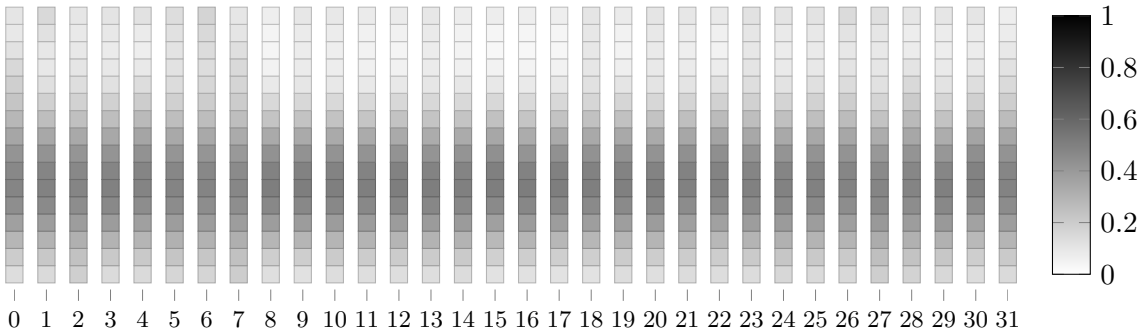


Figure 4.2: Simhash distribution over all simhashes generated from the `prague` corpus. 5-grams were used to compute a single hash in all the simshashes. This is sufficient to generate well distributed values over all positions of the simhash. The obvious shift into lower values is caused by the 5-gram requiring 5 consecutive bytes – only $32-5+1=28$ possible values could be generated on every position, but the normalization factor ignores this. FNV hash, 5-gram, fixed size of a chunk – 32 bytes, histograms not zoomed – show the entire values range.

simhash is summed (or averaged) over single 5-gram hashes, the resulting distribution is still very variable. FNV hash has proven to be very successful in the simhash computation on variable data.

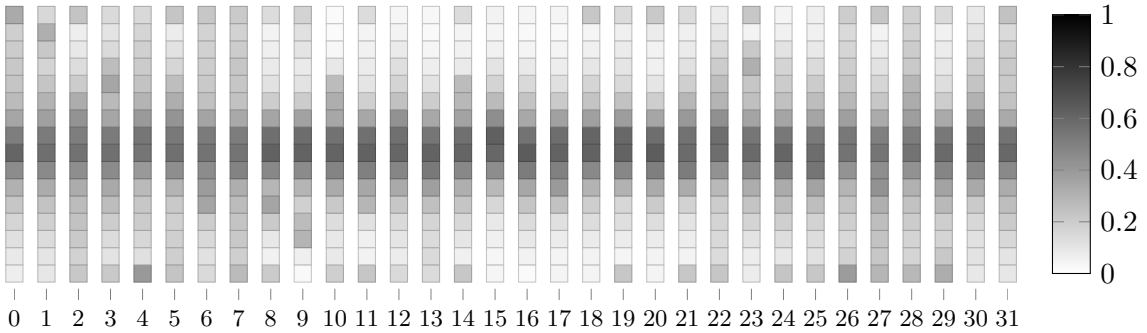


Figure 4.3: Simhash distribution over all simhashes generated from the `prague` corpus. For these simhash computations, a larger chunk size was used – span of 256 B to 1 KB. The Rabin chunkizer split the data according to content and these were then hashed. The variance of the simhashes is very high, since the prague corpus has variable data and these were chunked based on their content. This figure demonstrates the typical simhash distribution in the ICBCS system. FNV hash, 5-grams, variable size of a chunk – 256 B to 1 KB, histograms zoomed into the middle 50% values – overflow is part of the edge buckets.

The fourth example illustrates the same setup, however with `random` data, see Figure 4.4. All the simhash positions are very similar and massed around the center with their respective geometric distributions.

The FNV hashing function did not fail in this case. Average size of 512 bytes of uniform and independent data resulted in very similar chunks, that were subsequently hashed into very similar hashes. Simhash cannot deal with random data. Unfortunately

it cannot deal effectively even with partially random data, because the hash computation itself uses randomness to describe the chunk and so a little amount of information is likely to be lost.

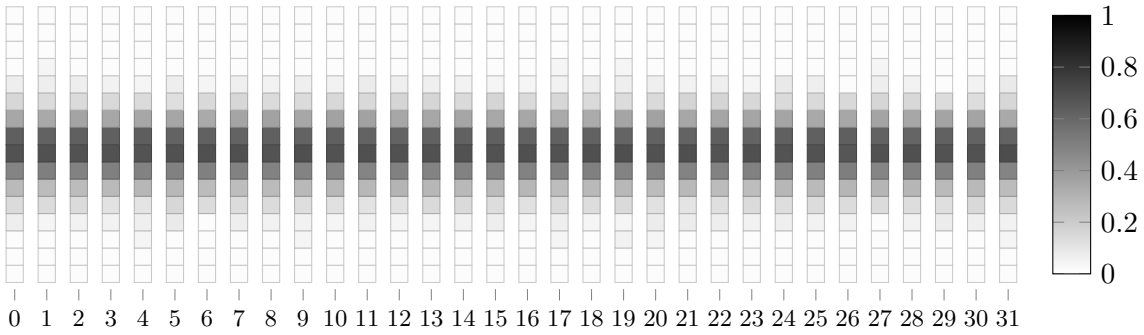


Figure 4.4: Simhash distribution over all simhashes generated from the `random` dataset. In this case, with a size span of 256 B to 1 KB again, the data in every chunk is 512 random bytes long on average. With only 32 byte simhash and random data, all the simhashes are very similar. FNV hash, 5-grams, variable size of a chunk – 256 B to 1 KB, histograms zoomed into the middle 25% values – overflow is part of the edge buckets.

Other fast hash functions than FNV do not work as well for the hashing. Not all hash function are applicable due to performance reasons, see Section 4.3.5 for more information. Other tested hash functions were Karp-Rabin, Cyclic Polynomials and Irreducible Polynomials from [28]. Calculating hashes directly from bit representation of 4-grams was also tested. None of these did as well as FNV in terms of simhash distribution and resulting chunk matching.

4.3.2 Simhash Variance

To measure the quality of similarity hashing, we can use standard statistical measures such as mean, standard deviation and variance.

As mentioned in the previous section, the chunk size plays the major role in degeneration of the simhash. The biggest concern with the degeneration is of course the loss of precision in distance computations, because the system does not use floating point to represent the dimension of a simhash. See Section 4.3.6 for detailed discussion.

We have measured these statistical variables for the `prague` and `random2` dataset and various chunk sizes. The results are present in Table 4.2. Even for large chunk sizes and the (very variable) `prague` dataset, the minimal distance between simhashes encountered was still more than sufficient for precise computations.

The standard deviation and variance, however, exhibit a steady decline for larger chunk sizes. This could potentially result in imprecisions. These could be encountered by either increasing the simhash size (the experiment was done with simhash value range of only $1 - ()2^{20} - 1$) – that is mere 20 bits), or by logarithmically scaling the simhash values. Unfortunately, both of these fixes would result in performance impairment. And performance was the reason we use integers instead of floats.

dataset	*Simhash distances*		*Average over all positions*		
Size	*Min*	*Max*	$\mu(mean)$	$\sigma(std.dev.)$	$\sigma^2(variance)$
`random2.dat`					
64 B	163 746	6 122 980	483 293	44 486	1.97e+09
256 B	61 575	4 528 186	513 999	22 928	5.25e+08
1 KB	53 498	16 459 776	521 432	16 300	2.65e+08
512 B – 2 KB	29 607	5 690 710	521 666	12 395	1.54e+08
4 KB	39 876	1 607 291	523 614	5 785	3.35e+07
2 KB-8 KB	68 582	2 663 550	523 636	5 691	3.24e+07
16 KB	51 200	1 436 115	524 093	2 839	8.08e+07
8 KB – 32 KB	26 674	868 198	524 084	2 748	7.59e+07
4 KB – 64 KB	27 615	1 085 114	524 072	3 160	1.00e+07
`prague.tar`					
64 B	8 160	11 948 191	482 803	63 160	4.08e+09
256 B	2 048	11 269 422	513 445	53 284	3.08e+09
1 KB	6 656	8 778 744	521 211	47 368	2.55e+09
512 B – 2 KB	1 024	10 634 866	521 004	45 596	2.33e+09
4 KB	2 176	10 076 414	523 296	46 673	2.53e+09
2 KB – 8 KB	2 364	10 622 315	523 148	44 219	2.24e+09
16 KB	768	7 215 461	523 850	46 021	2.49e+09
8 KB – 32 KB	3 337	8 168 472	524 210	45 558	2.44e+09
4 KB – 64 KB	1 699	10 812 962	524 436	47 234	2.66e+09

Table 4.2: Simhash min and max distances encountered and statistical mean, standard deviation and variance. Simhash range span: 0–$(2^{20} - 1)$ (20 bits).

4.3.3 Simhash Source – N-Gram

The n-grams are the only generator of the simhashes. The careful selection of appropriate n-gram is essential. Note that it is also possible to combine several of the n-gram sources (see Section 3.4.3), however due to performance reasons, we will restrict ourselves to a single source.

Note that n-gram = 4 is highly optimized, but this optimization was not used during the measurement process.

The Figure 4.5 shows the compression ratio relation to the size of the n-gram. The left plots on the left diagram are shifted so all the individual plots depict their compression ratio relatively to that of a 4-gram. From there, it is obvious that if we change the size of the n-gram in any direction, the overall compression ratio will be higher (thus worse configuration). The only exceptions are the `athens` dataset, which already has an extremely bad compression ratio, and the `prague` corpus, where the loss in compression ratio is negligible. The reason why low n-gram sizes fail is obvious – there is minimal multibyte redundancy information depicted in the simhash, plus the simhash is degenerate, as explained in Section 4.3.1 and 4.3.2.

The same figure shows that the increase in total time is very small with increasing

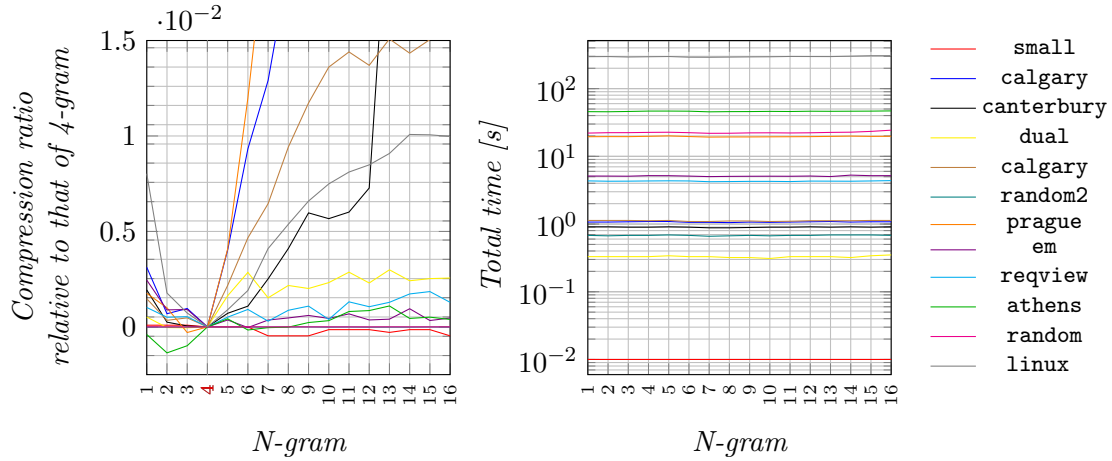


Figure 4.5: N-gram size (used as simhash generator) effect on final compression ratio and total time.

n-gram size. This is probably due to a high level of optimizations and full vectorization of the FNV hash used in simhash computation. The implementation of FNV module was transformed to horizontal computation over the string for better vectorization.

4.3.4 Simhash Width

The simhash width is another of the core parameters of ICBCS. It represents the size of every single simhash vector stored either in the system or in the archivedata.

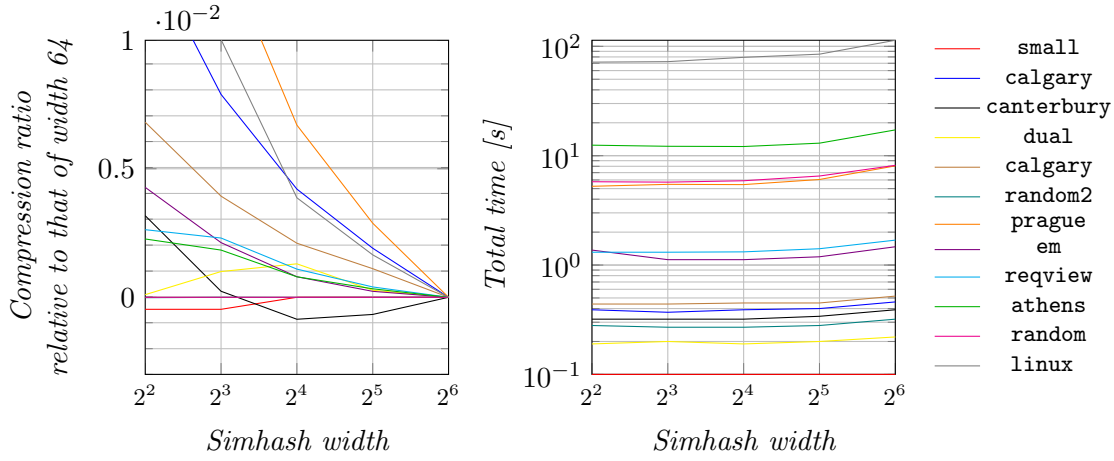


Figure 4.6: Simhash width effect on final compression ratio and total time.

The simhash width also has a very significant impact on the overall effectiveness of the compression system. Higher width negatively impacts the execution time – especially all distance calculations, since the simhash generator only merges 64b FNV hash. Width higher simhash width, the clustering is much more precise, resulting in deeper clustering tree and more granular final compression grouping. Also, since the simhashes are always stored in memory, this impacts the memory requirements directly. All these relations can be seen in Table 4.3 for the `prague` corpus.

Simhash width	*Compression ratio*	*# compression groups*	*Average depth*	*Total time*	*Memory*
4	0.6639	135	29.16	5.15	$1.64 \cdot 10^7$
8	0.6563	136	44.81	5.37	$1.7 \cdot 10^7$
16	0.6493	142	66.86	5.34	$1.79 \cdot 10^7$
32	0.6455	140	148.41	5.96	$2.02 \cdot 10^7$
64	0.6426	148	225.34	7.97	$2.68 \cdot 10^7$

Table 4.3: Simhash width effect on `prague` corpus.

Figure 4.6 shows clearly that the best compression ratio is achieved with the simhash width 64. This is not surprising because such causes the least effect on overall degeneration of the simhash vector system. The only exceptions are the `linux-kernels` dataset and `small` dataset (which is not important). Note that this figure depicts compression ratios relative to those of simhash width 64.

However, from the same figure, it is also apparent that the execution time increases drastically from width 32 to with 64. This is due to numerous distance calculations. The gain in compression ratio is too small, which is why the reference setup uses only simhashes of width 32.

Simhash width also has an effect on archivedata size. This comes directly from the archivedata design, for details, see Section 3.9.1 or Section 4.7 for experimental results based on simhash width on metadata and archivedata.

4.3.5 Performance Concerns

The FNV hashing algorithm has one disadvantage: it is not effectively utilized as a rolling hash. For string S rolling hash has to be able to produce hash of $S[i-1..k]$ from $S[i..k]$, however FNV doesn't have an operation that would remove a single byte from the hash and provide the hash of $S[i-1..k]$. The major advantage of the FNV is its speed and good spread over the entire width of the hash even with small number of bytes read.

Also note that complicates cryptographic or sophisticated hashes were not used. These are on the opposite performance spectrum of rolling hash functions. The security requirement for cryptographic hashes dictates time consuming computation. This is a contradictory requirement for this compression system.

4.3.6 Integral vs Floating Point Simhash Representation

As was shown in previous sections, the variance of simhashes decreases with increasing chunk size and increasing randomness in the data. This poses another possible problem.

Floating point operations provided very good precision as the variance of simhashes was getting smaller. However on the testing platform, floating point operations were 2 times slower than their integer equivalents.

Integer types are fast, but tend to degenerate in precision. This is problematic especially concerning very similar data. Also, nodes in the clustering tree also suffer from representing very similar data, especially in the lower levels, or in case the balancing is

disabled, entirely throughout the tree. Also, combining the hashes between subtrees of low sizes does not allow for precise weighted averaging. This can be solved by an artificial shift of the hashes into higher values, performing the weighted averaging and then shifting back to the original order.

The solution used in ICBCS relies on integers to represent a simhash. The simhash spread is set within the first 20 bits of the 32 bit integer. The other 12 bits can be used for the shift operations on weights of the combined vectors. Less than 12 bits are used if there is a possible overflow. On 64 bit platforms, the whole averaging operation is performed in 64 bit integers without any significant performance impact.

4.4 Rabin Chunkizer and Deduplication

The Rabin chunkizer represents the deduplication layer of ICBCS. It's effectiveness is highly dependent on the input data. However the superior speed and simplicity make it into and an ideal tool to remove the most explicit redundancies.

For better visualization, several histograms of unique and duplicate chunks are available. The histograms of real-world large scale example `linux-kernels` are depicted in Figure 4.7. Note that the size range 512 B – 32 KB was used. This is quite a reasonable range, as most of the chunks are of size 1 – 4 KB, which still results in acceptable execution times. (See Figure 4.10 for more details on the size and spread effects on execution time.)

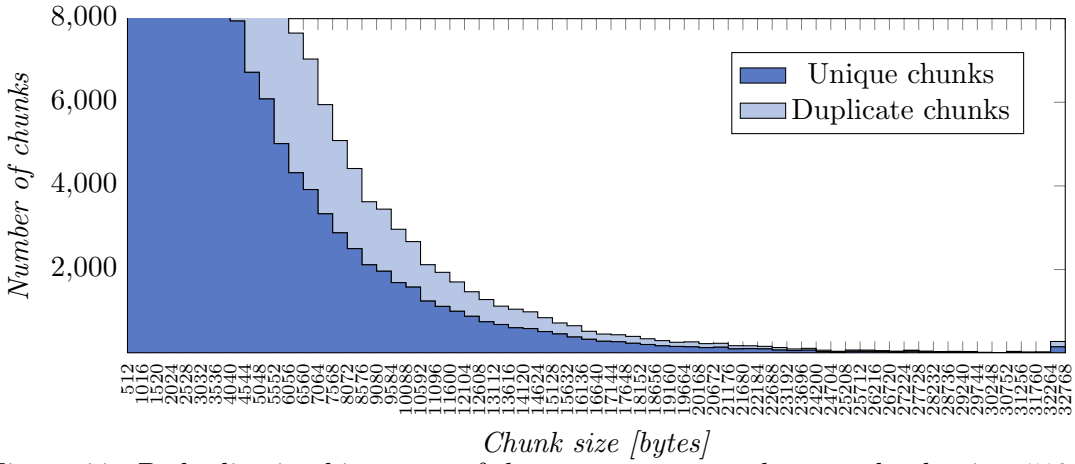


Figure 4.7: Deduplication histogram of the `linux-kernels` dataset, chunks sizes 512 B – 32 KB.

Histograms for aritifical dataset `dual` and `random` are depicted in Figure 4.8. In case of the `dual` dataset, the deduplication layer completely removes all but three duplicate chunks – chunks on the borderline of the two files, since the dataset was fed to ICBCS in a single tar file. In case the of `random` dataset, no duplicate chunks were found.

Examples of the distribution of chunk sizes can be seen from both Figures 4.7 and 4.8.

The overall deduplication ratio scales mainly with two parameters: average chunks size and chunk size spread.

With increasing average chunk size, less exact duplicates are found. This does not necessarily have to have a negative impact on the overall compression ratio. Deduplication, however is, faster than compression, so maximal deduplication is desirable.

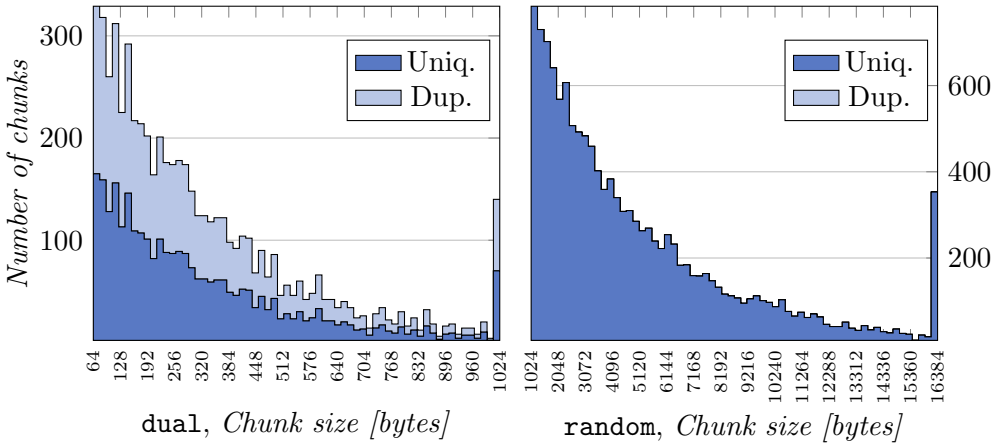


Figure 4.8: Deduplication histogram of the `dual` and `random` datasets. Chunk size range for the former one is 64 B – 1 kB, for the latter one 1 KB – 16 KB.

Figure 4.9 shows the decreasing deduplication ratio with increasing chunk size. The effect of chunk spread is rather insignificant, as long as there is at least any. No size spread means all the blocks are of the same size and the deduplication layer suffers from the same problems as standard fixed block deduplication. See Section 3.3 for details.

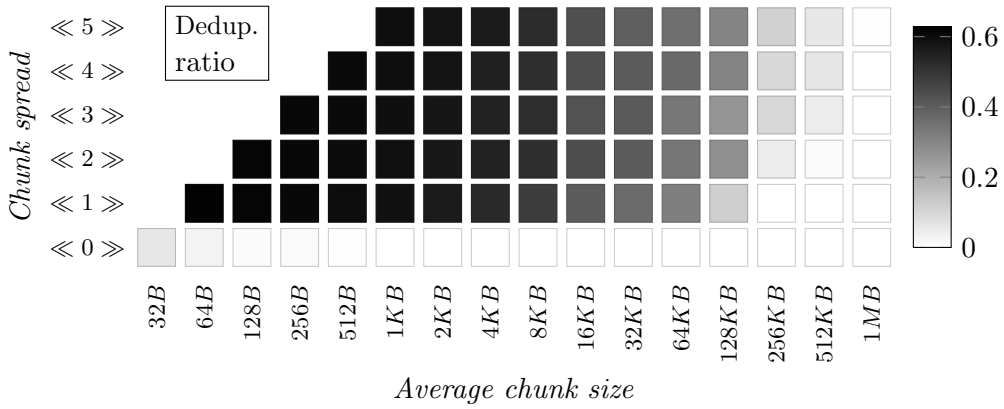


Figure 4.9: Deduplication ratio on `em` dataset. The x axis displays the average chunk size and the y axis displays the chunk spread parameter as given to the ICBCS (left and right shift on the average size by the given number of positions)

The chunk size also significantly impacts the overall execution time. Unfortunately the trend is opposite to that of a good deduplication ratio. Figure 4.10 depicts this relation. The optimal execution time values are achieved in the range of 2 KB – 8 KB average chunk size and chunk spread of 1 – 2. This is thereby used as a reference setup for ICBCS. Also note that the execution time increases with higher chunks sizes. This is due to the reduced effectiveness of explicit deduplication further described in the next Section 4.4.1.

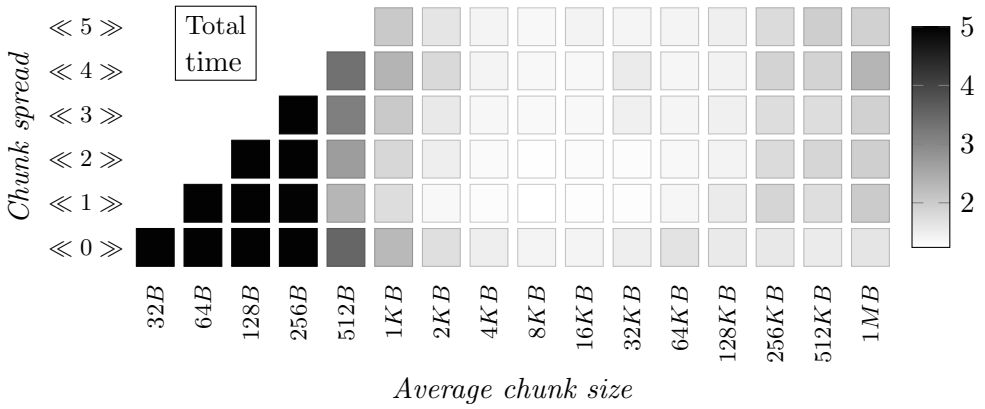


Figure 4.10: Total execution time on `em` dataset. The x axis displays average chunk size and the y axis displays chunk spread parameter as given to the ICBCS (left and right shift on the average size by the given number of positions). Note the increased time even for larger chunks – this is due to ineffective deduplication in those ranges.

4.4.1 Non-Explicit Deduplication

The term of *explicit* deduplication refers to deduplication performed by the deduplication layer (Section 3.3) and the term *implicit* deduplication refers to the fact that two matching chunks will be with assigned into the same compression group and compressed together resulting in almost-deduplication within the abilities of the compressor.

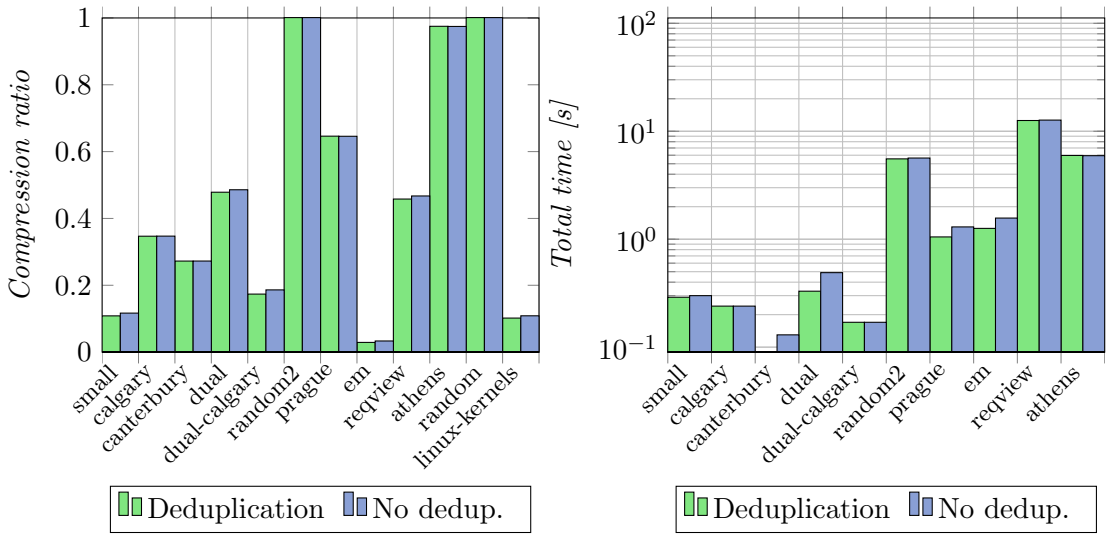


Figure 4.11: Explicite deduplication

Figure 4.11 shows the comparison of explicit and implicit deduplication on all the datasets. Implicit deduplication results in only a slightly better compression ratio. The absorption of the duplicate by the compressor is almost negligible compared to explicit deduplication. The major difference is the execution time. Chunks that would otherwise be removed during the early process of deduplication instead proceed further to the clustering process, which is costly compared to deduplication.

When using NCD, the implicit duplicates all have 0 distance among each other. This effectively paralyzes the clustering process, since it converges to an optimal clustering, although if some of the 0 distance pairs compress together better than others.

Another drawback of the implicit clustering is a possible cluster group overflow, resulting in another compression group with the same data.

4.5 Simhash vs. NCD

This section briefly compares the simhash distance based clustering with the NCD distance based based clustering. The NCD is forced to use the top-down clustering technique described in Section 3.6.1 for its represenative variants.

Full NCD clustering is considered the oracle for compression by clustering. It achieves better compression ratio than Simhash in almost any scenario. There is no better currently known distance than the NCD that describes the inter-compression efficiency of two strings. This comes at the price of non-existent distance to space transition for NCD and thus renders the need for excessive distance calculations (up to quadratic).

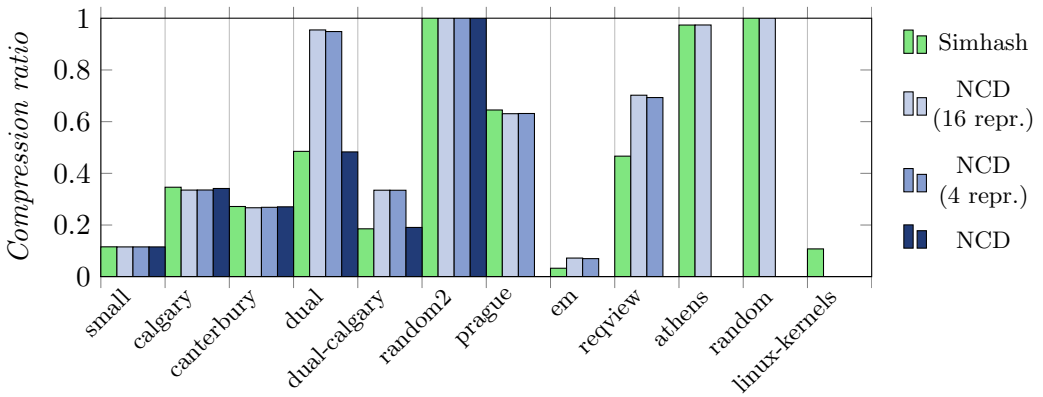


Figure 4.12: Compression ratio comparison among Simhash, NCD (4 representatives), NCD (16 representatives) and NCD (full) clusterings. No explicit deduplication. Some measurements with high total time were omitted. See Figure 4.13 for execution time.

Figures 4.12 and 4.13 show the 2 different clustering approaches – using simhash distance and NCD. The latter is displayed in 3 variants – using top-down clustering with 4 representatives or 16 representatives as a distance reference points, and the full NCD (top-down and bottom-up approach is irrelevant in this case in regards of the compression ratio).

The first Figure 4.12 clearly shows that full NCD is always the same or better than any other approach, which is an expected result. However, the representative alternatives of NCD tend to fail to compress very redundant data, especially those with many duplicates. The best clustering is achieved by min-wise pairing of similar clusters together. But because of its top down approach, representative NCD fails to recognize such pairs and without explicit deduplication falls terribly short compared to its competitors.

The execution time comparison is depicted in Figure 4.13. All of the NCD algorithms take much longer to cluster the dataset. Note that even for NCD clusterings with limited number of representatives, the time is almost 10 times higher. For complete NCD, the overall complexity is quadratic. NCD was not optimized as thoroughly as simhash,

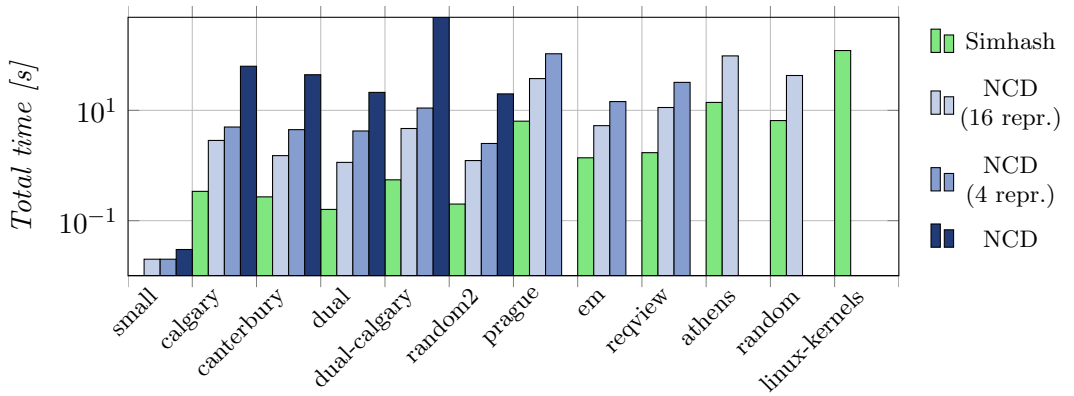


Figure 4.13: Total time comparison among Simhash, NCD (4 representatives), NCD (16 representatives) and NCD (full) clusterings. Some measurements with high total time were omitted. See Figure 4.12 for compression ratio.

	Simhash		*NCD (4 representatives)*	
`dataset`	*Comp. ratio*	*Total time*	*Comp. ratio*	*Total time*
small.tar	0.1157	0	0.1153	$1 \cdot 10^{-2}$
calgary.tar	0.3463	0.33	0.3352	2.84
canterbury.tar	0.2717	0.26	0.2668	1.5
dual.tar	0.485	0.15	0.9549	1.13
dual-calgary.tar	0.1854	0.54	0.3347	4.68
random2.dat	1.0003	0.19	1.0003	1.22
prague.tar	0.6451	6.36	0.6308	37.66
em.tar	0.0328	1.37	0.0722	5.27
reqview.tar	0.4665	1.7	0.7023	11.26
athens.tar	0.9738	13.92	0.9741	97.17
random.dat	1.0003	6.52	1.0003	42.87
linux-kernels.tar	0.1076	121.51	NaN	NaN

Table 4.4: Comparison of Simhash clustering and NCD clustering with 4 representatives. No deduplication.

however the theoretical time complexity would have been the sole denying factor for successful incremental and real-time environment.

A comparison of simhash and NCD with 4 representatives is summarized in Table 4.4. NCD with more representatives or with full scanning are much slower. The NCD with 4 representatives achieves only a slightly better compression ratio for datasets with very variable data, but as was previously mentioned, fails to cluster similar pairs in dual data – this is due to top-down clustering.

4.6 Clustering Quality

The overall quality of the clustering can be characterized with several properties. Some of these properties reflect directly into the final compression ratio, some of those may

affect the execution time, and some are profitable in the archiver scenario. Following is the list of variables we are concerned about:

Compression ratio is the obvious choice. Perfect clustering always has the minimal distance sets of chunks clustered together, resulting in an optimal compression ratio given a compression group size.

Total time is the time needed for data deduplication, clustering and compression. The time for clustering also affects the compression time, this is why these are not distinguished. Deduplication is turned off in these tests.

Clustering disbalance measures how many clusters do not satisfy the distance-between inequality, see Section 3.6.3. Higher rate results in gradual degeneration of the clustering, however only minor such relation was detected in any of the tests.

Average chunk depth affects both the speed of clustering and the final compression ratio, because the chunks are likely to be clustered into groups of more balanced sizes and distances. Note that this is somewhat mitigated by the compression group upmerging described in Section 3.7.2

 # compression groups directly affects the compression ratio. It also has an effect on archival capabilities.

All the tests in this section were run with disabled upmerging and small compression group sizes of 4-8 chunks. Note that this also successfully demonstrates how small groups are actually necessary for an efficient compression.

4.6.1 Balancing

The balancing refers to a technique of gradual improvement of the clustering during new chunk insertion. It is described in Section 3.6.3

The total time is affected by the depth of the clustering, since it has to go through the entire tree and update the nodes along the path to the root. However, it take much longer to balance the clustering tree properly than to add a little bit of computation during every chunk addition.

The clustering disbalance is increasing with more balancing operations, however the system does not degenerate at all. This is caused by the naive metric of the clustering disbalance, which only counts the number of disbalanced nodes, but not the scale of such disbalance. This means that the overall scaled (weighted) disbalance of the clustering is much lover, but the amount of disbalanced clusters is higher when balancing is used.

All these attributes together with number of compression groups for the `prague` dataset are summarized in Table 4.5.

The average depth of all chunks in the clustering is significantly decreased by clustering balancing. Even a single pass of of the balancing algorithm is sufficient for a large reduction in the average depth. This is displayed in Figure 4.14.

The most desired attribute – compression ratio, however, does not increase significantly with better balancing. It does never get worse for better and deeper balancing used (except for random data). This relation is shown in Figure 4.15.

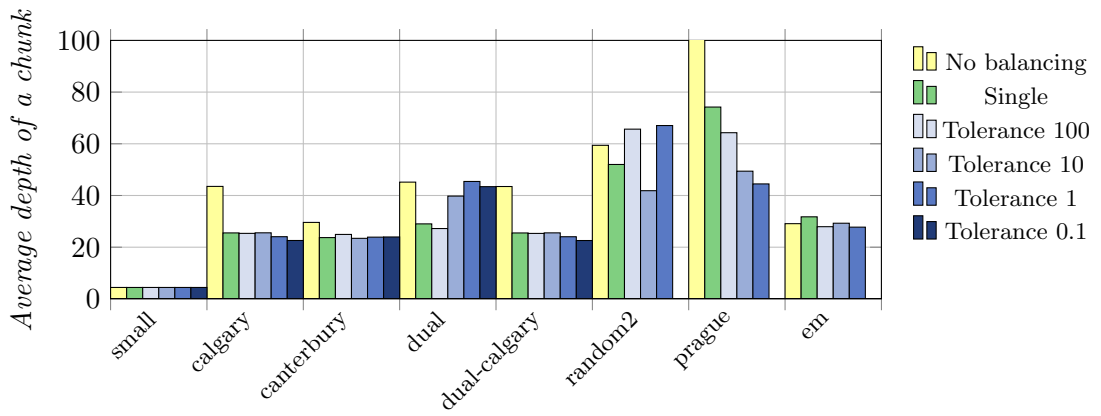


Figure 4.14: Effect of clustering balancing on the average depth of a chunk in the clustering. No upmerging, compression group span 4–8.

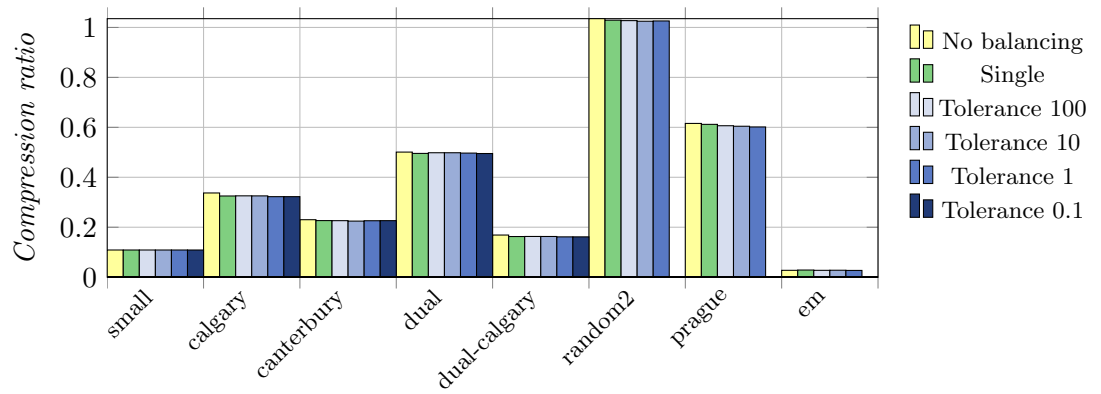


Figure 4.15: Effect of clustering balancing on the overall compression ratio. No upmerging, compression group span 4–8.

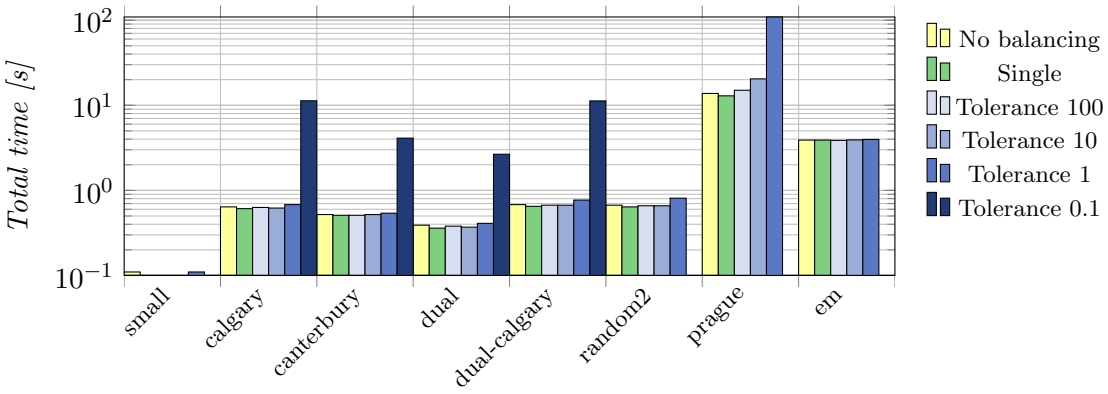


Figure 4.16: Effect of clustering balancing on the total execution time. No upmerging, compression group span 4–8.

Balancing	*C.Ratio*	*#C.Groups*	*Disbalance*	*Avg.Depth*	*Total Time*
No balancing	0.6157	3 864	2 750	148.41	13.67
Single	0.6119	2 594	2 453	74.21	12.81
Tolerance 100	0.6064	2 186	2 667	64.29	14.92
Tolerance 10	0.6044	1 779	2 906	49.41	20.33
Tolerance 1	0.6015	1 687	3 124	44.48	108.9

Table 4.5: Summary of balancing test on the `prague` dataset. No upmerging, compression group span 4–8.

Figure 4.16 shows the time consumption of different balancing levels. There is an overall increasing tendency for better balancing levels (lower tolerance of cluster disbalance). There are, however several exceptions. The time of clustering for the single balancing version is a little faster the one for single balancing. This is caused by a simple fact that even a single balancing decreases the average depth of the clustering tree significantly. Subsequent additions of new chunks then have to go shorter way to the top of the tree and perform less distance comparisons and balancing operations. The same effect can be seen in a tiny scale between balancing with tolerance of 100 vs. 10.

The oscillating compression ratio between no balancing, single balancing and balancing with a tolerance of 100 are causes by the fixed tolerance value. Sometimes the balancing does not trigger at all.

The overall conclusion in the balancing problem is to use single balancing that results in both better compression rate and execution times for most of the datasets.

4.6.2 Deep Balancing and Representatives

Another extended distance measure was tested – the deep distance and deep representative distance for simhashes. For description of these, please see Section 3.6.6.

Since extensive balancing described in the previous Section 4.6.1 had only a limited success, it is unlikely its extensions will perform significantly better.

The standard shallow simhash was tested against the deep simhash. The deep simhash can be parameterized by the maximal depth in which it will search for child clusters. This of course degenerates to full SLINK in case of an inifite depth. The improvement to compression ratio with deep simhash and single balancing was almost negligible; it was less than 1% in every case. However the computation time was significantly higher (up to two times) for all the measured datasets. See Figure 4.17 for details.

Another variant of simhash distance called the deep-repr was also tested, but with even lesser gains in the compression ratio and higher costs in the computational time.

4.7 Metadata and Archivedata Overhead

This section attempts to show the relationship between input size to size of produced metadata and archivedata. For more information about those, please see Sections 3.8

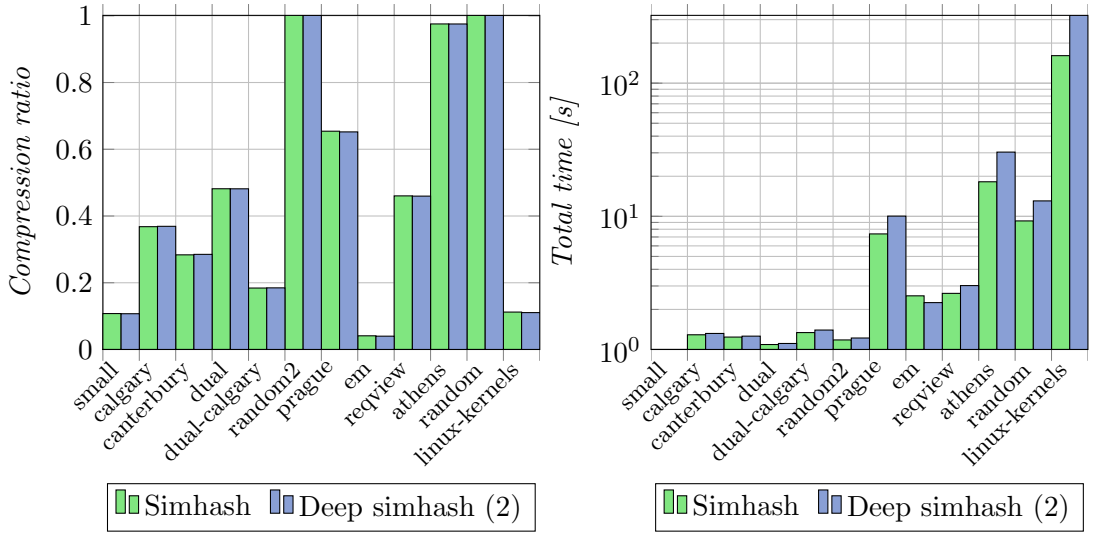


Figure 4.17: Deep simhash vs. shallow simhash distance and balancing. Deep simhash goes up to the depth of 2, meaning it compares at most 4 to 4 child simhashes. Single balancing, no upmerging, compression group span 4–8.

and 3.9.

In short, metadata contains all the information needed for successfull retrieval or decompression of a single file. If the ICBCS is used as a compressor, metadata is all that needs to be saved with the compressed data. The size of metada is strictly linear with the size of the input and number of chunks. All the possible optimizations of the metadata format are out of the scope of this thesis. A higher deduplication ratio may only have better constant in the memory complexity, based on the format used.

Archivedata saves the entire model including all chunk simhashes and the clustering. A basic implementation of the archivedata was used for the measurements to get a general idea about the size of the archivedata.

Metadata format used for these measurements was a simple ordered list of offsets (position in the original data) of chunks and duplicates, compressed with the same compressor as the data itself. In case of the reference setup, the amount of metadata accounts for less than 0.1% of the original size. For detailed comparison of several simhash widths and average chunk sizes, see Table 4.6.

Archivedata is linear with total size of unique chunks × simhash width. Explicit deduplication thus has an effect on archivedata size. The reference solution uses simple linearization of the clustering with simhashes attached in an ordered list. The representation of archivedata should be subject to further optimizations.

A detailed comparison of metadata and archivedata sizes for all the datasets can be seen in Figure 4.18.

4.8 Compression Parameters

A compressor is specific and one of the most characterizing parameters of ICBCS. It is used to compress all the compression groups of chunks. The main concerns of compression parametrization are the algorithms used (Section 4.8.1), compression level of

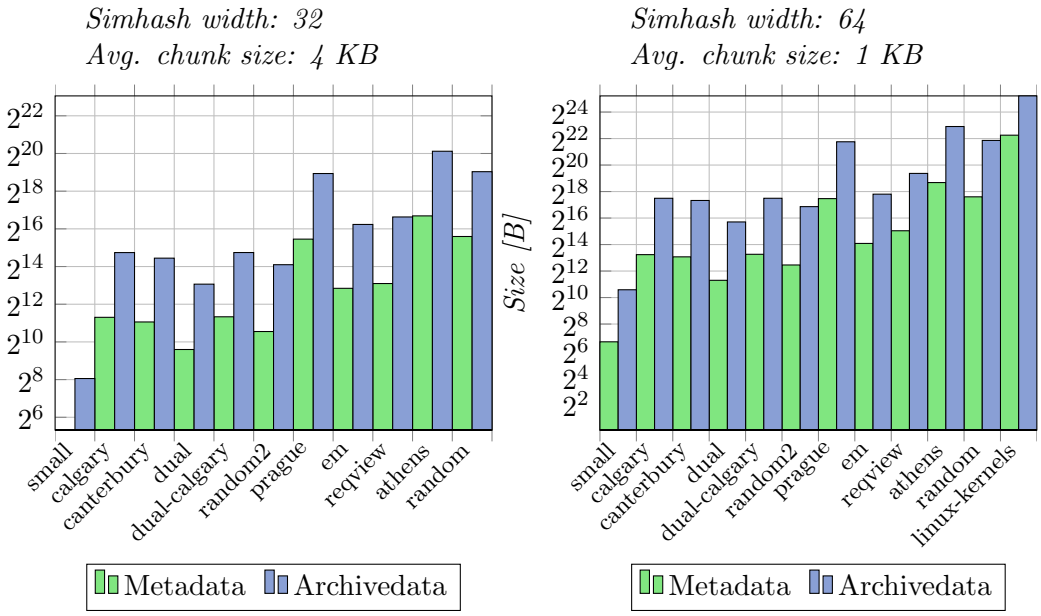


Figure 4.18: Metadata and Archivedata

Simhash width	*Avg. chunk size*	*Metadata [B]*	*Archivedata [B]*	*Comp. ratio w\o any data*	*Comp. ratio w\ metadata*	*Comp. ratio w\ metarchive data*
4	12	50 852	$3 \cdot 10^5$	0.6639	0.6647	0.6675
8	12	50 860	$3 \cdot 10^5$	0.6563	0.6571	0.6607
16	12	50 908	$4 \cdot 10^5$	0.6493	0.6501	0.6553
32	8	$8 \cdot 10^5$	$9 \cdot 10^6$	0.6737	0.6851	0.8098
32	10	$2 \cdot 10^5$	$3 \cdot 10^6$	0.6586	0.6617	0.6963
32	12	50 892	$6 \cdot 10^5$	0.6455	0.6462	0.6548
32	12	50 892	$6 \cdot 10^5$	0.6455	0.6462	0.6548
32	14	11 428	$1 \cdot 10^5$	0.6326	0.6328	0.6347
32	16	2 972	37 588	0.6287	0.6287	0.6292
64	12	50 956	$1 \cdot 10^6$	0.6426	0.6434	0.6586

Table 4.6: Impact of simhash width and chunk size on metadata, archivedata and the respective compression ratios on the `prague` corpus.

the corresponding level, where applicable (Section 4.8.1.1) and compression group size (Section 4.8.2). There is a technique described in Section 3.7.2 called compression group upmerging, that further optimizes the grouping into compression groups (Section 3.7.2).

4.8.1 Compression Algorithms Comparison

Two algorithms implemented in ICBCS are the bzip2 and deflate. Deflate made it into the reference setup due to its much faster speed and small loss in compression ratio.

ExCom [45] integration is one of the possible solutions to significantly increase the amount of available compressors.

Most of the tests especially in the following section split the measurement into bzip2 and deflate categories. See Figures 4.19, 4.23, 4.24 and Table 4.7 for respective details.

4.8.1.1 Compression Levels (Comparison to Solid and Non-Solid Compression)

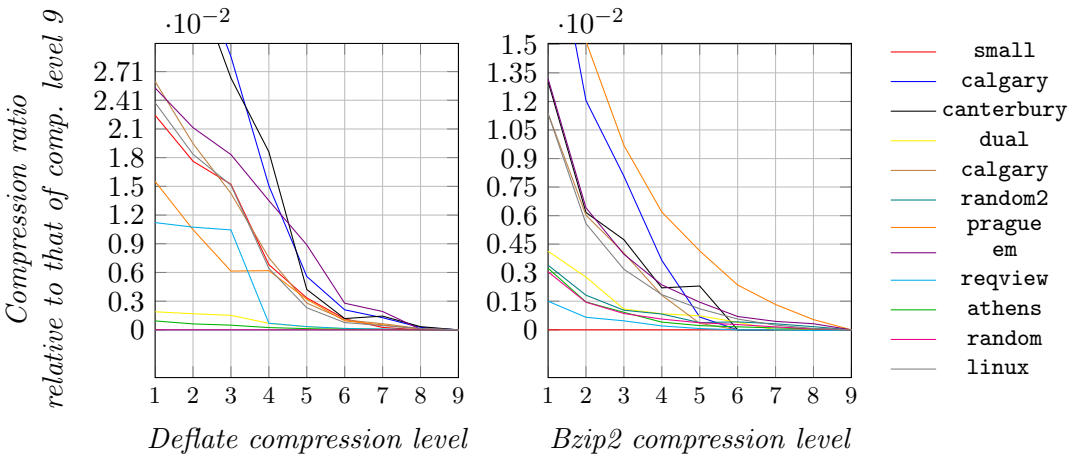


Figure 4.19: Deflate and Bzip2 compression level effect on compression ratio.

The compression level parameter is passed directly to the underlying compressor. The compression ratio gain above compression level 5 for bzip2 and 6 for deflate increases by only a small amount. The problem of compression level plays only a significant role for smaller compression group sizes, where it may cause the added advantage of a higher compression level of the compressor to be wasted due to small data amount in the compression group. Figure 4.19 shows the relative compression levels compared to compression level 9.

4.8.2 Compression Groups

The compression group is a subset of chunks that are compressed together. The similarity among all the chunks in the compression group has to be as low as possible. What is often overlooked is the fact that ordering the the compression group plays a very significant role. It is necessary to keep very similar chunks as close to each other in the group as possible. This is especially important for deflate compressor.

4.8.2.1 Group Sizes

The size of a compression group plays one of the most significant roles in the final compression output.

On the other hand, the group size also plays a huge role in the archival properties. Larger groups may cause a significant need for decompression and subsequent recompression of too large data (the entire group). In case of variable data, this may affect number of chunks proportional to the number of chunks in newly added data, but in case of uniform data, single large compression group may be created for those – this is however not important, since the overhead here is the number of unnecessarily uncompressed chunks, and that can only be mitigated by large number of small compression groups.

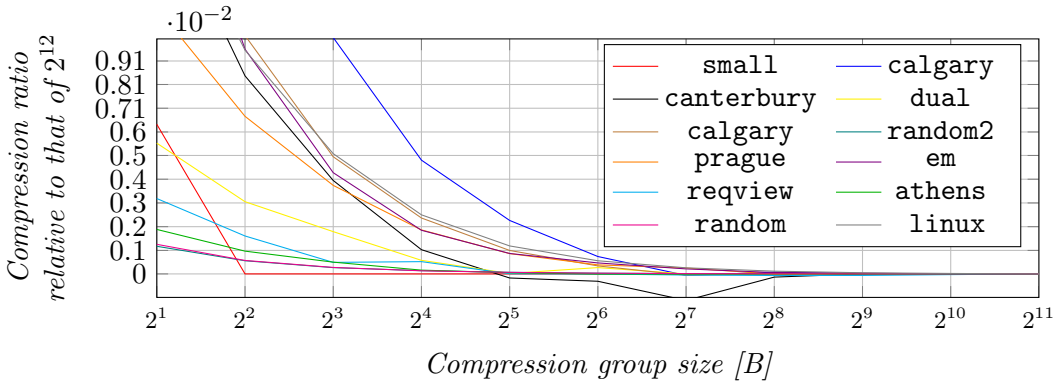


Figure 4.20: Compression group size effect on final compression ratio. No umperging.

Figure 4.20 shows the effect of compression group size to compression ratio. Starting with the size of 64 (this actually refers to the range 64 – 128 chunks) per compression group, the compression ratio gain starts to fade. That is why 64 was chosen as reference setup for ICBCS.

Proper balancing also has an effect on reducing the number of compression groups. See Section 4.6.1 for details.

4.8.2.2 Compression Groups Upmerging

The compression group upmerging (see Section 3.7.2) artificially adds chunks or small compression groups to the nearest larger compression groups until the capacity is filled. This results in higher compression ratios for no significant additional computational time expense, however the archival properties are reduced with this technique. See Figure 4.21 for details.

4.9 Memory Requirements

The preliminary testing implementation of ICBCS was only optimized for time in many aspects. Memory requirements were not optimized in any way. A large, constant portion of data was held in memory during the entire run. Also all the simhash data were redundantly saved in both the clustering and KD trees.

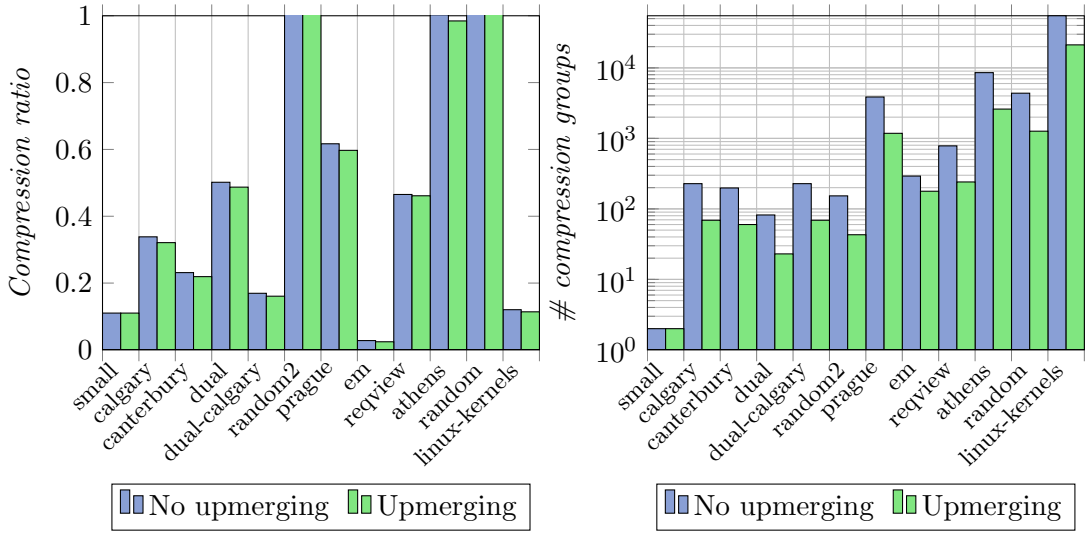


Figure 4.21: Upmerging effect on compression ratio and number of compression group. Bzip2 compressor, size of compression group: 8.

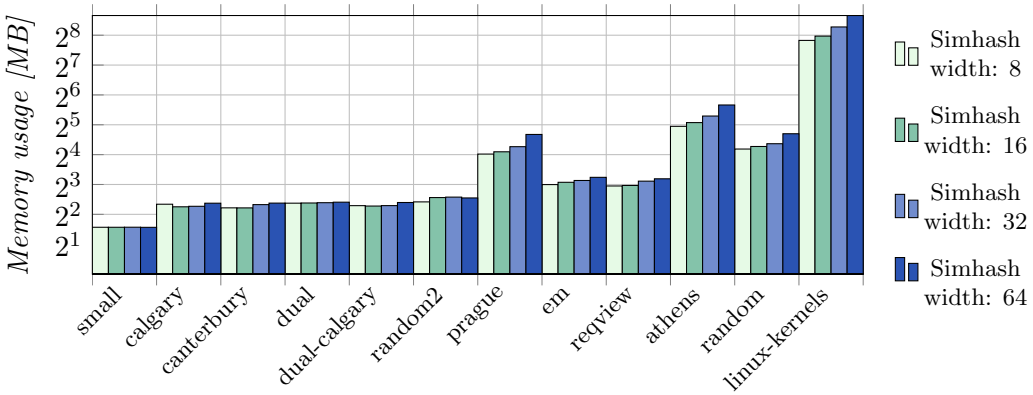


Figure 4.22: Memory usage of ICBCS.

The sole purpose of this section is to show the impact on memory consumption for different simhash widths and different input sizes. See Figure 4.22 for details. The memory consumption indeed is linear with the size of the input with a small amount of constant memory. The constant of the linear factor could be significantly minimized with memory usage optimizations.

4.10 Performance Summary

The overall performance of ICBCS is compared to standard Bzip2 and Deflate compressors, since ICBCS can be seen as an improvement of the previously mentioned compressors with additional archival properties.

ICBCS was not compared to deduplication systems, because the idea of ICBCS is not just to deduplicate, but the ICBCS merely uses deduplication to make the process of redundancy removal faster and more effective (Section 3.3. Other deduplication systems

only use compression on single blocks or on sets of very similar blocks (standard binary simhash, see survey in Section 2.6.6).

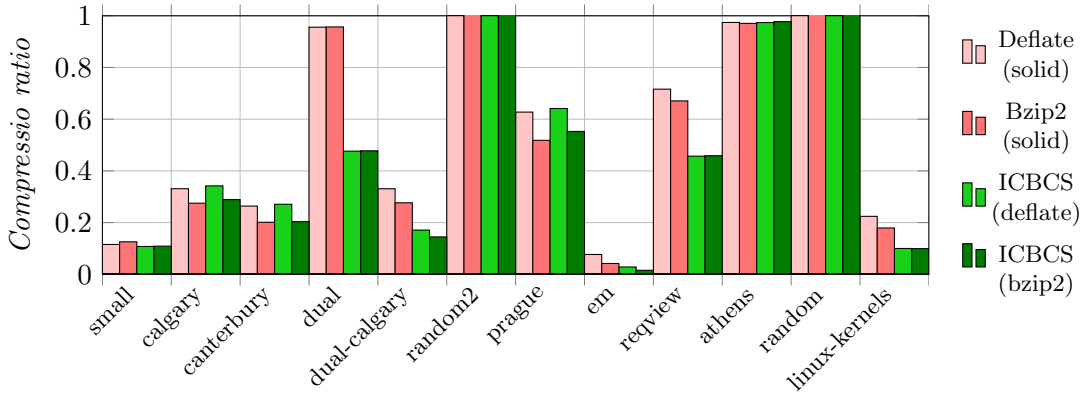


Figure 4.23: Comparison of compression ratio among ICBCS, deflate and bzip2.

The compression ratio of ICBCS is much better in every case of duplicate or redundant data. The datasets `dual`, `dual-calgary`, `em`, `reqview` and `linux-kernels` all exhibit much better compression than if Bzip2 or Delfate were used for a solid compression.

On the other hand, variable datasets as `calgary`, `canterbury` and `prague` all exhibit only slightly worse compression ratio. This is due to the split of the compression into multiple compression groups. This of course allows the ICBCS to work as an archival system. It is actually interesting to see that even by splitting the data into tens or hundreds of separate compression groups, the compression ratio can be almost entirely preserved.

Artificial datasets as `random2`, `random` and an image heavy dataset `athens` all show no to minimal improvements.

The `small` dataset successfully demonstrates that smart reordering of the contents can also achieve better compression ratios.

The compression ratio over all these datasets is depicted in Figure 4.23. More attributes such as compression ratio with metadata and archivedata, number of compression groups and total time are summarized in Table 4.7 for all the datasets and the reference setup.

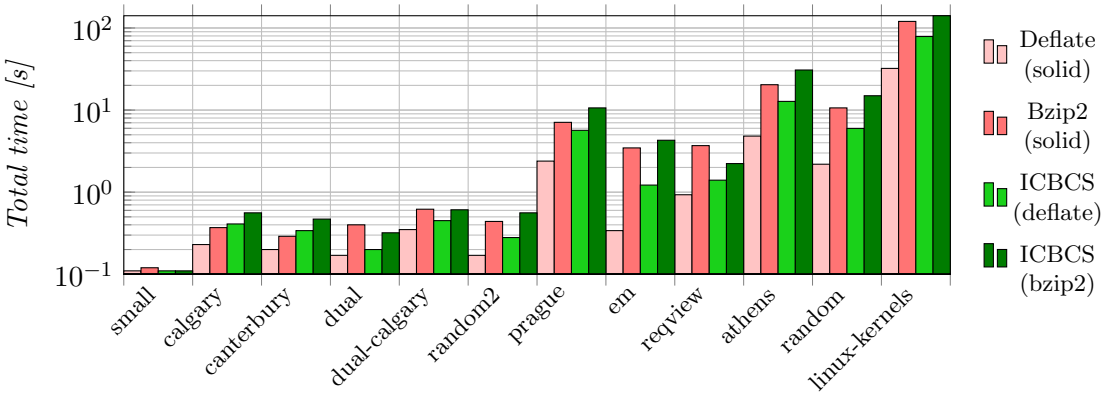


Figure 4.24: Comparison of total time among ICBCS, deflate and bzip2.

Dataset	*Compression ratio (metadata)*	*Compression ratio (metarchive)*	*# compression groups*	*Total time*
small	0.1084	0.1132	1.00	0.00
calgary	0.3428	0.3512	7.00	0.30
canterbury	0.2717	0.2796	7.00	0.23
dual	0.4768	0.4813	3.00	0.09
dual-calgary	0.1714	0.1756	7.00	0.34
random2	1.0011	1.0094	6.00	0.17
prague	0.6422	0.6507	143.00	5.56
em	0.0286	0.0322	23.00	1.11
reqview	0.4575	0.4629	28.00	1.29
athens	0.9746	0.9829	318.00	12.65
random	1.0011	1.0090	151.00	5.87
linux-kernels	0.1009	0.1073	2 535.00	78.65

Table 4.7: Summary of ICBCS performance on all datasets.

The overall CPU time of ICBCS is always higher by a multiplicative constant of 3–4 $\times$ log of the input size. In every case, the deduplication and clustering processes are the only parts $\in O(n \cdot \log(n))$, the chunking and compression process is $\in O(n)$ with the size of the input.

Note that the comparison was done against solid compressions. A significant part of current archivers and systems using compression relies on so called non-solid compression, where the files are compressed separately and then put into an archive. Our experiments show that for sufficiently large individual files, non-solid compression achieves the same compression characteristics as solid compression. Conversely, in cases where the file contents vary a lot, non-solid compression achieves better characteristics, due to separation of compression contexts that are likely to collide in solid compression. For small individual files or similar files, solid compression achieves better characteristics due to effective redundancy removal among separate files.

For the purpose of demonstrating ICBCS qualities, non-solid compression was deemed unnecessary and redundant.

Chapter 5

Conclusion

In the thesis, we went through a process of designing a novel system using a combination of techniques that have never been used together in our problem area. The overall abundance of resources especially on unsupervised learning algorithms, e.g. clustering, would have forked the design into too many branches, of which we could not theoretically evaluate efficiency for our problem. Chosen techniques are thus are not in the cutting edge in their respective areas. Instead, well understandable and scalable techniques were used, giving out system a proof-of-concept status.

Extensive research on state of the art of current deduplication and compression systems was made in order to understand the core ideas these systems are built on. The major technique present in all quality deduplication systems is the Rabin fingerprinting, where files are split into chunks of variable size based on the content, instead of fixed size blocks. Other important techniques were related to near identical or very similar chunk recognition. These techniques include the similarity hash, minhash and locality sensitive hashing. The similarity hash was extended in this work to increase the scope of similarity recognition.

Further in the work a deduplication, compression and archival system called ICBCS is introduced. The system's design tries to overcome the shortcomings of current deduplication and compression systems by extending the redundancy removal scope to thoroughly selected clusters – sets of chunks of input files. The system assumes roles of both a compressor and a file archiver. Modular architecture allows the systems to be further adapted to various scenarios and extended easily, as well as parameterized with many different compressors, distance measures, clustering algorithms, etc.

Through extensive parameterization a reference setup of ICBCS was established and subsequently tested. This reference setup is based solely on a proof-of-concept implementation. With tens of proposed improvements that didn't make it into the implementation, it is safe to say the system can be improved in multiple areas.

Last, it was demonstrated through experiments that ICBCS has little problems overcoming the drawbacks of standard compressors and deduplication and compression systems, especially of wide-range redundancy removal, however at a slight increase of computation resources. For some extreme parameter values, this increase of resources shifts from slight to significant.

The overall time overhead induced by the system was much smaller than that of high precision compressors, while providing much better compression ratios. Typical examples of this is the 1.5 GB Linux kernels dataset, where there are three Linux kernel sources

of different version, yielding moderately redundant data. On this (and similar dataset), using our system ICBCS with a fast and weak compressor such as Deflate (LZ77), will result in both more than twice faster compression and almost half the compression ratio than a simply applied strong compressor Bzip2 (BWT). On very variable datasets with minimal interfile redundancy, the compression is split into artificial groups, only slightly increasing the overall compression ratio.

Apart from these positive properties, where ICBCS was used as a straightforward compressor, the system can also be used as an archival system, where files are added, edited or deleted from the archive. ICBCS' design lays down a basis for the extension to an archival system, but the proof-of-concept implementation doesn't include such functionality. Combining properties of both a compressor and an archiver, the ICBCS creates an original deduplication, compression and archival system.

The system can be deployed in real world scenarios alongside other compression systems, either simple compressors, archival or deduplication systems. For this, more effort has to be invested into bringing the system up to the tier of its competitors.

5.1 Future Work

Most of the effort for future improvements of ICBCS should be invested into improving the systems functionality, completing and fine-tuning the current ideas, etc. reaching the tier of other similar systems available nowadays.

One of such extensions lies in file type recognition. Such recognition could either be explicit or implicit implied by the clustering and predetermined prototypes of image data, text data, audio, video, etc. Different file types would then be handled differently in terms of clustering, grouping and compression.

The functionality could be further extended to more space in the deduplication and compression space described in Section 2.1, e.g. as an online service, adding more compressors, or using a hardware acceleration.

The extension by additional compressors could be easily achieved with integration of the ExCom [45].

Effective parallelization of ICBCS is another broad topic. Multiple chunks could be processed at the same time, but it risks missing the similarities between files in the current batch.

Bibliography

[1] Arvind Agarwal, Jeff M Phillips, Hal Daumé III, and Suresh Venkatasubramanian. Incremental multi-dimensional scaling.

[2] Alexandr Andoni and Piotr Indyk. Near-optimal hashing algorithms for approximate nearest neighbor in high dimensions. *Commun. ACM*, 51(1):117–122, January 2008.

[3] Mihael Ankerst, Markus M Breunig, Hans-Peter Kriegel, and Jörg Sander. Optics: Ordering points to identify the clustering structure. In *ACM SIGMOD Record*, volume 28, pages 49–60. ACM, 1999.

[4] Ross Arnold and Tim Bell. A corpus for the evaluation of lossless compression algorithms. In *Data Compression Conference, 1997. DCC'97. Proceedings*, pages 201–210. IEEE, 1997.

[5] Lior Aronovich, Ron Asher, Eitan Bachmat, Haim Bitner, Michael Hirsch, and Shmuel T. Klein. The design of a similarity based deduplication system. In *Proceedings of SYSTOR 2009: The Israeli Experimental Systems Conference*, SYSTOR '09, pages 6:1–6:14, New York, NY, USA, 2009. ACM.

[6] Nikhil Bansal, Avrim Blum, and Shuchi Chawla. Correlation clustering. *Machine Learning*, 56(1-3):89–113, 2004.

[7] Tim Bell. Canterbury corpus. http://corpus.canterbury.ac.nz/descriptions/#calgary, 1990.

[8] D. Bhagwat, K. Eshghi, D. D E Long, and M. Lillibridge. Extreme binning: Scalable, parallel deduplication for chunk-based file backup. In *Modeling, Analysis Simulation of Computer and Telecommunication Systems, 2009. MASCOTS '09. IEEE International Symposium on*, pages 1–9, Sept 2009.

[9] Jeramiah Bowling. Opendedup: open-source deduplication put to the test. *Linux Journal*, 2013(228):2, 2013.

[10] AndreiZ. Broder. Identifying and filtering near-duplicate documents. In *Combinatorial Pattern Matching*, volume 1848 of *Lecture Notes in Computer Science*, pages 1–10. Springer Berlin Heidelberg, 2000.

[11] A.Z. Broder. On the resemblance and containment of documents. In *Compression and Complexity of Sequences 1997. Proceedings*, pages 21–29, Jun 1997.

[12] Leonid A. Broukhis. The calgary corpus compression & sha-1 crack challenge. http://mailcom.com/challenge/, May 1996.

[13] Roelof K Brouwer. Extending the rand, adjusted rand and jaccard indices to fuzzy partitions. *Journal of Intelligent Information Systems*, 32(3):213–235, 2009.

[14] Michael Burrows and David J Wheeler. A block-sorting lossless data compression algorithm. 1994.

[15] Ludwig M. Busse, Peter Orbanz, and Joachim M. Buhmann. Cluster analysis of heterogeneous rank data. In *Proceedings of the 24th International Conference on Machine Learning*, ICML '07, pages 113–120, New York, NY, USA, 2007. ACM.

[16] Adam Cannane and Hugh E. Williams. General-purpose compression for efficient retrieval. *J. Am. Soc. Inf. Sci. Technol.*, 52(5):430–437, March 2001.

[17] Adam Cannane and Hugh E. Williams. A general-purpose compression scheme for large collections. *ACM Trans. Inf. Syst.*, 20(3):329–355, July 2002.

[18] William B Cavnar, John M Trenkle, et al. N-gram-based text categorization. *Ann Arbor MI*, 48113(2):161–175, 1994.

[19] Daniele Cerra and Mihai Datcu. A model conditioned data compression based similarity measure. In *Data Compression Conference, 2008. DCC 2008*, pages 509–509. IEEE, 2008.

[20] Daniele Cerra and Mihai Datcu. A fast compression-based similarity measure with applications to content-based image retrieval. *Journal of Visual Communication and Image Representation*, 23(2):293–302, 2012.

[21] Moses Charikar, Chandra Chekuri, Tomás Feder, and Rajeev Motwani. Incremental clustering and dynamic information retrieval. In *Proceedings of the twenty-ninth annual ACM symposium on Theory of computing*, pages 626–635. ACM, 1997.

[22] Moses S Charikar. Similarity estimation techniques from rounding algorithms. In *Proceedings of the thiry-fourth annual ACM symposium on Theory of computing*, pages 380–388. ACM, 2002.

[23] Rudi Cilibrasi and Paul MB Vitányi. Clustering by compression. *Information Theory, IEEE Transactions on*, 51(4):1523–1545, 2005.

[24] John G Cleary and Ian Witten. Data compression using adaptive coding and partial string matching. *Communications, IEEE Transactions on*, 32(4):396–402, 1984.

[25] Jonathan D. Cohen. Recursive hashing functions for n-grams. *ACM Trans. Inf. Syst.*, 15(3):291–320, July 1997.

[26] Graham Cormode, Mike Paterson, Süleyman Cenk Sahinalp, and Uzi Vishkin. Communication complexity of document exchange. In *Proceedings of the eleventh annual ACM-SIAM symposium on Discrete algorithms*, pages 197–206. Society for Industrial and Applied Mathematics, 2000.

[27] Trevor F Cox and Michael AA Cox. *Multidimensional scaling*. CRC Press, 2000.

[28] Owen Kaser Daniel Lemire. Rolling hash c++ library. http://code.google.com/p/ngramhashing/, Oct 2010.

[29] Biplob Debnath, Sudipta Sengupta, and Jin Li. Chunkstash: Speeding up inline storage deduplication using flash memory. In *Proceedings of the 2010 USENIX Conference on USENIX Annual Technical Conference*, USENIXATC'10, pages 16–16, Berkeley, CA, USA, 2010. USENIX Association.

[30] Inderjit S. Dhillon and Dharmendra S. Modha. A data-clustering algorithm on distributed memory multiprocessors. In *Revised Papers from Large-Scale Parallel Data Mining, Workshop on Large-Scale Parallel KDD Systems, SIGKDD*, pages 245–260, London, UK, UK, 2000. Springer-Verlag.

[31] Martin Ester, Hans-Peter Kriegel, Jörg Sander, and Xiaowei Xu. A density-based algorithm for discovering clusters in large spatial databases with noise. In *KDD*, volume 96, pages 226–231, 1996.

[32] Douglas H Fisher. Knowledge acquisition via incremental conceptual clustering. *Machine learning*, 2(2):139–172, 1987.

[33] Brendan J Frey and Delbert Dueck. Clustering by passing messages between data points. *science*, 315(5814):972–976, 2007.

[34] Mohamed Medhat Gaber, Arkady Zaslavsky, and Shonali Krishnaswamy. Mining data streams: A review. *SIGMOD Rec.*, 34(2):18–26, June 2005.

[35] Aristides Gionis, Piotr Indyk, Rajeev Motwani, et al. Similarity search in high dimensions via hashing. In *VLDB*, volume 99, pages 518–529, 1999.

[36] Google. Google ngram viewer. http://books.google.com/ngrams/datasets, 2012.

[37] Michael J Greenacre. *Theory and applications of correspondence analysis*. 1984.

[38] Sudipto Guha, Nina Mishra, Rajeev Motwani, and Liadan O'Callaghan. Clustering data streams. In *Foundations of computer science, 2000. proceedings. 41st annual symposium on*, pages 359–366. IEEE, 2000.

[39] Sudipto Guha, Rajeev Rastogi, and Kyuseok Shim. Cure: An efficient clustering algorithm for large databases. *SIGMOD Rec.*, 27(2):73–84, June 1998.

[40] Jiawei Han. *Data Mining: Concepts and Techniques*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2005.

[41] David J. Hand, Padhraic Smyth, and Heikki Mannila. *Principles of Data Mining*. MIT Press, Cambridge, MA, USA, 2001.

[42] Sariel Har-Peled and Soham Mazumdar. On coresets for k-means and k-median clustering. In *Proceedings of the Thirty-sixth Annual ACM Symposium on Theory of Computing*, STOC '04, pages 291–300, New York, NY, USA, 2004. ACM.

[43] Trevor Hastie, Robert Tibshirani, and Jerome Friedman. *The Elements of Statistical Learning*. Springer Series in Statistics. Springer New York Inc., New York, NY, USA, 2001.

[44] Monika Henzinger. Finding near-duplicate web pages: a large-scale evaluation of algorithms. In *Proceedings of the 29th annual international ACM SIGIR conference on Research and development in information retrieval*, pages 284–291. ACM, 2006.

[45] Jan Holub, Jakub Řeznícek, and Filip Šimek. Lossless data compression testbed: Excom and prague corpus. In *Data Compression Conference (DCC), 2011*, pages 457–457. IEEE, 2011.

[46] David A Huffman et al. A method for the construction of minimum redundancy codes. *Proceedings of the IRE*, 40(9):1098–1101, 1952.

[47] Piotr Indyk and Rajeev Motwani. Approximate nearest neighbors: towards removing the curse of dimensionality. In *Proceedings of the thirtieth annual ACM symposium on Theory of computing*, pages 604–613. ACM, 1998.

[48] Anil K. Jain. Data clustering: 50 years beyond k-means. *Pattern Recogn. Lett.*, 31(8):651–666, June 2010.

[49] Matt Mahoney Jim Bowery and Marcus Hutter. Hutter challenge. http://prize.hutter1.net/, March 2006.

[50] Ian Jolliffe. *Principal component analysis*. Wiley Online Library, 2005.

[51] Hisashi Koga, Tetsuo Ishibashi, and Toshinori Watanabe. Fast agglomerative hierarchical clustering algorithm using locality-sensitive hashing. *Knowledge and Information Systems*, 12(1):25–53, 2007.

[52] Con Kolivas. e3compr - ext3 compression. http://e3compr.sourceforge.net/, March 2008.

[53] Con Kolivas. Long range zip algorithm. http://ck.kolivas.org/apps/lrzip/, March 2011.

[54] Hans-Peter Kriegel, Peer Kröger, and Arthur Zimek. Clustering high-dimensional data: A survey on subspace clustering, pattern-based clustering, and correlation clustering. *ACM Transactions on Knowledge Discovery from Data (TKDD)*, 3(1):1, 2009.

[55] Purushottam Kulkarni, Fred Douglis, Jason LaVoie, and John M. Tracey. Redundancy elimination within large collections of files. In *Proceedings of the Annual Conference on USENIX Annual Technical Conference*, ATEC '04, pages 5–5, Berkeley, CA, USA, 2004. USENIX Association.

[56] Tilman Lange, Martin H. C. Law, Anil K. Jain, and Joachim M. Buhmann. Learning with constrained and unlabelled data. In *Proceedings of the 2005 IEEE Computer Society Conference on Computer Vision and Pattern Recognition - Volume 01*, CVPR '05, pages 731–738, Washington, DC, USA, 2005. IEEE Computer Society.

[57] Debra A. Lelewer and Daniel S. Hirschberg. Data compression. *ACM Comput. Surv.*, 19(3):261–296, September 1987.

[58] Ming Li, Xin Chen, Xin Li, Bin Ma, and P.M.B. Vitanyi. The similarity metric. *Information Theory, IEEE Transactions on*, 50(12):3250–3264, Dec 2004.

[59] Ming Li and Paul M.B. Vitnyi. *An Introduction to Kolmogorov Complexity and Its Applications*. Springer Publishing Company, Incorporated, 3 edition, 2008.

[60] Ting Liu, Andrew W Moore, Alexander G Gray, and Ke Yang. An investigation of practical approximate nearest neighbor algorithms. In *NIPS*, volume 5, pages 32–33, 2004.

[61] Jean loup Gailly and Mark Adler. zlib 1.28. http://www.zlib.net/, 2013.

[62] J. B. MacQueen. Some methods for classification and analysis of multivariate observations. In *Proc. of the fifth Berkeley Symposium on Mathematical Statistics and Probability*, volume 1, pages 281–297. University of California Press, 1967.

[63] Matt Mahoney. Large text compression benchmark. *URL: http://www. mattmahoney. net/text/text. html*, 2009.

[64] Matt Mahoney. *Data Compression Explained*. 2013.

[65] Nagapramod Mandagere, Pin Zhou, Mark A Smith, and Sandeep Uttamchandani. Demystifying data deduplication. In *Proceedings of the ACM/IFIP/USENIX Middleware '08 Conference Companion*, pages 12–17, New York, NY, USA, 2008. ACM.

[66] Gurmeet Singh Manku, Arvind Jain, and Anish Das Sarma. Detecting near-duplicates for web crawling. In *Proceedings of the 16th international conference on World Wide Web*, pages 141–150. ACM, 2007.

[67] Christopher D Manning, Prabhakar Raghavan, and Hinrich Schütze. *Introduction to information retrieval*, volume 1. Cambridge university press Cambridge, 2008.

[68] Mark McCartin-Lim, Andrew McGregor, and Rui Wang. Approximate principal direction trees. 2012.

[69] Geoffrey J McLachlan and Kaye E Basford. Mixture models. inference and applications to clustering. *Statistics: Textbooks and Monographs, New York: Dekker, 1988*, 1, 1988.

[70] Chirag Mehta. Analyzing how people talk on twitter. http://ktype.net/wiki/research:articles:progress_20110209, 2 2011.

[71] Amar Mudrankit. A context aware block layer: The case for block layer deduplication. Master's thesis, Stony Brook University.

[72] Marius Muja and David G. Lowe. Fast approximate nearest neighbors with automatic algorithm configuration. In *In VISAPP International Conference on Computer Vision Theory and Applications*, pages 331–340, 2009.

[73] Marius Muja and David G Lowe. Fast approximate nearest neighbors with automatic algorithm configuration. In *VISAPP (1)*, pages 331–340, 2009.

[74] Marius Muja and David G Lowe. Flann, fast library for approximate nearest neighbors. http://www.cs.ubc.ca/research/flann/, 2009.

[75] Athicha Muthitacharoen, Benjie Chen, and David Mazières. A low-bandwidth network file system. *SIGOPS Oper. Syst. Rev.*, 35(5):174–187, October 2001.

[76] Gonzalo Navarro. A guided tour to approximate string matching. *ACM computing surveys (CSUR)*, 33(1):31–88, 2001.

[77] Landon Curt Noll. Fowler/noll/vo (fnv) hash. *Accessed Jan*, 2012.

[78] Fujitsu Intel SUSE STRATO Oracle, Red Hat. Btrfs (b-tree file system). https://btrfs.wiki.kernel.org/index.php/Compression, 2007.

[79] Zan Ouyang, Nasir D. Memon, Torsten Suel, and Dimitre Trendafilov. Cluster-based delta compression of a collection of files. In *Proceedings of the 3rd International Conference on Web Information Systems Engineering*, WISE '02, pages 257–268, Washington, DC, USA, 2002. IEEE Computer Society.

[80] Byung-Hoon Park and Hillol Kargupta. Distributed data mining: Algorithms, systems, and applications. pages 341–358, 2002.

[81] Hae-Sang Park and Chi-Hyuck Jun. A simple and fast algorithm for k-medoids clustering. *Expert Systems with Applications*, 36(2):3336–3341, 2009.

[82] Dan Pelleg, Andrew W Moore, et al. X-means: Extending k-means with efficient estimation of the number of clusters. In *ICML*, pages 727–734, 2000.

[83] Calicrates Policroniades and Ian Pratt. Alternatives for detecting redundancy in storage systems data. In *Proceedings of the Annual Conference on USENIX Annual Technical Conference*, ATEC '04, pages 6–6, Berkeley, CA, USA, 2004. USENIX Association.

[84] M. O. Rabin. Fingerprinting by random polynomials. *IBM Research Report*, 2003.

[85] Anand Rajaraman and Jeffrey David Ullman. *Mining of massive datasets*. Cambridge University Press, 2012.

[86] Raimundo Real and Juan M Vargas. The probabilistic basis of jaccard's index of similarity. *Systematic biology*, pages 380–385, 1996.

[87] Jorma Rissanen and Glen G Langdon Jr. Arithmetic coding. *IBM Journal of research and development*, 23(2):149–162, 1979.

[88] Vassil Roussev. Data fingerprinting with similarity digests. In *Advances in Digital Forensics VI*, pages 207–226. Springer, 2010.

[89] Mark Ruijter. Lessfs. http://www.cs.ubc.ca/research/flann/, 2009.

[90] Nachiketa Sahoo, Jamie Callan, Ramayya Krishnan, George Duncan, and Rema Padman. Incremental hierarchical clustering of text documents. In *Proceedings of the 15th ACM international conference on Information and knowledge management*, pages 357–366. ACM, 2006.

[91] Khalid Sayood. *Introduction to data compression*. Newnes, 2012.

[92] D Sculley and Carla E Brodley. Compression and machine learning: A new perspective on feature space vectors. In *Data Compression Conference, 2006. DCC 2006. Proceedings*, pages 332–341. IEEE, 2006.

[93] Julian Seward. bzip2 1.06. http://www.bzip.org/, 2010.

[94] Claude Elwood Shannon. A mathematical theory of communication. *ACM SIGMOBILE Mobile Computing and Communications Review*, 5(1):3–55, 2001.

[95] Philip Shilane, Grant Wallace, Mark Huang, and Windsor Hsu. Delta compressed and deduplicated storage using stream-informed locality. In *Proceedings of the 4th USENIX Conference on Hot Topics in Storage and File Systems*, HotStorage'12, pages 10–10, Berkeley, CA, USA, 2012. USENIX Association.

[96] Robin Sibson. Slink: an optimally efficient algorithm for the single-link cluster method. *The Computer Journal*, 16(1):30–34, 1973.

[97] Torsten Suel and Nasir Memon. Algorithms for delta compression and remote file synchronization. In *In Khalid Sayood, editor, Lossless Compression Handbook*. Academic Press, 2002.

[98] Milan Svoboda. Fusecompress. https://code.google.com/p/fusecompress/, July 2008.

[99] Miklos Szeredi. Filesystem in userspace (fuse). http://fuse.sourceforge.net/, July 2013.

[100] Ben Taskar. Probabilistic classification and clustering in relational data. In *In Proceedings of the Seventeenth International Joint Conference on Artificial Intelligence*, pages 870–878, 2001.

[101] Windsor W. Hsu Timothy E. Denehy. Duplicate management for reference data. *IBM Research Report*, 2003.

[102] Mark R Titchener, Radu Nicolescu, Ludwig Staiger, Aaron Gulliver, and Ulrich Speidel. Deterministic complexity and entropy. *Fundamenta Informaticae*, 64(1):443–461, 2005.

[103] Koji Tsuda and Taku Kudo. Clustering graphs by weighted substructure mining. In *Proceedings of the 23rd International Conference on Machine Learning*, ICML '06, pages 953–960, New York, NY, USA, 2006. ACM.

[104] Nakul Verma, Samory Kpotufe, and Sanjoy Dasgupta. Which spatial partition trees are adaptive to intrinsic dimension? In *Proceedings of the Twenty-Fifth Conference on Uncertainty in Artificial Intelligence*, pages 565–574. AUAI Press, 2009.

[105] Ulrike Von Luxburg. A tutorial on spectral clustering. *Statistics and computing*, 17(4):395–416, 2007.

[106] Robert A Wagner and Michael J Fischer. The string-to-string correction problem. *Journal of the ACM (JACM)*, 21(1):168–173, 1974.

[107] Terry A. Welch. A technique for high-performance data compression. *Computer*, 17(6):8–19, 1984.

[108] Dwi H Widyantoro, Thomas R Ioerger, and John Yen. An incremental approach to building a cluster hierarchy. In *Data Mining, 2002. ICDM 2003. Proceedings. 2002 IEEE International Conference on*, pages 705–708. IEEE, 2002.

[109] Matt Williams and Tamara Munzner. Steerable, progressive multidimensional scaling. In *Information Visualization, 2004. INFOVIS 2004. IEEE Symposium on*, pages 57–64. IEEE, 2004.

[110] Wen Xia, Hong Jiang, Dan Feng, and Yu Hua. Silo: A similarity-locality based near-exact deduplication scheme with low ram overhead and high throughput. In *Proceedings of the 2011 USENIX Conference on USENIX Annual Technical Conference*, USENIXATC'11, pages 26–28, Berkeley, CA, USA, 2011. USENIX Association.

[111] Rui Xu and D. Wunsch, II. Survey of clustering algorithms. *Trans. Neur. Netw.*, 16(3):645–678, May 2005.

[112] Lawrence You and Christos Karamanolis. Evaluation of efficient archival storage techniques. In *Proceedings of the 21st IEEE / 12th NASA Goddard Conference on Mass Storage Systems and Technologies*, April 2004.

[113] Lawrence L. You, Kristal T. Pollack, and Darrell D. E. Long. Deep store: An archival storage system architecture. In *Proceedings of the 21st International Conference on Data Engineering*, ICDE '05, pages 804–8015, Washington, DC, USA, 2005. IEEE Computer Society.

[114] Lawrence L. You, Kristal T. Pollack, Darrell D. E. Long, and K. Gopinath. Presidio: A framework for efficient archival data storage. *Trans. Storage*, 7(2):6:1–6:60, July 2011.

[115] Lihi Zelnik-Manor and Pietro Perona. Self-tuning spectral clustering. In *NIPS*, volume 17, page 16, 2004.

[116] Tian Zhang, Raghu Ramakrishnan, and Miron Livny. Birch: an efficient data clustering method for very large databases. In *ACM SIGMOD Record*, volume 25, pages 103–114. ACM, 1996.

[117] Benjamin Zhu, Kai Li, and Hugo Patterson. Avoiding the disk bottleneck in the data domain deduplication file system. In *Proceedings of the 6th USENIX Conference on File and Storage Technologies*, FAST'08, pages 18:1–18:14, Berkeley, CA, USA, 2008. USENIX Association.

[118] Jacob Ziv and Abraham Lempel. A universal algorithm for sequential data compression. *IEEE Transactions on information theory*, 23(3):337–343, 1977.

Appendix A

List of Abbreviations

FNV	Fowler–Noll–Vo hash [77]
ICBCS	Incremental Clustering-Based Compression System The system presented in this thesis (3)
n-gram	Substring of length n (2.6.7)
simhash	Similarity hash (3.4)
NCD	Normalized Compression Distance (2.6.5.1)
KD-tree	K-Dimensional tree (2.7.6)
block	Fixed size part of a file (3.3)
chunk	Variable (content-defined) part of a file (3.3)
chunk spread	Range of chunk sizes (4.9)
SLINK	Single linkage algorithm [96]
KNN	K-Nearest Neighbors (2.7.6)
ANN	Approximate K-Nearest Neighbors (2.7.6)
archivedata	Extended metadata to provide archival functionality (3.9.1)

Appendix B

ICBCS Program Options

B.1 CLI Runtime Parameters

```
Usage:
  icbcs [options] filename
  icbcs [options] --file=filename

Options:
-h [ --help ]                    Print help messages

-c [ --compressor ] arg (=deflate) Compression algorithm: bzip2, deflate.

-l [ --comprlevel ] arg (=5)     Compression level: 1-9.

-p [ --type ] arg (=clustering)  Compression type: solid, non-solid,
                                 clustering.

-s [ --chsize ] arg (=12)        Average size of a chunk in bytes. Given as
                                 binary exponent: 3-20 (8 = 256B, 12 = 4kB,
                                 20 = 1MB).

--chspread arg (=2)              Chunk size spread. Given as binary
                                 exponent delta. 0 means all chunks are of
                                 average size. Range: 0..5

-n [ --nodeduplication ]         Disable deduplication. (default: false)

-d [ --distance ] arg (=simhash) Distance function used. One of: ncd,
                                 ncd-repr, simhash, simhash-deep,
                                 simhash-deep-repr

--ncd-repr arg (=4)              Number of representative chunks for NCD
                                 computation. Only valid for
                                 --distance=ncd-repr.
```

```
-w [ --sh-width ] arg (=32)      Simhash width. One of: 4, 8, 16, 32, 64.
                                 Only valid for --distance=simhash*.

--sh-ngram arg (=4)              N-gram to use for simhash generating.Int:
                                 1-128. Only valid for --distance=simhash*.
                                 Note that 4-gram is highly optimized.

--sh-kd-checks arg (=32)         Number of KD tree leaf checks. Only valid
                                 for --distance=simhash*.

--sh-depth arg (=2)              Depth of descent for inter-cluster
                                 distance evaluation. Only valid for
                                 --distance=simhash-deep or
                                 --distance=simhash-deep-repr.

--sh-repr arg (=2)               Number of representatives for each
                                 clusters. Only valid for
                                 --distance=simhash-deep-repr.

--sh-repr-depth arg (=2)         Depth of search for representative
                                 candidates used to determine
                                 representatives. Only valid for
                                 --distance=simhash-deep-repr.

-u [ --noupmerging ]             Disable upmerging. (default: false)

-g [ --groupsize ] arg (=64)     Size of a compression group in chunks.

-t [ --tolerance ] arg (=-2)     Balancing tolerance - accumulates to
                                 prevent excessive balancing. Bigger number
                                 > 0 means faster, but less precise
                                 balancing, -1 for single step balancing,
                                 -2 for no balancing at all.
```

B.2 Compile Options

As defines to CMake:

```
Profiling using gprof:
    -DProfiling=ON

Additional debug/measurement outputs:
    -DSimhashDistance_MINMAX
    -DSimhashDistance_VARIANCE
    -DSimpleSystem_VARIANCE
    -DStorage_VARIANCE
```

Appendix C

Measured Variables

This is a complete list of all measured variables that were used in the Chapter 4. All data from the measurements are attached to this thesis.

```
Chunkizer:
    Total unique chunks size
    Total duplicate chunks size
    Average unique chunk size

Deduplication:
    Unique chunks
    Duplicate chunks
    Deduplication ratio

Clustering:
    HC Disbalance
    HC Average depth

Grouping:
    Compressed chunks
    Compression groups

Metadata:
    Metadata size
    Compressed metadata size
    Relative metadata size
    Relative compressed metadata size

Archivedata:
    Archivedata size
    Compressed archivedata size
    Relative archivedata size
    Relative compressed archivedata size
```

```
Compression ratio:
    Original size
    Compressed size
    Compression ratio
    Compression with metadata ratio
    Compression with metarchivedata ratio

Running times:
    Chunking Time
    Chunking User+Sys time
    Clustering Time
    Clustering User+Sys time
    Compression Time
    Compression User+Sys time
    Total Time
    Total User+Sys time

Memory:
    Peak memory usage
    Peak memory without src file
    Peak memory stripped
```

Appendix D

Contents of the Attached DVD

```
/
├── datasets ..... datasets used for measurements
├── src
│   ├── icbcs ..... ICBCS source
│   │   ├── ...
│   │   ├── CMakeList.txt
│   │   └── generate-calltree.sh ..... calltree graph generator
│   ├── data ..... measured data
│   └── thesis ..... thesis source
└── text
    └── krcallub-mt2014.pdf ..... this thesis
```