String Hashing for Collection-Based Compression

Andrew Gregory Peel

orcid.org/0000-0001-6710-7513

Submitted in total fulfilment of the requirements of the degree of Doctor of Philosophy

December 2015

Department of Computing and Information Systems
Melbourne School of Engineering
The University of Melbourne

Produced on archival quality paper
Abstract

Data collections are traditionally stored as individually compressed files. Where the files have a significant degree of similarity, such as genomes, incremental backup archives, versioned repositories, and web archives, additional compression can be achieved using references to matching data from other files in the collection. We describe compression using long-range or inter-file similarities as *collection-based compression* (CBC). The principal problem of CBC is the efficient location of matching data. A common technique for multiple string search uses an index of hashes, referred to as fingerprints, of strings sampled from the text, which are then compared with hashes of substrings from the search string. In this thesis, we investigate the suitability of this technique to the problem of CBC.

A CBC system, *cobald*, was developed which employs a two-step scheme: a preliminary long-range delta encoding step using the fingerprint index, followed by a compression of the delta file by a standard compression utility. Tests were performed on data collections from two sources: snapshots of web crawls (54 Gbytes) and multiple versions of a genome (26 Gbytes). Using an index of hashes of fixed-length substrings of length 1024 bytes, significantly improved compression was achieved. The compression of the web collection was six times more than the compression achieved by *gzip* and three times *7-zip*. The genome collection was compressed ten times better than *gzip* and *7-zip*. The compression time was less than taken by *7-zip* to compress the files individually. The use
of content-defined selection of substrings for the fingerprint index was also investigated, using chunking techniques. The overall result is a dramatic improvement on compression compared to existing approaches.

String hash functions are an essential component of the CBC system, which have not been thoroughly investigated in the literature. As the fingerprint must remain unchanged for the lifetime of the collection, schemes such as universal hashing which may require re-hashing are not suitable. Instead, hash functions having randomness properties similar to cryptographic hash functions, but with much faster computation time, are preferred. Many functions in wide use are purported to have these properties, but the reported test results do not support conclusions about the statistical significance of the tests. A protocol for testing the collision properties of hash functions was developed having a well-defined statistical significance measure, and used to assess the performance of several popular string hashing functions.

Avalanche is a property of hash functions which is considered to indicate good randomness. Avalanche is essential for good cryptographic hash functions, but the necessity of avalanche for non-cryptographic hash functions, which do not need to conceal their operation, has not been thoroughly investigated. This question was addressed indirectly by investigating the effect of function resilience when hashing biased data. The resilience of a function directly limits the degree of avalanche. A proof is given that a highly resilient function is optimal when hashing data from a zero-order source, while experiments show that high resilience is less advantageous for data biased in other ways, such as natural language text. Consequently, we conclude that a good hash function requires some degree of both resilience and avalanche.
Declaration

I declare that:

(i) this thesis comprises only my original work towards the PhD except where indicated in the Preface;

(ii) due acknowledgement has been made in the text to all other material used; and

(iii) the thesis is fewer than 100,000 words in length, exclusive of tables, maps, bibliographies and appendices.

Andrew Peel,
December 3, 2015.
Preface

Chapter 3 and Chapter 4 were substantially previously published as:


None of the work described in this thesis has been submitted for other qualifications. None of the work described in this thesis was carried out prior to PhD candidature enrolment. Writing and editing of this thesis was undertaken solely by the author.
Acknowledgements

I am immensely grateful to my supervisors, Professor Justin Zobel and Associate Professor Tony Wirth. This thesis would not have been possible without their oversight and advice. Having taken over five years, it has placed an unusually high demand on their time and attention. They have guided my development as a scholar and their constant care and encouragement are greatly appreciated. I am also grateful for the support given by Associate Professor Andrew Turpin, the third member of my thesis committee.

My studies were financially supported by an Australian Postgraduate Award scholarship from the Australian Government, and a supplementary scholarship from NICTA, which is funded by the Australian Government as represented by the Department of Broadband, Communications and the Digital Economy and the Australian Research Council through the ICT Centre of Excellence program. My travel to the CIKM and SPIRE conferences in 2011 was partially funded by a grant from Google inc. In addition, the University of Melbourne provided facilities and technical support for the duration of my studies.

Thanks to Pedro Reviriego who provided his Matlab code to calculate the exact solution of the balls-in-bins problem. Thanks also to Halil Ali who provided the Web Snap-shots data collection.

I extend special thanks to Dr Kerri Morgan for introducing me to the mathematics of finite fields, for reading my paper drafts, and for her encouragement and
friendship. I also thank Professor Tony Wirth for convening TRICS, a reading
group in theoretical computer science, where I obtained a broad understanding
of the field beyond the narrow focus of my PhD.

I thank the professional staff of the CIS department, especially Rhonda
Smithies, Madalain Dolic and Leonora Strocio, for their assistance on many occa-
sions. Also, the Engineering Faculty IT team, especially Thomas Weichert, who
supported my computing needs.

I am also grateful for the experience of tutoring and lecturing an undergrad-
uate subject for the CIS department, and the guidance and support of Professor
Peter Stuckey, the previous lecturer.

Finally, I am immensely grateful to my family, Gilda, Lorenza and Giuliana.
This thesis has taken substantially longer than anticipated, and they have waited
patiently for me to finish before we embark on new plans. I would not have
completed this thesis without their continual affection and support.
Contents

Abstract i

Declaration iii

Preface v

Acknowledgements vii

1 Introduction 1
  1.1 Problem Statement ................................. 4
  1.2 Aim and Scope ................................... 8
  1.3 Significance .................................... 10
  1.4 Thesis Overview .................................. 11

2 Literature Review 13
  2.1 String Hashing .................................. 14
    2.1.1 Notation .................................. 15
    2.1.2 Hash Functions ............................. 15
    2.1.3 Desirable Properties ....................... 15
    2.1.4 Ideal Hash Function ....................... 17
    2.1.5 Universal Hashing ......................... 18
    2.1.6 Balanced Hash Functions for Biased Keys .......... 22
2.2 Non-Universal Hash Functions for Strings .................................. 30
  2.2.1 Testing Hash Functions .................................................. 36
2.3 Recursive Hash Functions ..................................................... 41
2.4 String Search ................................................................. 49
  2.4.1 Fundamental Algorithms for String Search ......................... 49
  2.4.2 The Karp-Rabin Algorithm .............................................. 51
  2.4.3 Fingerprint Indexes for String Search ............................... 53
  2.4.4 Compressed Self-Indexes ................................................. 58
2.5 Delta Encoding ............................................................... 61
  2.5.1 Optimal Encoding ......................................................... 63
  2.5.2 Suboptimal Encoding with Minimum Match Length .............. 65
2.6 Content-Defined Selection and Chunking ................................. 69
2.7 Delta Compression Systems .................................................. 80
2.8 Resemblance Detection ....................................................... 82
2.9 Collection Compression ...................................................... 84
2.10 Summary ........................................................................... 90

3 Collection-Based Compression .................................................. 91
  3.1 Overview of the CBC Scheme ................................................ 93
  3.2 Detailed Description ........................................................... 97
  3.3 Independence of Recursive Hash Functions ......................... 106
  3.4 Complexity of the Encoding Algorithm ............................. 108
  3.5 Performance Evaluation ....................................................... 112
  3.6 Data Collections .............................................................. 113
  3.7 Results: Fixed-Offset Selection ........................................... 115
  3.8 Content-Defined Chunk Selection ....................................... 124
  3.9 Results: Content-Defined Selection ................................... 130
Contents

3.10 Comparison With Other Systems ....................................... 133
3.11 Discussion ...................................................................... 142

4 CBC with Compressed Collections ........................................... 145
  4.1 Evaluation ...................................................................... 150
  4.2 Results: Compressed Collections ...................................... 151
  4.3 Results: CBC on a Compressed Collection .......................... 165
  4.4 Discussion ...................................................................... 171

5 Hash Function Testing .............................................................. 175
  5.1 Black-Box Collision Test .................................................. 177
  5.2 Validation of the Collision Test ........................................... 187
  5.3 Results Without Bit-Selection ............................................ 189
  5.4 Results With Bit-Selection ............................................... 193
  5.5 Discussion ...................................................................... 198

6 Hashing Biased Data ................................................................. 201
  6.1 Resilient Hash Functions on Biased Data ............................ 203
  6.2 Case Study of a Function Family ....................................... 213
  6.3 Test For Effective Resilience ............................................. 223
  6.4 Validation of the Resilience Test ....................................... 227
  6.5 Hashing English Text ....................................................... 231
  6.6 Resilience of Popular Hash Functions ............................... 236
  6.7 Discussion ...................................................................... 238

7 Conclusions ........................................................................ 243
  7.1 Thesis Outcomes ............................................................. 244
  7.2 Further Work ................................................................. 246
Bibliography 251
List of Figures

1.1 Conceptual illustration of delta compression ............... 6

2.1 SpookyHash function inner loop .......................... 33
2.2 MurmurHash3 function inner loop .......................... 34
2.3 Illustration of the phased offset problem .................... 71
2.4 Content-defined selection methods .......................... 75

3.1 Functional diagram of the collection-based compression scheme .. 94
3.2 Encoding example ........................................... 96
3.3 Encoding example with data structure values ................... 98
3.4 Index data structure .......................................... 101
3.5 Encoding effectiveness with empty and filled indexes ............. 117
3.6 Compression with a range of n-gram lengths ................... 119
3.7 The location of n-grams in content-defined chunks .............. 126
3.8 Compression with a range of maximum chunk lengths ............ 131
3.9 Observed distribution of chunk lengths ....................... 135
3.10 Encoding time with a range of maximum chunk lengths .......... 136

4.1 Functional diagram of the collection compression scheme ........ 147
4.2 Example of encoding and inserting two files into a collection ...... 148
4.3 Sizes of compressed delta files ............................. 156
<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.4</td>
<td>Cumulative compressed collection size</td>
<td>158</td>
</tr>
<tr>
<td>4.5</td>
<td>Cumulative compressed collection size with various parameters</td>
<td>159</td>
</tr>
<tr>
<td>4.6</td>
<td>Collection compression times</td>
<td>161</td>
</tr>
<tr>
<td>4.7</td>
<td>Encode and insert times with various parameters</td>
<td>162</td>
</tr>
<tr>
<td>4.8</td>
<td>Maximum memory used for collection compression</td>
<td>164</td>
</tr>
<tr>
<td>4.9</td>
<td>Maximum compression memory with various parameters</td>
<td>166</td>
</tr>
<tr>
<td>4.10</td>
<td>Linearity of the CBC encode time with input file size</td>
<td>167</td>
</tr>
<tr>
<td>4.11</td>
<td>Linearity of CBC memory use with the size of the collection</td>
<td>170</td>
</tr>
<tr>
<td>5.1</td>
<td>Probability distribution of LLPS for 256 balls into 128 bins</td>
<td>179</td>
</tr>
<tr>
<td>5.2</td>
<td>Relation between balance and unused bins</td>
<td>183</td>
</tr>
<tr>
<td>5.3</td>
<td>Collision test statistical power for several numbers of trials</td>
<td>184</td>
</tr>
<tr>
<td>5.4</td>
<td>Derived function hash value distributions</td>
<td>190</td>
</tr>
<tr>
<td>5.5</td>
<td>Results for the test without bit-selection</td>
<td>192</td>
</tr>
<tr>
<td>5.6</td>
<td>Results for the test using bit selection</td>
<td>196</td>
</tr>
<tr>
<td>5.7</td>
<td>Comparison of observed LLPS with and without bit-selection</td>
<td>199</td>
</tr>
<tr>
<td>6.1</td>
<td>Representative values of unbalanced fraction, $f_{j,t}$</td>
<td>210</td>
</tr>
<tr>
<td>6.2</td>
<td>Biased symbol probability distributions from values of $\alpha$</td>
<td>229</td>
</tr>
<tr>
<td>6.3</td>
<td>Results of the resilience test</td>
<td>233</td>
</tr>
<tr>
<td>6.4</td>
<td>Hashing English text with resilient functions</td>
<td>235</td>
</tr>
<tr>
<td>6.5</td>
<td>Resilience of popular hash functions</td>
<td>237</td>
</tr>
</tbody>
</table>
List of Tables

2.1 Comparison of content-defined selection methods ............... 73

3.1 Summary of data collections ..................................... 114
3.2 Encoding effectiveness with empty and filled indexes .......... 116
3.3 Compression with a range of n-gram lengths .................. 120
3.4 Resource use by cobald ........................................... 123
3.5 Encoding effectiveness with various maximum chunk lengths ... 132
3.6 Resource use with content-defined selection .................. 134
3.7 Compression by cobald compared with other delta encoders ... 137
3.8 Maximum memory used by cobald and other delta encoders .... 139
3.9 Compression time by cobald and other delta encoders .......... 140
3.10 Decompression time by cobald and other delta encoders ....... 141

4.1 Configurations of cobald in the collection compression experiments. 152
4.2 Sizes of the compressed collections ............................ 153
4.3 Compressed collection improvement due to cobald ............. 154

5.1 Probability distributions of balls into bins .................... 179
5.2 Collision test critical regions with several numbers of trials .. 181
5.3 Results for the test without bit-selection ..................... 191
5.4 Results for the test using bit-selection ......................... 195
List of Tables

6.1 Function case study: mapping of elements of $F_s$ . . . . . . . . . . . . . . . 216
6.2 Function case study: hash probability distribution. . . . . . . . . 217
6.3 Function case study: function pre-images. . . . . . . . . . . . . . 218
6.4 Function case study: symbol counts in pre-images of $f_0$ . . . . . . . 219
6.5 Function case study: 2-gram counts in pre-images of $f_1$ . . . . . . . 220
6.6 Example: number of unbalanced keys . . . . . . . . . . . . . . . . . . . . . . 222
6.7 Standard deviation of key counts vs $k$. . . . . . . . . . . . . . . . . . . 224
6.9 Degree of bias of symbol distribution for the test validation. . . . 230
6.10 Results of the resilience test . . . . . . . . . . . . . . . . . . . . . . . . . . . 232
List of Algorithms

1. FNV-1a hash function ........................................... 32
2. Non-recursive hash by general polynomial division .......... 44
3. Recursive hash by general polynomial division ............... 45
4. The Karp-Rabin string search algorithm ....................... 51
5. String search using a sampled fingerprint index ............. 56
6. Greedy algorithm for delta encoding ............................ 63
7. Encoding with fixed-offset sampled fingerprint index ........ 104
8. Encoding with content-defined sampled fingerprint index .... 127
Chapter 1

Introduction

Efficient storage of large data collections is a pressing problem. While advances in storage technology continue to reduce the cost of data storage, these savings are currently outpaced by the demand for additional storage. As a result, the need for data storage is an increasingly restrictive constraint on data-intensive endeavours. This thesis is an exploration of novel approaches to repository-scale compression.

To illustrate, consider the field of genomics, which is the original motivating application for this thesis. Since 2000, when the first sequence of the human genome was completed, sequencing technology has advanced extremely rapidly, and now a human genome can be sequenced for less than US$5,000 (Hayden, 2014). Enabled by these technological advances, a deluge of data is being generated and stored in scientific collections of genome sequences from numerous species and from thousands of individual humans. Furthermore, some leading clinics¹ now sequence the genomes of patients for routine medical diagnosis. The difficulty of storing these huge and rapidly growing collections is widely recog-

¹For instance, the Mayo clinic medical genome facility, http://mayoresearch.mayo.edu/center-for-individualized-medicine/medical-genome-facility.asp
nised (Schatz and Langmead, 2013). Researchers, having filled their data stores, are having to delete old data in order to accommodate new data, and are no longer able to store some intermediate data (Dublin, 2011).

The total size of some genomic data collections is vast. For example, the 1000 genomes project\(^2\) occupied 464 Tbytes\(^3\) in 2013. A human genome, having slightly more than 3.3 billion base pairs with each pair encoding two bits of information, can be represented as approximately 750 Mbytes of data. In practice, an individual genome occupies many gigabytes as additional metadata associated with each base pair is stored. If we conservatively estimate that an individual genome is 1 Gbyte in size, the genomes of 1 million people is approximately 1 petabyte (PB, \(10^{15}\) bytes), costing at least US$1.5 million to store for 5 years.\(^4\) Moreover, the rapid advance of storage technology is not expected to solve the problem. The density of magnetic storage on hard disks increases approximately 40\% per year, and alternative storage media are unlikely to surpass its cost effectiveness before the year 2020 (Kryder and Kim, 2009). While this is a remarkably rapid rate of progress, substantially faster than Moore’s law (the observation that the density of transistors on an integrated circuit doubles every two years), it has been outpaced by improvements in the cost of genomic sequencing.

Data compression has an essential role in managing the genome data storage problem, and the large public genome collections typically use standard compression tools. For example, the 1000 genomes project and the Ensembl project\(^5\), a database of annotated vertebrate genomes, publish their genome collections as

\(^2\)http://www.1000genomes.org
\(^3\)Throughout this thesis, data quantities are expressed in binary units: Kbytes = \(2^{10}\) bytes, Mbytes = \(2^{20}\) bytes, Gbytes = \(2^{30}\) bytes and Tbytes = \(2^{40}\) bytes. Where powers of 10 are used, this will be indicated explicitly, and written as KB (\(10^3\) bytes), MB (10^6 bytes), and so on.
\(^4\)Assuming each genome is uncompressed. The google cloud storage rate is US$0.026 / (Gbyte month) on 14 August, 2014. From https://developers.google.com/storage/pricing
\(^5\)http://www.ensembl.org
individually compressed *gzip* files. However, while compression of individual files is a mature technology, there is scope to obtain greater compression by using additional information about the files in the collection. The key source of additional information of relevance to this thesis is that genomes are similar to one another; for the human genome, individuals typically differ in approximately 0.4% of their base pairs (Tishkoff and Kidd, 2004). Hence, a genome can be encoded as the difference from a reference genome, such as by the scheme of Kuruppu et al. (2012), to obtain improved compression.

Genomics is not the only domain with high similarity between files in a collection, and in this thesis we pursue a general approach, rather than developing a special-purpose scheme tailored to each type of data we may encounter. Other sources of highly similar data include regular archiving of data that is continuously expanded and revised over time, such as web sites, repositories of computer source code and executables, and archives of email and other business documents that must include regular snap-shots in order to satisfy corporate regulations and enable legal discovery. A well-known instance from the web is the Wikimedia archive,\(^6\) which includes the text of Wikipedia. The monthly dumps are highly similar, differing by the changes to Wikipedia content since the previous dump. In August 2013, the archive was 30 Tbytes of compressed files, uncompressing to 4 to 5 times larger. Each monthly dump adds a further 1.8 Tbytes to the archive. However, the similarity between versions is not exploited to reduce the storage needs; files on the Wikimedia archive are individually compressed with 7-zip or *bzip2*.

Traditional, single-file compressors, such as *gzip*, *bzip2*, and 7-zip, achieve compression by finding similarities between two or more parts of the data, and encode later occurrences in terms of the earlier ones. However, they are only able

\(^6\)http://meta.wikimedia.org/wiki/Data_dumps
to identify similar data that is close to the first occurrence as they process the
input file in blocks, or in a sliding window, in order to limit their memory use.
We refer to similar data that cannot be identified by a single-file compressor,
either by being outside of the sliding window or block, or in separate files, as
long-range matching data, and a collection of these files has long-range similarity.
We define compression using similar data from any location in a collection of files
as collection-based compression (CBC).

1.1 Problem Statement

A CBC system aims to achieve improved compression by identifying matching
data in a large collection. This presents a significant search problem, and it is
clear that an index is essential: reading the entire collection each time a new
file is compressed is not feasible. In light of this observation, a successful CBC
system:

1. is able to compress an input string or file using information from any loca-
tion in the collection;

2. does not read the entire collection for each compression operation, but in-
stead uses an index to support the search for similarity;

3. achieves greater compression than traditional compression systems;

4. has time complexity linear with the size of the input file being compressed,
and the compression time should be comparable to or faster than compres-
sion and decompression with traditional compression systems;

5. has memory complexity which is sublinear in the collection size, or has a
constant memory limit with a gradual degradation in compression performance;

6. may have permanent supporting files, such as indexes, but they need to be small compared to the size of the compressed collection;

7. is able to delete a file from the collection in time sublinear in the collection size.

Compression of a genome by encoding the differences from a single reference genome does not meet our definition of CBC, as it does not satisfy the first point. Consider a collection of genomes compressed in this manner; the genomes are compressed with respect to the reference genome, and do not refer to any similar data from other genomes in the collection. A CBC system, as we have defined it, will achieve better compression when similar data exists in multiple files, but at the cost of indexing the entire collection, rather than just a single reference file.

**Delta compression** Many existing methods for compressing collections are based on *delta encoding*, the technique of encoding a file as a sequence of differences to a reference file. When a string in the input file has a duplicate in the reference file, it is replaced in the encoding, known as the *delta file*, by a pointer to the location in the reference file, and the length of match. By also encoding duplicates that match strings occurring earlier in the input file, Hunt et al. (1998) developed *delta compression*. Figure 1.1 illustrates how delta compression could be used for CBC.

To date, delta compression has not been used directly for CBC. There are several delta compressors available, such as *xdelta*, *vcdiff*, and *hsadelta*, which compress with respect to a single reference file, but compression using multiple reference files has, to our knowledge, not been pursued. Furthermore, the existing
1.1. Problem Statement

delta compressors read the reference file prior to reading the input file each time they compress a file, contrary to point 2 of our CBC requirements. Extending delta compression to perform CBC is the broad problem investigated in this thesis. We present a CBC system, named cobald, and use it to investigate several algorithms for indexing the collection files, and to demonstrate the effectiveness of CBC on suitable data.

**Hashing** There are many aspects of CBC that are worthy of detailed investigation, but we have given particular attention to the use of hashing in the search for matching strings. Early experience in this investigation showed us that match
discovery was a key obstacle to effective CBC. A key step in the delta compression algorithm is identifying matching strings in the input and reference files. The existing delta compressors use variations of the Karp-Rabin string search algorithm (Karp and Rabin, 1987), the most efficient method of searching for multiple patterns simultaneously in a string, taking time proportional to the total length of the string and patterns. A subset of the strings in the collection are stored in an index, referenced by their hash value, then matching strings in the input can be identified as they have the same hash. In many delta compressors, a rolling-hash function is employed, which efficiently finds the hash of a substring using the hash of the string at the previous position.

However, while hash-based search is the most efficient method for finding matching strings, there is evidence that the rolling-hash function is not a sensible choice for CBC. Lemire and Kaser (2010) showed that rolling-hash functions can be no better than pairwise independent, and may therefore have a higher collision rate in the hash index compared to an ideal random hash function. Excess collisions would slow the CBC operation, or perhaps reduce the compression achieved, depending on the implementation.

We are unable to turn to universal hashing (Carter and Wegman, 1979), in which amortized constant index query time is obtained by the rare necessity of rehashing the entire index with a new hash function. In a CBC scenario, this involves reading the entire collection, which we consider to be impractical. Instead, a single hash function must be used for the lifetime of the collection. One solution, employed in data de-duplication systems, is to use a cryptographic hash function. However, these are substantially slower than rolling-hash functions.

An alternative group of hash functions that may be suitable are the non-universal, non-cryptographic hash functions. These aim to achieve randomness properties similar to cryptographic hash functions at faster speeds. Recent exam-
1.2. Aim and Scope

In view of the uncertainty concerning the appropriate hash function for CBC, and the dependence of the algorithms on the choice of hash function, this thesis focuses on the aspects of CBC relating to hashing. The research aim is

To investigate effective collection-based compression using string hashing.
Several research questions arise immediately, which guide the course of the investigation: What is an effective length for the hashed substrings? Which substrings should be inserted in the index? Which hash-based string search algorithm is most effective for delta encoding? How large should the hashes be? Are the rolling-hash functions safe for a large collection? Can the non-universal, non-cryptographic functions be used instead? How can the hash function be tested for randomness?

The interdependence of these questions is an important aspect of this research investigation. For instance, inserting more substrings into the index allows more matching substrings to be found, achieving better compression of individual files. However, the index will be larger, increasing the overall storage required for the collection. Increasing the number of hash bits stored in the index will reduce the number of collisions, perhaps mitigating any departure from randomness of the hash function, but again at the cost of a larger index. There are many more examples of dependencies and trade-offs that are described throughout this thesis.

While we pursue a general approach to CBC, there are some types of data which are not amenable to such compression, and are beyond the scope of this research. Multimedia data is a significant instance. While it might be expected that multiple photographs of the same scene will be very similar, the random noise inherent in photography, coupled with compression in common multimedia file formats, results in files with little byte-level similarity. Similar problems occur in video and audio recordings, and medical images.

Compression for transmission of data is not investigated, with attention limited to compression for storage. The transmission problem is related, and our CBC approach may be useful for transmission, but it requires additional assumptions and models concerning the synchronisation of collection data available to the sender and receiver that we do not pursue in this thesis.
While we report experimental results for delta file sizes, we are primarily concerned with the amount of data removed during the encoding process, and have not given attention to efficient compression of the encoded file. This was because efficient coding has a separate and extensive literature that is largely peripheral to the problems of string search and delta encoding. Furthermore, efficient formats already exist for delta files, and the delta files were subsequently compressed using a traditional single-file compressor, which removed at least part of the redundancy in our simply formatted delta files.

Last, we assume that the data collections contain long-range similarity, usually as matching data between multiple files. The test collections used in experiments were selected from domains that generate data with this desired property.

1.3 Significance

This thesis makes several contributions. We have developed a CBC system, cobald, and demonstrated that CBC is feasible and provides a significant advantage over single-file compression using realistic data. We perform an empirical comparison of several algorithms and parameter choices for CBC, and identify those which achieve good compression.

A second outcome is a rigorous test to evaluate whether a hash function is indistinguishable from the ideal random hash function on the given test data. Currently, hash functions are usually evaluated by a relative comparison with other hash functions (Estébanez et al., 2013).

A third outcome is a study of the necessity of function resilience for non-universal, non-cryptographic hash functions. We prove that resilient functions are optimal for a certain type of zero-order biased data, then use this result to develop an empirical test for function resilience. This is a significant contribution
as the existing test regimes for non-universal, non-cryptographic hash functions do not have a test for resilience.

1.4 Thesis Overview

There are six further chapters in this thesis. In Chapter 2, we review the literature on string hashing, and algorithms employing hashing, from the perspective of CBC, including the desirable properties of hash functions, testing hash functions, delta encoding and compression, content-defined chunking and archive systems. In Chapters 3 and 4 we describe our CBC system, cobald, and explore the effectiveness of several algorithms and parameter choices by experimental comparisons. Chapter 3 focuses on the delta encoding operation when the collection is uncompressed, while Chapter 4 is concerned with the overall outcome of compressing entire collections, and performing CBC on a compressed collection. In Chapter 5 we present a statistical test for hash function randomness, and an experimental validation of its effectiveness. In Chapter 6 we study the effect of hash function resilience when hashing several classes of biased data. We prove that resilient functions are optimal for a certain type of zero-order biased data, then describe an empirical test for function resilience. Last, in Chapter 7 the conclusions of the study are presented, and a description of the further research directions arising from it.
1.4. Thesis Overview
Chapter 2

Literature Review

The task of collection-based compression (CBC) can be divided into several distinct steps or processes which have been widely investigated in the literature. The key processes are multi-pattern string search, primarily using the Karp-Rabin search algorithm, delta encoding and delta compression. The numerous variations of these techniques share a common thread: they employ string hashing in various ways. In this chapter, we describe these algorithms and review their limitations and trade-offs from the perspective of CBC.

One theme of this review is the behaviour of hash functions with biased data, and the effect that has on algorithms employing hashing. Hash-based algorithms typically work well in practise, but have poor worst case bounds for a limited set of situations that occur with very low probability. We consider the mathematical properties of effective hash functions and the relationship between the function and bias in the data. A second theme is the evaluation of hash functions. How do we decide whether a hash function is suitable for our CBC system? The final major theme is the necessity of heuristic algorithms for several steps in the CBC scheme. This arises from the poor worst-case bounds from non-universal hashing, as well as from hard algorithms at higher levels, such as delta encoding.
and collection compression.

This review is structured in a bottom-up manner. We begin with a presentation of string hashing as it is the foundational technique for the algorithms described in the later sections, and familiarity with the probabilistic limitations of hashing is necessary to understand the operation of these higher level techniques. This is followed by string search using the Karp-Rabin algorithm, then the extensive literature on delta encoding and delta compression. Finally, the prior work on compression of large data collections is reviewed.

2.1 String Hashing

We begin our review at the ground level, as it is necessary to understand string hashing before we can move on to algorithms which employ it, and also because the limitations of string hashing constrain their operation. After first introducing some notation and definitions, we consider the requirements for a good hash function in the context of the collection-based compression problem. The tension between randomness and determinism inherent in all practical hash functions is reviewed by considering the ideal hash function and why it is impractical. Then we describe the traditional approach of universal hashing, explaining why it is not suitable for CBC, before moving on to the mathematical properties of non-universal hash functions, giving a detailed description of the $t$-resilient functions, which are needed for the work in Chapter 6. We then review methods for evaluating hash functions using statistical tests. Finally, we described recursive or rolling hash functions in detail as they are widely used in systems employing the Karp-Rabin string search algorithm that is described in later sections.
2.1.1 Notation

A string $S = s_1 s_2 \ldots s_D$ is a sequence of symbols, letters, or characters from an alphabet $\Sigma$, $s_i \in \Sigma$, and $S \in \Sigma^D$, the set of all strings of length $D$. We can also write $s_i$, the $i$th elements of $S$, as $S[i]$, and say that $s_i$ is at the $i$th position or location in $S$. A substring $Q$ of string $S$ is a string which occurs in $S$ and can be formed by removing elements from the beginning and end of $S$. A substring beginning at position $i$ and ending at position $j$ in $S$ is written as $Q = S[i, j]$. A substring is a special instance of a subsequence, a selection of symbols from $S$ in the order they occur in $S$, but which are not necessarily adjacent. For example, in the string “echidna”, “hid” is a substring, and “chin” is a subsequence but not a substring. In this thesis we are concerned primarily with substrings.

2.1.2 Hash Functions

A hash function $h : U \rightarrow V$ maps the universe of possible input data $U$ to a smaller set $V$. The domain may be either a string of fixed length, $U = \Sigma^D$, or a string of variable length up to a maximum size, $U = \cup_{k \leq D} \Sigma^k$. The range is the set of natural numbers modulo $M$, $\{0, 1, \ldots, M - 1\}$. Typically, the $u \in U$ are referred to as keys and $v = h(u), v \in V$ as the values, hashes, or fingerprints. The hash function is a many-to-one mapping, and the pairs $u_1, u_2 \in U; u_1 \neq u_2, h(u_1) = h(u_2)$ are called collisions. The set of all keys mapping to a single value $\{h^{-1}(v)\}$ is the pre-image of $v$.

2.1.3 Desirable Properties

Hash functions should be deterministic, have low computational cost, uniformity, and randomness, which is a form of pseudo-randomness that we clarify below. A deterministic hash function always gives the same output to the same input. A
fast hash function is a necessity in applications involving frequent calculations, such as hashing each value of a large data set, or hashing all of the prefixes of a long string. A related aspect of the cost of a hash function is the rate of collisions. Collision resolution involves comparing the keys directly, which may incur significant cost, such as retrieval from external storage.

Informally, a hash function is said to be uniform if the probability of generating each value is equal. Uniformity depends both on the pre-images of the hash function and on the statistical properties of the key selection process. To relate this to a deterministic hash function, a balanced function has pre-images of equal size. It is readily apparent that a balanced hash function, $h$, hashing randomly selected keys will be uniform with $Pr[h(u_{random}) = v] = \frac{1}{M}$. A hash function obtaining a uniform distribution of values has the minimum expected number of collisions.

In practical situations however, the input may not be randomly selected but is a highly biased subset $X \subset U$. To minimise collisions, the hash function should provide a uniform distribution of hashes over all $X$ that may be expected. To achieve this, the hash function needs an additional property loosely described as randomness, which is considered in depth in the next section. To illustrate that balance alone is insufficient, a balanced hash function with $M = 2^8$ can be made by setting the value equal to the first byte of the string. As the function is balanced, the hash distribution will be uniform when the input is randomly selected from all possible strings. But in nearly all practical uses the values will be highly biased. For example, randomly selecting words from an English dictionary will produce many more $v = 'e'$ than $v = 'z'$.
2.1.4 Ideal Hash Function

A random oracle is an idealised hash function where the value for each key is randomly assigned (Bellare and Rogaway, 1993). When the oracle is asked for the value for a previously unseen key, it randomly selects a new value. Consequently, the expected value $h(u)$ of any key is uniformly distributed and the distribution of values over any subset of the input $X \subset U$ will be uniform. To appear deterministic, the oracle remembers all of its previous answers and provides the same answer to a previously requested key.

The random oracle has the useful property that the hash of a key is truly independent of all other keys. Knowledge of the value for some keys provides no information on the values for other keys (Bellare and Rogaway, 1993). However construction of a random oracle is not practical. The previously seen keys must be stored and each request requires increasing time to search the list of previous keys. An alternate construction for an ideal hash function over a finite domain would be to completely fill the truth table at the beginning, requiring space proportional to the size of the domain, which is exponential in the number of bits in each key (Mitzenmacher and Vadhan, 2008).

For ideal hash functions over a finite domain, a distinction is made between random ideal hash functions and regular ideal hash functions (Bellare and Kohno, 2004). A random hash function is a random selection from all possible mappings of $U \rightarrow V$ and many of these functions will not be balanced. Regular ideal hash functions are selected from the subset of mappings which are balanced.

While construction of the random oracle is random, it is a deterministic hash function. The probability distribution of values over $X \subset U$ is found by summing the probability of the keys in each pre-image. Observe, however, that many $X \subset U$ exist for which $h : X \rightarrow V$ is not balanced. An extreme example is when
X is a pre-image of $h$. These non-uniform outcomes cannot be eliminated, but may occur with a small but finite probability. So the behaviour of the random oracle on real input keys depends on the input data as well as the function. While a regular ideal hash function guarantees independence and balance, it can only achieve uniformity with some probability. Nonetheless, it is the best that can be done, and defines the benchmark for evaluating real hash functions.

A related observation is that, with an unbiased key set, all balanced hash functions have the same expected probability of a non-uniform hash distribution as a random allocation of balls into bins. So, when evaluating a real balanced hash function, we are concerned entirely with its performance on biased input data. As we have already observed, a hash function cannot be uniform on all possible sets of biased keys, and we are only able to achieve high expected uniformity over a subdomain. In Section 2.1.6 we review mathematical properties of balanced hash functions, and consider their relation to biased keys, and in Section 2.2.1 we review empirical methods for evaluating the uniformity of hash functions on biased keys.

### 2.1.5 Universal Hashing

Data structures based on hashing are limited in the worst case by the possibility that hash collisions may be common in the input data. For instance, a hash table storing $N$ keys with no collisions requires a single access to retrieve any key, but, when all keys collide at a single hash value, $O(N)$ accesses are needed, which is no better than traversing a linked list. All hash functions are subject to this limitation, including an ideal hash function, as it is a consequence of the existence of collisions.

*Universal hashing* (Carter and Wegman, 1979) overcomes this limitation by
selecting the hash function from a function family, achieving constant worst-case amortised time to query a separately-chained hash table. While universal hashing is not used in this thesis, we provide a brief description in order to explain why it is not applicable. Furthermore, it provides a context for key results regarding the effect of independence on the maximum number of collisions, which are needed in the investigation of hashing of biased data in Chapter 6.

When using universal hashing to fill a hash table, a function $h$ is randomly selected from a universal family $\mathcal{H}$ of hash functions $\{h \in \mathcal{H} : U \rightarrow V\}$ having the property that no pair of distinct input data are mapped to the same hash value by more than $\frac{1}{M}$ of the functions in $\mathcal{H}$. Specifically,

$$\{\forall x, y \in U; x \neq y\}, \quad \Pr_{h \in \mathcal{H}} [h(x) = h(y)] \leq \frac{1}{M}. \quad (2.1)$$

If a high number of collisions occurs during hash table construction, the entire input is re-hashed using another function randomly selected from $\mathcal{H}$. This procedure limits the worst-case behaviour, and the cost of re-hashing $O(N)$ can be amortised over all functions from $\mathcal{H}$ such that the expected amortised time per query is $O(1)$. Importantly, Equation 2.1 does not depend on the probability of $x$ or $y$ occurring, and universal hashing achieves the constant time bound regardless of the bias in the input data.

Universality is a weaker criterion than pairwise independence, whereby, for any pair of distinct keys $x, y, x \neq y$ and any hash values $v_1, v_2$,

$$\Pr [h(x) = v_1 \land h(y) = v_2] = \Pr [h(x) = v_1] \Pr [h(y) = v_2] = \frac{1}{M^2}. \quad (2.2)$$

Pairwise independent function families are described as strongly universal.2 or
2-independent (Wegman and Carter, 1979). Generally, a \textit{k-independent} or \textit{k-universal} function family has the property that the hash values for any \textit{k} keys are independent. For example, Wegman and Carter (1981) described the construction of a \textit{k-independent} family using polynomials on a finite field, \( h(x) = \left( \sum_{i=0}^{k-1} a_i x^i \right) \mod p \) \( \mod M \), where \( p \) is a prime \( \geq |U| \), and \( 0 < a_i < p \). The second modulo reduces the hash to the desired table size. When \( k = 1 \) and \( a_0 = 0 \), \( h(x) \) is universal, but not 2-independent.

Despite the theoretical elegance of universal hashing, there are hashing applications where it is not used, principally because the cost of rehashing the entire data structure is too high. Fingerprint indexes of large data sets on external storage, which is the scenario of interest in this thesis, is one such application. Rehashing the index entails reading the entire collection, taking many hours or perhaps days. As it is not feasible to rehash in normal operation, the hash function initially selected must be used for the entire lifetime of the data structure, and the worst case time bounds of universal hashing are no longer applicable. One remedy is to increase the size of the hash to accommodate the anticipated maximum size of the data structure as well as some additional bits as a precaution against the possibility of a non-uniform distribution of hashes. Also, care is taken to use a non-universal hash function which is likely to work well with similar data. A common approach is to use a cryptographic hash function and truncate it to the desired length. However cryptographic hash functions are very slow compared to non-cryptographic alternatives reviewed in the later sections.

Universal hashing has been used to obtain theoretical bounds on the expected number of hash collisions which demonstrate the direct connection between independence and uniformity. One measure of uniformity is the length of the longest chain in a hash table, which is the number of keys hashing to the value having the most collisions. Consider a separately-chained hash table of size \( M \) storing
$N$ keys, where the table size is increased as additional keys are inserted such that $M = \Theta(N)$. Using an ideal hash function the worst-case query time is $O\left(\frac{\lg N}{\lg\lg N}\right)$ with high probability, meaning that

$$\Pr \left[ l > (c + 1) \frac{\ln N}{\ln \ln N} \right] < \frac{1}{N^{c-1}} \quad (2.3)$$

where $l$ is the length of the longest chain, and $c$ is a constant chosen to obtain the desired bound (Raab and Steger, 1998; Gonnet, 1981). However, this bound is not achieved by function families with low independence. Alon et al. (1999) demonstrated a 2-independent family for which a set of keys of size $N$ exists such that the maximum chain length is always $\Omega(\sqrt{N})$. Schmidt et al. (1995) showed that $\Theta\left(\frac{\lg N}{\lg\lg N}\right)$-wise independence is sufficient to achieve the same bound as the ideal hash function. Pătraşcu and Thorup (2012) showed that a 3-independent hash function that achieves the ideal bound can be constructed using simple tabulation, in which small blocks of the input are hashed using a random lookup table, and then the block hashes are combined with a bit-wise exclude-OR operation. A similar dependence of uniformity on independence has been described for linearly probed hash tables (Knuth, 1973; Pătraşcu and Thorup, 2012; Pagh et al., 2009). In summary, pairwise independence achieves some degree of uniformity, but more independence is needed to match the uniformity of the ideal hash function.

We have already observed that uniformity depends on both the hash function and the input data. Mitzenmacher and Vadhan (2008) investigated the contribution to uniformity of randomness in the keys, motivated by the observation that universal hash functions nearly always perform better than predicted by worst case analysis. They found minimum bounds on the input data entropy necessary to asymptotically achieve a similar worst-case query time to an ideal
hash function, and observed a clear trade-off between independence and randomness in the input. Using Renyi entropy, a generalisation of the Shannon entropy, they found that, for a chained hash table, a 2-independent family requires approximately $3 \log N$ bits of entropy per key, and a 4-independent family requires $2 \log N$ bits. For linear probing, a 2-independent family requires $4 \log M$ bits, and a 4-independent family requires $2.5 \log M$ bits.

### 2.1.6 Balanced Hash Functions for Biased Keys

It was noted in Section 2.1.4 that an ideal hash function, in which the hashes are guaranteed to be mutually independent, requires exponential space. A practical hash function using polynomial space necessarily achieves less independence between hashes. On the other hand, it was also noted that any balanced hash function is expected to generate a uniform hash distribution with random input data. Clearly, the hash distribution depends on a combination of two sources of randomness: the variation in the data, and the pseudo-randomness of the hash function. If the combined randomness from these two sources is insufficient, a uniform hash distribution is unlikely to occur.

The property of “pseudo-randomness” is an informal term used here somewhat loosely. It highlights that a practical hash function cannot have true randomness as it is deterministic, and cannot have complete independence between the hash values. The term is adopted from a similar distinction made for pseudo-random sequences generated by deterministic functions. Here we review the mathematical properties of hash functions that are thought to contribute to their pseudo-randomness.

These concepts have been extensively investigated for cryptography. The essential goal of encryption is to transform a message (the plaintext) into a form
(the ciphertext) that appears to be independent of the plaintext. The ciphertext is indistinguishable from a random string, but the encryption must be deterministic in order for the original plaintext to be decrypted. Shannon (1949), investigating this tension between randomness and determinism, observed that a good cipher requires both diffusion to spread the information over as much of the ciphertext as possible, and confusion to hide any correlations between the message and the ciphertext, and that diffusion and confusion are in opposition to each other within a single function. The framework of confusion and diffusion is a convenient viewpoint of the cryptographic literature for our investigation of non-cryptographic hash functions. Function properties relating to confusion seem unlikely to be of interest as there is no need for a non-cryptographic hash function to hide its operation, while properties relating to diffusion will be desirable.

After introducing some notation and terminology, we describe several function properties and discuss their relevance to cryptographic and non-cryptographic functions. Much of this material is from the literature on cryptology as well as that on error-correcting codes. Our focus is on the relationship between the properties and the behaviour of the function on keys biased in certain ways. The cryptographic viewpoint considers biased keys from the point of view of mounting an attack on the function by hashing a carefully constructed key set. A deeper overview can be found in the reviews by Carlet (2010a,b). We are unaware of a review of the properties of these functions from a non-cryptographic perspective.

The function properties described below (resilience, non-linearity, avalanche) were first defined for Boolean functions, which have a single output taking the values \( \{0, 1\} \), also written as \( \mathbb{F}_2 \), the finite field with 2 elements. The properties have been extended to vector Boolean functions, \( \mathbb{F}_2^c \rightarrow \mathbb{F}_2^b, b \leq c \), being a set of \( b \) Boolean functions whose outputs are considered jointly. Vector Boolean functions are also referred to as \( S \)-boxes, or substitution boxes, in the cryptology literature.
2.1. String Hashing

When the number of output bits $b$ has a common divisor $d$ with the number of input bits $c$, the function may be represented as a univariate polynomial on the subfield $\mathbb{F}_q$ of $\mathbb{F}_2^n$ and $\mathbb{F}_2^b$, where $q = 2^d$, giving a function $\mathbb{F}_q^n \to \mathbb{F}_q^m$, $n = c/d$, $m = b/d$. This representation imposes an equivalence between $\mathbb{F}_2^d$ and $\mathbb{F}_2^{2d}$, and requires a mapping between the elements of $\mathbb{F}_2^d$ and $\mathbb{F}_2^{2d}$. Typically, the elements of $\mathbb{F}_2^{2d}$ are represented by polynomials of degree $d-1$ on an indeterminate variable in $\mathbb{F}_2$, and a simple mapping to $\mathbb{F}_2^d$ is obtained by interpreting the coefficient of the term with exponent $i$ as the value of the $i$-th vector Boolean function over $\mathbb{F}_2^d$.

For this investigation, the univariate polynomial representation is convenient as the input is grouped into symbols of $d$ bits in length, being elements of $\mathbb{F}_q$, which allows the expression of the input bias as a probability distribution over a larger alphabet. For example, $d = 7$ is sufficient to encode the ASCII alphabet. The representation restricts the hash value length to multiples of $d$ bits, but this is not a significant constraint in practise as the hash value can be truncated if a shorter hash is desired.

It is useful to express hash function $h$ as a polynomial, or sum of products, known as algebraic normal form (ANF),

\[
h(x) = a_{0,0,...,0} + a_{1,0,...,0}x_1 + a_{0,1,...,0}x_2 + \ldots + a_{0,0,...,1}x_n \\
+ a_{2,0,...,0}x_1^2 + a_{1,1,...,0}x_1x_2 + \ldots + a_{0,0,...,1}x_{n-1}x_n + a_{0,0,...,2}x_n^2 \\
+ \ldots \\
+ a_{q-1,q-1,...,q-1}x_1^{q-1}x_2^{q-1}\ldots x_n^{q-1} \\
= \sum_{i_1=0}^{q-1} \sum_{i_2=0}^{q-1} \ldots \sum_{i_n=0}^{q-1} a_{i_1,i_2,...,i_n}x_1^{i_1}x_2^{i_2}\ldots x_n^{i_n},
\]  

(2.4)
where \( x_i \in \mathbb{F}_q \) is the \( i \)-th symbol of the input key, and \( a_i \in \mathbb{F}_q \) are the polynomial coefficients. The coefficients are identified by \( n \) subscripts, each taking values from 0 to \( q - 1 \) matching the exponents of the variables in their corresponding term.

The algebraic degree of a function is the maximum of the sums of the exponents in each term. Specifically, any term may be written as \( a \prod_{i=1}^{n} x_i^{b_i} \). The algebraic degree is the maximum of \( \sum_{i=1}^{n} b_i \) of all terms with \( a \neq 0 \). For vector Boolean functions the maximum exponent is 1, so the degree is the number of variables in the longest term.

There is a complex relationship between the algebraic degree of a vector Boolean function and the degree of its univariate form. The vector Boolean degree can be obtained from the univariate ANF by finding the binary expansion of its exponents. The binary expansion of a number \( j \) is a vector \((j_0, j_1, \ldots)\) such that \( j = \sum_{s=0}^{2^d-1} j_s 2^s \). See Carlet (2010b) for more detail on this relationship, but for our purpose it suffices to observe that the maximum exponent of a univariate variable \( 2^{d-1} = q - 1 \) coincides with the maximum possible degree, \( d \), of its vector Boolean form.

A linear function has terms in its algebraic normal form containing a single variable with exponent equal to 1, as shown in the second line of Equation 2.4, and an affine function may also have a constant term. These functions may be written with simplified coefficient subscripts as \( h(x) = a_0 + a_1 x_1 + a_2 x_2 + \ldots + a_n x_n \). The univariate form of a linear vector Boolean function, denoted as \( \mathbb{F}_2 \)-linear, may have terms \( ax^{2^j}, j \in \mathcal{N} \). As with the algebraic degree, this relates to the binary decomposition of the exponent, specifically that \( 2^j \) has a single element in its binary decomposition.
Non-linearity  A function which is not affine is *non-linear*, and there are several definitions in use for the degree of non-linearity. A simple measure is the algebraic degree of the vector Boolean function. Affine functions have degree 1, and non-linear functions have degree 2 or more, with a higher degree corresponding to more non-linearity. A more precise definition of the degree of non-linearity is the minimum Hamming distance between the function and all affine functions on the input (Carlet, 2010b).

For cryptographic functions, non-linearity is necessary to counter the *linear attack* (Matsui, 1994). A large set of plaintext – ciphertext pairs is analysed to find linear functions on bits of the plaintext and ciphertext that have a high probability of predicting a bit of the encryption key. This involves constructing a linear model of the non-linear parts of the cipher, so a high degree of non-linearity in the cipher will increase the complexity of the model. Non-linearity is considered to contribute to confusion, so it seems that a high degree of non-linearity may not be necessary for non-cryptographic hash functions.

Resilience  A Boolean function $h$ is known as *t-resilient* if it is balanced and has the property that, when any $t$ bits of the input are arbitrarily fixed, the sub-function of $h$ over all of the remaining $n - t$ free inputs is balanced (Chor et al., 1985). Resilient functions have the property of *correlation immunity*: every $t$-tuple obtained by fixing $t$ elements of the input $x$ is statistically independent of the output $h(x)$ (Siegenthaler, 1984). Resilience is invariant to permutations of the alphabet and bit-flips of the input (Meier and Staffelbach, 1990). The definition of $t$-resilient functions has been extended to functions over finite fields (Camion and Canteaut, 1999), and resilient functions have been characterised in terms of several other linear structures: matrices, Fourier transforms, orthogonal arrays (Gopalakrishnan and Stinson, 1995) and multipermutations (Camion and
Chapter 2. Literature Review

Canteaut, 1999).

Definition 1 (t-resilient function over a finite field). A function \( h : \mathbb{F}_q^n \rightarrow V \), where \( V \) is a finite set, is \( t \)-resilient over \( \mathbb{F}_q \) if, when \( t \) of the inputs are fixed, \( h(x) \) is balanced over the remaining \( n - t \) inputs, for every subset \( \{i_1, \ldots, i_t\} \subseteq \{1, 2, \ldots, n\} \) of \( t \) inputs, taking each value \( x_i \in \mathbb{F}_q \).

Resilient functions are so named as they are resilient to the correlation attack (Siegenthaler, 1984). A typical structure for a cipher, which is known to the attacker, is a set of linear feedback shift registers (LFSR) whose outputs are merged using a combining function. The attacker exhaustively searches through the configurations of \( t \) of the LSFRs. If the combining function has resilience less than \( t \), when the attacker uses the correct LSFR configurations, a correlation will be observed between the plaintext and the ciphertext. In cryptographic functions, resilience is considered to be a source of diffusion (Schnorr and Vaudenay, 1995; Camion and Canteaut, 1999).

From the non-cryptographic point of view, a \( t \)-resilient function hashing a set of biased keys having more than \( t \) bits in common will not generate a uniform hash distribution. Conversely, if the input is biased such that each key pair has \( t \) or fewer bits in common, then each pair of hashes is pairwise independent, and we expect a uniform hash distribution.

The resilience of a function is bounded by its algebraic degree, described by Siegenthaler (1984) for Boolean functions, and generalised to functions on a finite field by Camion and Canteaut (1999).

Theorem 2 (Resilience bound). (Camion and Canteaut, 1999, Corollary 1) Let \( h : \mathbb{F}_q^n \rightarrow \mathbb{F}_q^l \) be a \( t \)-resilient function over \( \mathbb{F}_q \). If \( n \neq l + t \) and \( q^l \neq 2 \), then the
2.1. String Hashing

**total degree** $d$ of its algebraic normal form satisfies

$$d + t \leq (q - 1)n - 1 \quad (2.5)$$

Hence there is a trade-off between the non-linearity and resilience of a function. This agrees with the earlier observation that non-linearity relates primarily to confusion, while resilience relates to diffusion.

Resilient functions can be constructed from orthogonal arrays, or from linear codes. Camion and Canteaut (1999) describe construction of resilient functions over finite fields by composition of smaller (meaning over fewer input variables) resilient functions.

**Completeness and Avalanche** A function is *complete* if every output bit depends on every input bit, and not just on a proper subset of the input bits (Kam and Davida, 1979). This is satisfied if every input variable appears in one or more terms of the algebraic normal form.

The *avalanche effect* was described informally by Feistel (1973) as the phenomenon of a small change to the input key resulting in a large and unpredictable change in the function output. Specifically, complementing a single bit of the key should result in an output with on average $1/2$ of the bits changing, and $1/2$ equal to their original values.

As it is possible for an incomplete function to have good avalanche, completeness and avalanche were combined into a single *strict avalanche criterion (SAC)* for Boolean functions by Webster and Tavares (1986).

**Definition 3** (Strict avalanche criterion). Let $\mathbf{x}$ and $\mathbf{x}_i$ be two $n$-bit binary input vectors such that $\mathbf{x}$ and $\mathbf{x}_i$ differ only in bit $i$, $1 \leq i \leq n$. Let $V_i = h(\mathbf{x}) \oplus h(\mathbf{x}_i)$. 
Function $h$ meets the strict avalanche criterion if the probability that each bit in $V_i$ is equal to 1 is 0.5 over the set of all possible input vectors $x$ and $x_i$. This is true for all values of $i$, assuming a uniform distribution of input bits.

An important property of the SAC is that, for a given set of $V_i$ obtained by complementing a single input bit, the resulting output bits are pairwise independent (Webster and Tavares, 1986).

Several generalisations of avalanche have been proposed. The definition of SAC was extended to multiple bits by Forrée (1990), and to finite fields by Li and Cusick (2007). The propagation criterion is an alternative generalisation of avalanche proposed by Preneel et al. (1991). For our purpose, the key result is that the degree of propagation places a lower bound on the non-linearity of a function (Zhang and Zheng, 1996).

A high degree of avalanche is necessary for a cryptographic function to resist a differential attack. The simplest differential attack is to choose a plaintext $m$, from which ciphertext $c$ is obtained, and form a second plaintext by making a small modification $m + \alpha$ from which ciphertext $c'$ is generated. Then find the bitwise difference of the ciphertexts $c + c' = \beta$. The attack proceeds by finding pairs $(\alpha, \beta)$ such that $\beta$ occurs with a higher probability than random chance (Biham and Shamir, 1991). Non-linear functions are resistant to the differential attack.

We can now describe the type of input bias which influences the uniformity of a non-cryptographic hash function with a certain avalanche degree. Consider a set of keys which is biased such that all key pairs differ in less than $k$ bit positions. A function with avalanche degree $k$ will generate hash pairs which are uncorrelated, and the hash distribution is expected to be uniform.
**Relationship between resilience, avalanche and non-linearity**

Avalanche is considered to contribute primarily to diffusion of a cryptographic function (Meier and Staffelbach, 1990; Zheng and Zhang, 2003; Carlet, 2010a). However the existence of lower bounds on algebraic degree for both avalanche and non-linearity suggests that they are related. On the other hand, we observed that resilience and non-linearity are inversely related. This apparent tension between resilience and avalanche was explicitly described by Zheng and Zhang (2003), who showed that if a $t$-resilient function has avalanche degree $l$, then $t + l \leq n - 2$.

From the perspective of non-cryptographic hash functions, this relation between resilience and avalanche reflects their differing behaviour with biased input data. Recall that a $t$-resilient function is balanced when less than $t$ bits of the keys are fixed. Conversely, a function with avalanche of degree $k$ has uncorrelated values when $k$ or more bits of the keys are fixed. The mutual limitation between resilience and avalanche states that a function does not exist having expected uniformity for both types of biased keys: those with many bits in common, and keys with few. The effect of resilience on hashing of biased data is explored further in Chapter 6.

### 2.2 Non-Universal Hash Functions for Strings

Numerous functions have been used to hash strings. Many of these were designed for the purpose of non-universal, non-cryptographic hashing, while others have been co-opted from other purposes. The cyclic redundancy check (CRC) algorithms such as CRC-32 and Adler32, and cryptographic hash functions such as MD4 and MD5 are common instances. We now briefly describe several of the leading functions, and observe that their design is not directly related to the mathematical properties described in Section 2.1.6, nor were the functions sub-
lected academic peer review before widespread adoption. Instead, the design process follows an engineering approach (Vincenti, 1990) involving incremental improvements, heuristics, empirical testing and informal peer review by users.

Evaluation and selection of hash functions is described in detail in Section 2.2.1, but we briefly note the leading functions here. There are several websites maintaining informal comparisons of string hash functions (Kankowski, 2012; Jenkins, 1997; Mulvey, 2007). Estébanez et al. (2013) recently published a peer-reviewed comparison of many string hash functions, noting three 32-bit hash functions that have good avalanche, collision, and speed properties: MurmurHash2 (Appleby, 2008), lookup3 (Jenkins, 2006) and SuperFastHash (Hsieh, 2008). Of these, SuperFastHash was the fastest, hashing 1000000 1024-byte keys in 177 seconds on a 2.8 GHz Intel CPU, murmurhash2 slightly slower, and lookup3 being only half as fast.

There are two main classes of non-universal string hash algorithms: multiplication hashes which use prime number arithmetic, and shift-add-XOR hashes using only those basic register operations. Estébanez et al. (2013) compared these two classes, and concluded that the functions based on primes generally suffered from reduced avalanche performance. Here we will describe four hash functions. Three of them are shift-add-XOR style hashes that are not included in the Estébanez et al. (2013), while the fourth, FNVhash is included as an example of a hash based on prime number arithmetic. Each of these functions are widely used.

The Fowler, Noll and Vo hash (FNVhash) was proposed in 1991 and is documented in an IETF internet draft (Fowler et al., 2012). The essential elements of the hash function are shown in Algorithm 1. It is a multiplication-style hash using a prime number, and the algorithm contains an implicit modulo 2^b operation as a consequence of performing the computations in a register of length b bits. The FNVprime is carefully selected. The authors suggest that the number of 1
2.2. Non-Universal Hash Functions for Strings

Algorithm 1 FNV-1a hash function

Data: The key, \( S = s_1s_2\ldots s_n \).

1: \( h \leftarrow FNVoffsetBasis \)
2: for each \( i \) from 1 to \( n \) do
3: \( h \leftarrow h \oplus s_i \) \hspace{1cm} \text{(the bit-wise exclusive-or operation)}
4: \( h \leftarrow h \times FNVprime \)
5: return \( h \)

bits should not be too high, and that a small prime does not achieve good mixing for short keys. Mixing is an informal property relating to avalanche: changes in all parts of the key lead to changes in all bits of the hash value. For example, the prime for the 64-bit hash is \( 00000100000001B3 \)\text{hex}. \text{FNVhash} is used in numerous applications database indexes, hash maps in Microsoft Visual C++, hash tables in various network protocol implementations, \text{libmemcached} distributed key–value store by Twitter Inc (Adams, 2010), email spam detection, and several word search systems (Fowler et al., 2013). The \text{hash()} function for strings in python is a variation on this algorithm (it uses a smaller prime, 1 000 003, among other minor modifications). Interestingly, the hash function was revised in response to user testing. The original function had steps 3 and 4 in the reverse order, and the new FNV-1a algorithm was observed to have improved avalanche. It was tested by Estébanez et al. (2013) and found to have comparatively poor avalanche performance.

Jenkins (2013) has been influential, publishing several hash functions, including \text{lookup3}, providing test software and benchmarks that have been widely used and adapted, and maintaining a performance comparison table of numerous hash functions (Jenkins, 1997). His most recent function is \text{SpookyHash} (Jenkins, 2012), a 128-bit shift-add-XOR style hash, motivated by the speed improvement obtained by fully utilising 64-bit registers. The inner loop C++ code is shown in Figure 2.1. It consumes 3 bytes of input per cycle, obtaining a significant speed
Mix(const uint64 *data) 
{
    s0 += data[0]; s2 ^= s10; s11 ^= s0; s0 = Rot64(s0,11); s11 += s1;
    s1 += data[1]; s3 ^= s11; s0 ^= s1; s1 = Rot64(s1,32); s0 += s2;
    s2 += data[2]; s4 ^= s0; s1 ^= s2; s2 = Rot64(s2,43); s1 += s3;
    s3 += data[3]; s5 ^= s1; s2 ^= s3; s3 = Rot64(s3,31); s2 += s4;
    s4 += data[4]; s6 ^= s2; s3 ^= s4; s4 = Rot64(s4,17); s3 += s5;
    s5 += data[5]; s7 ^= s3; s4 ^= s5; s5 = Rot64(s5,28); s4 += s6;
    s6 += data[6]; s8 ^= s4; s5 ^= s6; s6 = Rot64(s6,39); s5 += s7;
    s7 += data[7]; s9 ^= s5; s6 ^= s7; s7 = Rot64(s7,57); s6 += s8;
    s8 += data[8]; s10 ^= s6; s7 ^= s8; s8 = Rot64(s8,55); s7 += s9;
    s9 += data[9]; s11 ^= s7; s8 ^= s9; s9 = Rot64(s9,54); s8 += s10;
    s10 += data[10]; s0 ^= s8; s9 ^= s10; s10 = Rot64(s10,22); s9 += s11;
    s11 += data[11]; s1 ^= s9; s10 ^= s11; s11 = Rot64(s11,46); s10 += s0;
}

Figure 2.1: The 128-bit SpookyHash inner loop, from the file SpookyV2.h (Jenkins, 2012) It is a carefully chosen sequence of shift, add and XOR operations in 64-bit registers.

improvement compared to hash functions that use 1 byte of input per iteration of their inner loop, such as FNVhash. The core operation of Spookyhash performs a sequence of additions, exclusive-ors and rotations on the state which is maintained in 12 64-bit registers. It avoids a multiplication operation, which is slower than other operations on most processors. The sequence of rotations has been carefully tuned to achieve good collision and avalanche performance.

Appleby (2008) also made a sustained endeavour to develop non-universal, non-cryptographic hash functions, creating murmurhash2 and murmurhash3, which are used for Bloom filters in the Apache hadoop\(^1\) distributed application framework, and the Apache hbase distributed no-SQL database. This effort has also produced the SMHasher suite of tests for non-cryptographic hash functions, including tests for speed, avalanche, and collisions, which is described in Section 2.2.1. The C++ code for the MurmurHash3 inner loop is shown in Figure 2.2. It is a hybrid of both the multiplication and shift-add-XOR styles. The input is

---
\(^1\)http://hadoop.apache.org/
const uint64_t * blocks = (const uint64_t *)(data);
for(int i = 0; i < n64blocks; i++) {
    uint64_t k1 = getblock64(blocks,i*2+0); // get next 64-bit block
    uint64_t k2 = getblock64(blocks,i*2+1); // get next 64-bit block
    k1 *= 0x87c37b91114253d5; k1 = ROTL64(k1,31);
    k1 *= 0x4cf5ad432745937f;
    h1 ^= k1; h1 = ROTL64(h1,27);
    h1 += h2; h1 = h1*5+0x52dce729;
    k2 *= 0x4cf5ad432745937f; k2 = ROTL64(k2,33);
    k2 *= 0x87c37b91114253d5;
    h2 ^= k2; h2 = ROTL64(h2,31);
    h2 += h1; h2 = h2*5+0x38495ab5;
}

// h2 and h1 contain the 128-bit hash

**Figure 2.2:** The inner loop of the 64-bit version of MurmurHash3, from the file MurmurHash3.cpp (Appleby, 2008). The function is a sequence of multiplication, addition and exclusive-or operations.

read in blocks of 128 bits, and a sequence of multiply, rotation, exclusive-or and addition operations are applied. The function uses several large constants that have been tuned to achieve good collision and avalanche performance.

The development of MurmurHash3 is a clear example of the evolutionary nature of the design process. An earlier version (MurmurHash2) was released but found by a user to have poor performance with keys that have a certain periodic structure. Appleby responded by redesigning the hash, released as MurmurHash3 and adding a new test to the SMHasher test suite.

CityHash is a new hash published by Google Inc. (2011). Its development was strongly influenced by MurmurHash and achieves increased speed by using the built-in CRC instruction on some new Intel CPUs, as well as using multiple independent mathematical operations at each step to exploit multiple CPU cores. Google use it extensively in hash maps. The code is not shown here as it is more complex than the previous functions, and not instructive for the present discussion. The speed of these hash functions depends greatly on the hardware.
Chapter 2. Literature Review

CityHash is clearly faster if the CPU possesses the CRC instruction, but has similar speed when the instruction is absent and may in fact be slower than Spookyhash on short keys (Jenkins, 2012)

In summary, we can see that the design of non-universal, non-cryptographic hash functions follows an engineering approach. The functions meet several competing design goals (speed, avalanche, low collisions on expected data) that are evaluated by testing. The documented examples of function modification in response to feedback from users, as well as blog and email list entries by the designers, demonstrate that design progresses incrementally in response to outcomes from repeated testing and operational performance. The design loop of tune-and-test, or trial-and-error, results in an evolutionary design cycle.

A key observation is that the hash function designs are not mathematically derived to have desired properties such as avalanche degree or resilience. Nor are the functions analysed to confirm that they possess the desired properties. There is a disconnection between the theoretical results and design practice. We suggest several potential explanations. First, it is difficult to translate a hash algorithm into a mathematical expression of the function, such as algebraic normal form, on which to perform further analysis. The ANF has $O(q^n)$ terms and large functions will be intractable. Second, there is no analytic test to determine the degree of avalanche or resilience of a function, only bounds based on the algebraic degree, as described in Section 2.1.6.

It can be seen that the design of hash function is as much an art as a science. This is reflected in the fact that the designers of these functions have not published an analysis of their designs for peer-review, nor provide detailed explanations of their design decisions in the documentation or source code. But they do publish the source code with open source licences and solicit users to perform tests and make suggestions. From the user’s perspective, in the absence of mathematical
guarantees, thorough empirical testing is the most reliable means of evaluating the functions.

2.2.1 Testing Hash Functions

Non-universal, non-cryptographic hash functions are validated using an engineering approach. *Black-box tests*, which observe the function response to carefully selected inputs, are used to obtain an estimate of the important function properties. This also allows direct comparison of hash functions to identify the best function, at least in the environment of the tests.

There is no standardised or formally codified suite of tests for non-cryptographic hash functions, in contrast to cryptographic functions, for which there is a standardised testing regime (Rukhin et al., 2010). Nonetheless, a common set of tests has evolved over time that appears to be generally accepted. The development of these tests has partly been informal in the sense that some tests have not been the subject of academic peer review and analysis. Rather, hash testing has largely been situated in on-line conversations, and on web pages of test authors and hash function developers, in a manner similar to open source software development. Noteworthy examples are Jenkins (1997), Mulvey (2007), Appleby (2008), and Hsieh (2008).

The tests are of two main kinds: collision tests and avalanche tests. In this thesis, we are principally interested in collision tests as they directly measure hash function uniformity. These are investigated in Chapters 5 and 6. Briefly, avalanche tests involve selecting a sample of keys, then hashing all of the keys differing from the original key sample by one bit. The results are displayed in an *avalanche matrix* where the correlation between each key bit and value bit is encoded using a colour range so that bit combinations with high correlations
are easily identified. Avalanche testing and avalanche matrices are described in
detail in Mulvey (2007) and Appleby (2008), and presented in a peer-reviewed
publication in Estébanez et al. (2013).

**Theoretical collision results** A hash collision test involves hashing $N$ keys
followed by analysis of the distribution of the resulting hashes over the $M$ possible
hash values. Essentially, the keys are inserted into a hash table of size $M$. The
observed distribution is compared to the distribution expected from an ideal hash
function. Mathematically, this is known as the *balls in bins* problem, where a
ball (a key) is randomly assigned (the hash function) into a bin (a hash value).

One commonly used statistic for characterising the distribution of balls into
bins is the *number of collisions*. This can be divided by the number of bins to give
the *collision rate*. Assuming each ball has a uniform probability of being assigned
to each bin, the expected number of collisions is $M - N + N \left(1 - 1/N\right)^M$, which
can be easily derived from the expected number of empty bins $N \left(1 - 1/N\right)^M$.
Note that this is not a useful measure when $N \gg M$ as once all of the bins
have at least one ball, each subsequent ball must be a collision and the number
of collisions provides no additional information about the uniformity of the hash
distribution.

Another measure of the theoretical distribution of collisions is the number
of balls in the fullest bin. In the hashing literature, a related measure is used,
the *length of the longest probe sequence (LLPS)* in a separately chained hash
table. The longest probe sequence involves an initial read of the hash table
to retrieve the address of the linked list of keys with the requested hash value,
then further reads of each node in the list to retrieve the last element; thus
LLPS = 1 + balls in the fullest bin.

Ramakrishna (1988) proposed a recurrent expression providing an exact so-
2.2. Non-Universal Hash Functions for Strings

olution for the expected value of LLPS. Reviriego et al. (2011) proposed an exact analytical expression for the LLPS distribution, providing both the expected value and the variance. We do not give the expression here due to its length. The calculation has several steps involving combinatoric sums, and is costly to perform, only being practical for small numbers (< 1000) of balls and bins. Gonnet (1981) gives an approximation for the expected value of LLPS for large hash tables, which was described earlier in Section 2.1.5.

Published collision test methodologies The first peer reviewed investigation of string hashing performance was by Ramakrishna and Zobel (1997). They fill a hash table of size $M$ with $N = 1000$ keys and measured the average number of keys hashing to each value, which has an expected value of $N/M$ for an ideal hash function, and the LLPS. They varied $M$ to achieve several hash table load factors. Presciently, they hypothesised that shift-add-XOR style functions are superior to functions based on multiplication, even though they did not consider avalanche as a desirable property. They were particularly interested in testing the universality of functions, and performed empirical tests over a subset of the seed space. They performed a large number of trials, (for example, they tested 10,000 seeds in the universality tests) and presented the mean and standard deviation of the measurements along with the theoretical expected values, allowing an assessment of the significance of each result in isolation as well as comparatively.

Henke et al. (2008) reviewed a range of cryptographic and non-cryptographic hash functions for rapid selection of network packets. They used a $\chi^2$-test to evaluate the independence of the hash values from categorical variables relating to the input packets: network protocol, packet length ranges, and particular values at specific positions in the highly structured headers of network packets. Note that these tests may not detect a biased function if the bias is such that
the hashes are uniformly distributed among the tested categories. Nonetheless, the approach could be modified to fully test for randomness by comparison with the distribution of a statistic representing the un-grouped collision rate, such as LLPS, if its distribution was known.

Estébanez et al. (2013) evaluated collision tests using two measures. The Bhattacharyya distance, $B_d$, measures the similarity between two distributions, in this case being the observed distribution of hash values and the ideal uniform distribution. Samples from distribution $a$ and $b$ are partitioned into $M$ bins, then $B_d(a, b) = -\ln(\sum_{i=1}^{M} \sqrt{a_i b_i})$, where $a_i, b_i$ are the counts of the samples from each distribution $a, b$ in the $i$-th bin (Bhattacharyya, 1943). $B_d$ ranges from 0 for identical distributions to $\infty$ for distributions having no overlap. Estébanez et al. (2013) only used the Bhattacharyya distance for comparison between hash functions, and did not consider the statistical significance of the results. The second measure used by Estébanez et al. (2013) was the collision rate, the ratio of the number of collisions to the number of hashes. Again, the measure was only used for comparison of hash functions, and the statistical significance of the results was not discussed. They tested a single set of 1000 or 2000 keys (depending on the data set used), which were inserted into hash tables with load factors of 0.5 and 2.0.

The most recent of the informal hash tests is the SMHasher test suite (Appleby, 2008), which was developed in conjunction with the development of Murmurhash, and subsequently contributed to the development of CityHash (Google Inc., 2011). SMHasher was informed by the earlier work of Jenkins (1997). The SMHasher collision test fails if the number of collisions is more than twice the expected number. The statistical significance of this outcome is not described, but is rather used as an imprecise indication of an obviously bad outcome. The collision test is applied to several key sets that are chosen to probe specific aspects
of the hash function, as well as any additional key sets provided by the user:

- **Differential test.** A set of keys is created from a randomly selected key by modifying a small number of its bits. The test fails if 3 or more keys collide. The number of “differential” keys increases for longer keys, so only a subset of the differentials is tested. For a 64-bit key, SMHasher tests all 5-bit differentials, all 4-bit differentials for a 128-bit key, and all 3-bit differentials for a 256-bit key.

- **Cyclic key test.** A key set is created (10 million keys) with each key being $N$ repetitions of $M$ bits. There will be a higher than expected collision rate if stages of the hash function cancel each other out. This test was created by Appleby to show that MurmurHash2 has this defect.

- **Sparse key test.** The key set contains all $N$-bit keys with only $M$-bits set, $N \gg M$. This test will expose a function which does not provide good mixing of 0-bits.

- **Permuted key test.** Ten small blocks of data are created, which are concatenated to create a key. The key set consists of all 3.6 million permutations of these blocks. A higher number of collisions will be observed if the hash function contains duplicate sub-functions for different input blocks.

- **Zero test.** The key set contains strings of zeroes of different lengths. These should hash to different values.

- **Seed test.** If the hash function uses a seed value, a single key is hashed with a set of different seeds.

In summary, while a common procedure for performing collision tests has emerged, evaluation of the results is *ad hoc*. Surprisingly little consideration has
been given to the statistical significance of the test results and the related question of whether enough testing has been performed to support any conclusions inferred from the test. We propose a statistically rigorous test methodology in Chapter 5.

Nonetheless, current practice appears to detect poor hash functions with high reliability. Evaluation of test outcomes typically involves comparison between several functions undergoing the same test, and, if one or more of the functions is good, then a poor function will be readily identified. While the significance of the tests is not known, any uncertainty or ambiguity arising from conflicting results is usually resolved by performing more tests. It follows that we can have reasonable confidence that the hash functions developed using extensive testing have good randomness properties, and the many tests and observations of operational performance reported by numerous users is effective at identifying problematic functions in the long run.

### 2.3 Recursive Hash Functions

*Recursive hash functions* for strings, also known as *rolling hash functions*, allow the efficient calculation of a hash on a window sliding over a string. They are an essential element of the Karp-Rabin string search algorithm described later. The hash is calculated using the hash calculated from the previous position of the sliding window: hash, $v_i$, of the substring starting at position $i$ and of length $n$ is $\text{hash} = h(s_{i-1}, s_{i+n}, v_{i-1})$ where $v_{i-1}$ is hash of the preceding window, $s_{i-1}$ is the symbol at the start of the preceding window and $s_{i+n}$ is the symbol at the end of the current window. Observe that the symbols $s_i$ to $s_{i+n-1}$ occur in both the current and the preceding window, and their contribution to the hash value is included in $v_{i-1}$. This allows the hashes of succeeding substrings to be calculated from the preceding hash using very few operations.
Karp-Rabin Rolling Hash Function

The Karp and Rabin (1987) rolling hash algorithm, based on division by a prime number, is widely used. A substring $u$ of length $n$ of a string $S$ can be expressed as a number $v'$ in base $q$, the alphabet size. Typically $q = 256$ so that the input string is interpreted as a sequence of bytes. For a substring starting at position $i$,

$$v' = q^0 s_{i+n-1} + q^1 s_{i+n-2} + q^2 s_{i+n-3} + \cdots + q^{n-1} s_i. \quad (2.6)$$

The integer $v'$ is converted to a hash $v$ by taking the modulus with respect to a large prime number $p$,

$$h(u) = v \equiv v' \pmod{p}. \quad (2.7)$$

Gonnet and Baeza-Yates (1990) observed that the prime should be chosen so that $q$ is a primitive root modulo $p$, to ensure that the hash function has maximum cycle. Specifically, this means that

$$\min_k (q^k \equiv 1 \pmod{p}) = p - 1. \quad (2.8)$$

The function $h$ can be calculated using fixed-width integer arithmetic by $n$ iterations of the recurrence relation

$$v_j \equiv (qv_{j-1} + s_{i+j-1}) \pmod{p}, \quad (2.9)$$

with $j = 1, 2, \ldots, n$ and $v_0 = 0$, provided that the integer $q(p+1)$ does not overflow the register. The final hash value $v$ of Equation 2.7 is $v = v_n$. The modulo operation after each iteration prevents the calculation from overflowing, although the $n$ modulo operations are time-consuming. For example, with an 8-bit alphabet ($q = 256$) a 54-bit hash can be calculated using 64-bit signed integer
arithmetic.

The advantage of the rolling hash method is that, once the initial hash has been calculated, the hash of the next string $u_{i+1}$ can be efficiently calculated from $h(u_i)$ with the rolling hash equation

$$h(u_{i+1}) \equiv (s_{i+n} + qh(u_i) - q^n s_i) \pmod{p}.$$  \hspace{1cm} (2.10)

The term $q^n \pmod{p}$ can be pre-calculated, and the calculation of $h(u_{i+1})$ requires a single modulo operation, a substantial speed-up on the $n$ modulo operations required to calculate the initial hash. Gonnet and Baeza-Yates (1990) proposed a more efficient variation of the algorithm where the radix is a prime and the modulo is to a power of 2, which is chosen to be the word size so that the modulo happens as a natural result of multiplication in a fixed size register. Again, the prime is carefully chosen to ensure the hash function has maximum cycle.

**Hashing by General Polynomial Division**

Cohen (1997) describes several recursive hash functions where the input strings are mapped to polynomials over a finite field. One of these methods, *hashing by general polynomial division*, is described in detail below. It was shown by Lemire and Kaser (2010) to be pairwise independent, and has several advantages compared to the Karp-Rabin hashing algorithm. The calculation is more efficient, requiring only bitwise shift and exclusive-or operations, while Karp-Rabin requires integer multiplication. Also, the range of hash values is conveniently a power of 2, rather than a prime number. It is described in detail here as it is used extensively in later chapters.

Conceptually, the hash function is similar to the Karp-Rabin function: the
Algorithm 2 Non-recursive calculation of hash by general polynomial division implementing Equation 2.12.

**Data:** $S = s_0s_1 \ldots$, the input string.
- $n$, the length of the key.
- $i$, the position of the first symbol of the key in $S$.

**Parameters:** $p$, the irreducible polynomial of degree $d$, expressed as a binary number.

1: $v' \leftarrow 0$ the hash state variable.
2: for each symbol $s_j$, $j = i$ to $i + n - 1$ do
3:    $v' \leftarrow v' \ll 1$ left shift 1 bit.
4:    if bit $d$ of $v' = 1$ then
5:       $v' \leftarrow v' \oplus p$ bit-wise exclusive-or with $p$.
6:    $v' \leftarrow v' \oplus T(s_j)$ bit-wise exclusive-or with next symbol.
7: $h(S[i, i + n - 1]) \leftarrow v'$ return the hash value.

input string is converted to a large number represented as an element of a finite field, rather than as an integer, to which a modulo operation is applied where the divisor is an irreducible polynomial in a finite field rather than a prime number.

Let the input key be the substring $u_i = s_is_{i+1} \ldots s_{i+n-1}$ of $n$ consecutive input symbols beginning at position $i$ of the input string. The input key is mapped to $v'_i \in F_2[x]$, the field of polynomials in the indeterminate variable $x$ having coefficients that are elements of $F_2$, the finite field with two elements, $\{0, 1\}$, also known as Galois field GF(2),

$$v'_i = \sum_{j=1}^{n} x^{n-j} T(s_{i+j-1}). \quad (2.11)$$

The hash function $h$ reduces the polynomial $v'_i$ modulo an irreducible polynomial $p(x)$ so that the range of the hash function is the ring $R = F_2[x]/\langle p(x) \rangle$. The function $T$ maps the input symbols into $R$, and it is recommended that the mapping is random to avoid bunching of hash values at low values, especially for
Algorithm 3 Recursive calculation of hash by general polynomial division implementing Equation 2.13.

Data: \( S = s_0s_1 \ldots \), the input string.
\( n \), the length of the key.
\( i \), the position of the first symbol of the key in \( S \).

Parameters: \( p \), the irreducible polynomial of degree \( d \), expressed as a binary number.
\( h(S[i - 1, i + n - 2]) \), the hash of the preceding key.

1: \( v' \leftarrow h(S[i - 1, i + n - 2]) \) the hash state variable.
2: \( v' \leftarrow v' \ll 1 \) left shift 1 bit.
3: if bit \( d \) of \( v' \) = 1 then
4: \( v' \leftarrow v' \oplus p \) bit-wise exclusive-or with \( p \).
5: \( v' \leftarrow v' \oplus T(s_{i+n-1}) \) bit-wise exclusive-or with new symbol.
6: \( v' \leftarrow v' \oplus x^nT(s_{i-1}) \) bit-wise exclusive-or with preceding symbol.
7: \( h(S[i, i + n - 1]) \leftarrow v' \) return the hash value.

short keys.

\[
h(u_i) = v_i = \sum_{j=1}^{n} x^{n-j}T(s_{i+j-1}) \pmod{p(x)}.
\]

(2.12)

The order of \( R = 2^d \) is the size of the hash table, and \( d \) is the degree of \( p(x) \), which is the length of the hash value in bits. As \( R \) is an integral domain the ring \( R \) is cyclic and the distribution of keys to elements of \( R \) is balanced. Finally, the hash value \( v \in R \) is converted to an integer by simply interpreting its polynomial coefficients as a binary number, for example \( x^4 + x + 1 \) becomes 10011.

Example As a running example to illustrate this algorithm, consider hashing the string gram using an alphabet size \( q = 256 = 2^8 \). Furthermore, assume that the hash value is 8 bits long, so all of our calculations are performed in \( \mathbb{F}_{2^8} \). The elements of the field are expressed as polynomials in an indeterminate variable \( x \) with coefficients from \( \mathbb{F}_2 = \{0, 1\} \). We describe \( x \) as indeterminate as we never solve the polynomial for \( x \). Rather, the polynomials are added and multiplied to obtain new elements of \( \mathbb{F}_{2^8} \). These operations follow the arithmetic rules of \( \mathbb{F}_2 \), where addition is the bitwise exclusive-OR operation, multiplication is the
bit-wise AND operation. So \( x^3 + x + 1 \) and \( x^3 + x^2 + 1 \) are elements of \( \mathbb{F}_{2^8} \), and their sum \( x^3 + x^3 + x^2 + x + 1 + 1 = x^2 + x \) is also an element. There is a direct mapping between the polynomial coefficients and binary numbers, such as the number \( 1010011011 \leftrightarrow x^9 + x^7 + x^4 + x^3 + x^1 + x^0 \). Observe that this binary number is longer than our 8-bit hash values, and the polynomial is not an element of \( \mathbb{F}_{2^8} \) because of the \( x^9 \) term. To convert it to \( \mathbb{F}_{2^8} \), we divide by an irreducible polynomial of order 8, \( p(x) = x^8 + x^6 + x^5 + x^2 + 1 \leftrightarrow 101100101 \). Irreducible polynomials cannot be factored, and are analogous to prime numbers in the field of integers. For example \( x^5 + x + 1 = (x^2 + x + 1)(x^3 + x^2 + 1) \) is reducible.

Next, we assume that \( T(s_i) \) is the ASCII value of \( s_i \), so that \( T(g') = 67_{\text{hex}} = 10100111 \leftrightarrow x^6 + x^5 + x^2 + x + 1, T(r') = 72_{\text{hex}}, T(a') = 61_{\text{hex}}, \) and \( T(m') = 6d_{\text{hex}}. \) Then, applying Equation 2.12,

\[
\begin{align*}
h'(\text{gram}') &= \sum_{j=1}^{4} x^{4-j} T(s_{j-1}) \pmod{p(x)} \\
&= x^3(x^6 + x^5 + x^2 + x + 1) + x^2(x^6 + x^5 + x^4 + x) \\
&\quad + x(x^6 + x^5 + 1) + (x^6 + x^5 + x^3 + x^2 + 1) \pmod{p(x)} \\
&= x^9 + x^8 + x^5 + x^4 + x^3 + x^8 + x^7 + x^6 + x^3 \\
&\quad + x^7 + x^6 + x + x^6 + x^5 + x^3 + x^2 + 1 \pmod{p(x)} \\
&= x^9 + x^6 + x^4 + x^3 + x^2 + x + 1 \pmod{p(x)} 
\end{align*}
\]

Now, dividing by \( p(x) \), we find that

\[
\begin{align*}
&= x^2 p(x) + x^7 + x^4 + x^2 + 1 \pmod{p(x)} \\
&= x^7 + x^4 + x^2 + 1 \\
&\rightarrow 10010101,
\end{align*}
\]
which is our hash value.

Once the initial hash has been calculated, the previous hash value can be used to calculate the hash of the next key using the rolling hash function

$$h(u_i) = xh(u_{i-1}) + T(s_{i+n-1}) + x^nT(s_{i-1}) \pmod{p(x)}$$ \hfill (2.13)

Calculation of $xh(u_{i-1})$ requires a small number of bit-wise operations. Let $p(x) = x^d + p_{d-1}x^{d-1} + p_{d-2}x^{d-2} + \cdots + p_0$ and as $p(x) = 0$ we can write $x^d = p_{d-1}x^{d-1} + p_{d-2}x^{d-2} + \cdots + p_0$. For a polynomial $w(x) \in R$ where $w(x) = w_{d-1}x^{d-1} + w_{d-2}x^{d-2} + \cdots + w_0$,

$$xw(x) = w_{d-1}x^d + w_{d-2}x^{d-1} + \cdots + w_0x \hfill (2.14)$$

$$= \begin{cases} w_{d-2}x^{d-1} + w_{d-3}x^{d-2} + \cdots + w_0x, & w_{d-1} = 0 \\ (w_{d-2} + p_{d-1})x^{d-1} + (w_{d-3} + p_{d-2})x^{d-2} + \cdots + p_0, & w_{d-1} = 1 \end{cases}$$ \hfill (2.15)

Hence, when representing $p(x)$ and $w(x)$ as binary numbers $p$ and $w$, finding $w(x) \leftarrow xw(x)$ is achieved as follows:

1. Shift $q$ left without carry. Shift in 0 on the right hand side, so that $w_0 = 0$.

2. Test $w_d$. (was $w_{d-1}$ before the left shift)

3. If $w_d = 1$, $w = w \oplus p$. ($\oplus$ is the exclusive-or operation)

$T(s_{i+n-1})$ and $x^nT(s_{i-1})$ can be precalculated and implemented in look-up tables of $q$ elements. Addition of the terms in the recursion is performed by their bit-wise exclusive-or. Note that the result of this addition will be an element of $R$, and there is no need to perform an additional operation for the modulo
2.3. Recursive Hash Functions

$p(x)$. Algorithms 2 and 3 show the complete calculation of $h(u)$ using both the recursive and non-recursive methods.

**Example (continued)** We calculate the hash from the previous example using the non-recursive Algorithm 2.

<table>
<thead>
<tr>
<th>iteration</th>
<th>operation</th>
<th>result in register</th>
</tr>
</thead>
<tbody>
<tr>
<td>(1)</td>
<td>$v' = 0$</td>
<td>0000 0000</td>
</tr>
<tr>
<td></td>
<td>$v' = T('g') = 67_{\text{hex}}$</td>
<td>0110 0111</td>
</tr>
<tr>
<td>(2)</td>
<td>$v' = v' \ll 1$</td>
<td>1100 1110</td>
</tr>
<tr>
<td></td>
<td>$T('r') = 72_{\text{hex}}$</td>
<td>0111 0010</td>
</tr>
<tr>
<td></td>
<td>$v' = v' \oplus T('r')$</td>
<td>1011 1100</td>
</tr>
<tr>
<td>(3)</td>
<td>$v' = v' \ll 1$</td>
<td>1 0111 1000</td>
</tr>
<tr>
<td></td>
<td>$p = 1$</td>
<td>0110 0101</td>
</tr>
<tr>
<td></td>
<td>$v' = v' \oplus p$</td>
<td>0001 1101</td>
</tr>
<tr>
<td></td>
<td>$T('a') = 61_{\text{hex}}$</td>
<td>0110 0001</td>
</tr>
<tr>
<td></td>
<td>$v' = v' \oplus T('a')$</td>
<td>0111 1100</td>
</tr>
<tr>
<td>(4)</td>
<td>$v' = v' \ll 1$</td>
<td>1111 1000</td>
</tr>
<tr>
<td></td>
<td>$T('m') = 6d_{\text{hex}}$</td>
<td>0110 1101</td>
</tr>
<tr>
<td></td>
<td>$v' = v' \oplus T('m')$</td>
<td>1001 0101</td>
</tr>
</tbody>
</table>

and the final hash value, 1001 0101, is the same as the previous calculation.

Now, for the recursive calculation in Algorithm 3, we assume that the preceding character was ‘n’ (the entire string is then ngram), so that $T('n') = 6e_{\text{hex}}$, and that we have already calculated $x^nT('n') = be_{\text{hex}}$. Also, the previous hash $h('\text{ngra}') = a3_{\text{hex}}$.

<table>
<thead>
<tr>
<th>algorithm line number</th>
<th>operation</th>
<th>result in register</th>
</tr>
</thead>
<tbody>
<tr>
<td>(1)</td>
<td>$v' = h('\text{ngra}')$</td>
<td>1010 0011</td>
</tr>
<tr>
<td>(2)</td>
<td>$v' = v' \ll 1$</td>
<td>1 0100 0110</td>
</tr>
<tr>
<td></td>
<td>$p = 1$</td>
<td>0110 0101</td>
</tr>
<tr>
<td>(4)</td>
<td>$v' = v' \oplus p$</td>
<td>0010 0011</td>
</tr>
<tr>
<td></td>
<td>$T(s_5) = 0x6d$</td>
<td>0110 1101</td>
</tr>
<tr>
<td>(6)</td>
<td>$v' = v' \oplus T(s_5)$</td>
<td>0100 1110</td>
</tr>
<tr>
<td></td>
<td>$x^nT(s_1)$</td>
<td>1101 1011</td>
</tr>
<tr>
<td>(7)</td>
<td>$v' = v' \oplus x^nT(s_1)$</td>
<td>1001 0101</td>
</tr>
</tbody>
</table>

As before, the hash value is 1001 0101. □
Calculation of the recursion requires at most five bit-wise operations (a left shift, a bit test, an exclusive-OR operation if \( q_{d-1} = 1 \), and two more exclusive-OR operations to add the second and third terms) and two table lookups. On most processors, this will be faster than calculation of the Karp-Rabin recursive hash which requires an integer multiplication, along with several other operations.

## 2.4 String Search

String search is the problem of finding the occurrences of a substring in a longer string. Formally, the search is for a string of length \( N \), called the pattern or target, \( Q = q_1 q_2 \ldots q_N \), within a longer string of length \( M \) called the text or reference string, \( R = r_1 r_2 \ldots r_M \). Two strings are said to match or to be identical if they have the same symbols at each position. A pattern \( Q \) matching a substring of \( R \) starting at position \( k \) has \( q_1 = r_k, q_2 = r_{k+1}, \ldots, q_N = r_{k+N-1} \). The cost of string comparison is \( O(N) \) as each symbol in \( Q \) must be inspected.

### 2.4.1 Fundamental Algorithms for String Search

Fundamental string search techniques are briefly reviewed, describing their performance and limitations, to explain why the Karp-Rabin string search algorithm, which is reviewed in the next section, is preferred for our application. Detailed descriptions of many of these algorithms can be found in string processing texts such as Crochemore and Rytter (2002) or Navarro and Raffinot (2002).

A brute force string search algorithm takes \( O(MN) \) time. The pattern \( Q \) is compared with the \( M - N \) substrings of \( R \) of length \( N \), and each string comparison is \( O(N) \). Note that each position in \( R \) is compared up to \( N \) times.

There are several algorithms that complete the search in linear time by avoiding multiple comparisons at each position in \( R \). They are used for one-time
searches of a pattern in a text, and achieve their improved efficiency by pre-
processing the pattern.

The Knuth-Morris-Pratt algorithm (Knuth et al., 1977) requires $O(N + M)$
comparisons using $O(N)$ additional space for an array recording the length of
each prefix of $Q$ matching the suffix of the substring ending at each position in
$Q$. As the text is processed left to right, the algorithm can be used for on-line,
or stream processing.

The Boyer-Moore algorithm, with improvements by Galil (1979), also requires
$O(N + M)$ time. It takes $O(N)$ time to pre-process the pattern, then makes
$O(M)$ comparisons in the text. At a location in the text, it performs the string
comparison from right to left, then skips a number of positions following a partial
match. The average search time is $O((M \log N)/N)$, and in practice it is generally
faster than the Knuth-Morris-Pratt algorithm. It is especially suitable for long
patterns and large alphabets. A variant is the Horspool (1980) algorithm, which
uses more space for the pre-processed pattern to achieve a lower constant factor
for time in the average case, although the worst case search time is $O(NM)$.

The Aho and Corasick (1975) algorithm searches for multiple patterns in a
single a pass of the text in $O(M + N)$ time, where $N$ is total length of the pat-
terns. The pre-processing stage involves construction of a finite state automaton
encoding the dictionary of patterns. Unlike the previous algorithms, the space
needed to represent the multiple patterns is reduced when they have substantial
similarity.

The usefulness of these algorithms for collection-based compression is limited
as they need to read the entire data collection for each search. To reduce the
search time it is necessary to create data structures that describe the text. Various
alternatives are discussed in the next sections.
Algorithm 4 The Karp-Rabin string search algorithm

**Data:** $Q$, the pattern string of length $N$.

$R$, the text string of length $M$.

**Result:** The positions of the occurrences of $Q$ in $R$.

1: $p \leftarrow$ a randomly chosen prime from the range $1, \ldots, NM^2$, selecting function $\text{FINGERPRINT}(\cdot)$ from the family of hash functions.
2: $h_Q \leftarrow \text{FINGERPRINT}(Q)$.
   The fingerprint of the pattern, $Q$.
3: $L$ is an empty list of matches.
4: for $i = 1$ to $M - N + 1$ do
5:   if $\text{FINGERPRINT}(R[i, i+N-1]) = h_Q$ then
6:     if $Q = R[i..i+N-1]$ by symbol-wise comparison. then
7:       append $i$ to $L$.
8: return $L$.

2.4.2 The Karp-Rabin Algorithm

An alternative method for string search is to calculate a hash of the pattern, and hashes of the length $N$ substrings of the text, then compare the hashes. As hashes typically fit into a single 32-bit or 64-bit word, the comparison is performed in a single operation. This approach is usually inferior to the best methods described in the previous section as the hashing operation is performed at every position in the text, and symbol-wise string comparisons are still needed when the hashes match. However, it is the preferred method when searching for multiple patterns of the same length, as the pattern hashes can be compared in a single hash table query.

Karp and Rabin (1987) described three algorithms using this string search method. The basic algorithm (algorithm 2 in their paper) is shown in Algorithm 4. After first calculating the hash of the pattern $Q$, the length $N$ substrings of the text $R$ are hashed using the Karp-Rabin rolling hash function, or any of the recursive hash functions described in Section 2.3. If the hashes differ, the
algorithm moves to the next position in $R$, and, if the hashes match, a collision resolution step is performed by directly comparing the substring in $R$ with $Q$. It is possible, although extremely unlikely, when using a given hash function that every position in $R$ results in a hash collision, resulting in a worst-case running time of $O(NM)$. Karp and Rabin showed that by randomly selecting the hash function from a family, there is a negligible probability of a run of collisions and the expected worst case running time is $O(N + M)$. For the family of Karp-Rabin rolling hash functions, each member uses a different prime number, which is selected on line 1 of the algorithm. When searching for multiple patterns, $N$ is the combined length of the patterns.

It is a common practice to omit the prime selection step in many domains, such as delta-encoding, content-defined chunking, plagiarism and similarity detection which are described in later sections. This is also the case for collection-based compression, for similar reasons that universal hashing is not suitable: it is impractical to re-hash the collection with a new function.

An interesting variation of Algorithm 4 considered by Karp and Rabin (1987) (their first algorithm) uses multiple hash functions. A string matching collision occurs when all $k$ functions generate a hash collision, which has greatly diminished probability as $k$ increases. For a large $k$, the probability of error can be disregarded and the string match is accepted without performing the symbol-wise string comparison. This is an example of the compare-by-hash technique, which is widely implemented using cryptographic hash functions in systems such as rsync (Tridgell and Mackerras, 1996).

The Karp-Rabin algorithm uses $O(N)$ space, needing to store the patterns $Q$, the pattern hashes, the current substring in $R$ of length $N$ and its hash. Compared to the basic string search algorithms, it is especially well suited to searching for multiple patterns, and for longer patterns on small alphabets.
2.4.3 Fingerprint Indexes for String Search

In order to avoid re-reading the text for each search, a fingerprint index can be maintained, which stores hashes of selected substrings of the text. We refer to these hashes as fingerprints in order to avoid confusion with other hash values that may be used in the index data structure. A typical such structure is a hash table in memory in which the fingerprints are keys which may then be hashed a second time into a reduced range according to the table size. This allows the fingerprint to be longer, such as 32 or 64 bits, without requiring a hash table of that size.

We first describe the approach of Bentley and McIlroy (2001), before covering several variations on the data structure. The text, which for collection-based compression is the collection of files, is regarded as a large set of pattern strings. The fingerprints of the text patterns are pre-computed and stored in the index. String search proceeds by scanning the query string for matching patterns in the text in a manner similar to Algorithm 4: the query string is traversed with a rolling hash function, and the fingerprint of each substring is queried in the index. The matching substrings are then retrieved from the collection for direct string comparison to resolve collisions. If the fingerprint index resides in memory, the only external reads of the collection data are to confirm the matching data and resolve collisions.

The string search algorithm using an index requires three key modifications to Algorithm 4. First, the hash function cannot be randomly selected at query time. Instead, all fingerprints in the index are calculated using a hash function that is selected when the index is first created, and is used for the life of the collection.

Second, another significant change to the Karp-Rabin algorithm arises because
the index is created before the length of the query string is known. In Algorithm 4, the fingerprints are calculated on substrings of the text having the same length as the pattern string, but the index must support search for query strings of various lengths. The solution is to insert substrings having some common property that supports comparison by their fingerprints. Bentley and McIlroy (2001) inserted patterns of a fixed length, $n$, which are referred to as $n$-grams. Some alternative properties are discussed in Section 2.6 on content-defined chunking and selection. When $n$-grams are used, only strings of length $n$ can be found in the index, so it is not possible to find a match for a query string shorter than $n$. A query string with length $m > n$ can be found by searching for each of its $m - n + 1$ substrings of length $n$.

Third, the last modification needed to Algorithm 4 arises because storing the fingerprints of all of the $n$-grams in the text requires more space than the text itself. A text of length $M$ bytes, has $M - n + 1$ $n$-grams, requiring $8(M - n + 1)$ bytes to store the 64-bit fingerprints. The solution is to discard most of the fingerprints, creating a sampled fingerprint index. If the index contains the fingerprint of every $k$-th $n$-gram, then a matching substring of length $k + n - 1$ is certain to be found as one of its $n$-grams will be contained in the index, and shorter matches which are longer than $n$ have some likelihood of being found. A consequence of fingerprint sampling is that a query substring (which may be much longer than $n$) with a match in the text will not be completely identified by $n$-gram fingerprints in the index: many of the $n$-gram fingerprints will have been discarded in the sampling process. This is resolved by a symbol-by-symbol comparison at the same time as comparison of the matching $n$-grams for collision resolution. Bentley and McIlroy (2001) referred to this as extending the match forwards and backwards to find its full extent. An example of the modified algorithm for string search using a sampled fingerprint index is shown in Algorithm 5.
When searching for a complete string which is much longer than \( n \), these additional symbols create the possibility of false positive queries, whereby all of the \( n \)-grams match but the entire string does not. Resolving this situation requires an unnecessary read from the collection to compare the non-matching string. On the other hand, this is not problematic when searching for substrings of the query string, rather than an entire match. The modified algorithm for string search using a sampled fingerprint index is shown in Algorithm 5.

Consider the example of a text containing the string

\[
R = \text{abcdefghijklmnopqrstuvwxyz}
\]

When \( n = 4 \) and \( k = 4 \), the index will contain fingerprints of the \( n \)-grams

\[
\text{abcd, efg, ijk, mnop, qrs, tuvw}
\]

For the query string \( Q_1 = \text{fghijklmnopqrs} \), the \( n \)-grams shown in bold have matching fingerprints in the index, and the entire match will be confirmed during the symbol-wise comparison. Another query string \( Q_2 = \text{cddefghijklm} \) also has two matching \( n \)-grams in the index, but will fail the symbol-wise comparison as the final symbol does not match.

An important effect of using a sampled hash index is the creation of false negative index queries, when there is a matching \( n \)-gram in the collection, but its fingerprint has not been inserted into the index due to sampling. In order to find a matching substring, \( n - 1 \) unsuccessful index queries may be required before a matching \( n \)-gram is found.

Algorithm 5, often with minor variations, has been used in many applications. Important areas for CBC are delta encoding and delta compression, described in detail in Sections 2.5 and 2.7, where it is common to use a fingerprint index for repeated searches of a reference string for the longest match to a prefix of
Algorithm 5 String search using a sampled fingerprint index

**Data:**
- $Q$, the query string of length $N$.
- $R$, the reference string of length $M$.
- $n$, the length of the $n$-grams.
- $k$, the index sample period. The fingerprint of every $k$-th $n$-gram is inserted in the index.

**Result:** The positions of the occurrences of $Q$ in $R$.

1. For each $n$-gram in $R$ from position $i = 0$ to $M - n$ do
   - Create the index. Insert into the index the position and fingerprint of every $k$-th $n$-gram of $R$.
2. If $i \equiv 0 \pmod{k}$ then
3. \hspace{1em} $f_p \leftarrow \text{FINGERPRINT}(R[i, i + n - 1])$
4. \hspace{1em} \text{INDEXINSERT}(i, f_p)$

5. Let $L$ be an empty list of positions in $R$ partially matching $Q$.
6. For $i = 1$ to $k$ do
   - Find the fingerprint matches in the first $k$ $n$-grams of $Q$.
7. \hspace{1em} $f_p \leftarrow \text{FINGERPRINT}(Q[i, i + n - 1])$
8. \hspace{1em} \text{results} = \text{INDEXQUERY}(f_p)$
9. \hspace{1em} For each result $r$ in results, append $r - i$, the beginning of the potential match in $R$, to $L$.

10. For each partial match, $r$, in the list $L$ do
    - Perform a byte-wise comparison of the potential matches to detect hash collision and to extend beyond the matching $n$-gram.
11. For each position $j$ in $Q$ from $j = 0$ to $N - 1$ do
12. \hspace{1em} If $Q[j] \neq R[r + j]$ then
13. \hspace{2em} Delete $r$ from $L$.
14. Return $L$
the query string. Algorithm 5 closely follows two implementations by Bentley and McIlroy (2001) that demonstrated the effectiveness of a sampled fingerprint index for data compression. Also, the \texttt{xdelta} delta encoding utility (MacDonald, 2000) uses a sampled hash index, and the \texttt{vdiff} delta encoder (Hunt et al., 1998) uses a hash index without sampling, limiting the size of the hash table by discarding fingerprints that have already occurred in the input. These delta encoding systems will be discussed in detail in Section 2.5.

One variation on Algorithm 5, used by Ajtai et al. (2002), uses the first matching \(n\)-gram that is found, rather than retrieving all of matches. The match is retrieved from the reference string for collision resolution, and if it matches the \(n\)-gram in the query string, the direct symbol-wise comparison is extended to find the full extent of the string match. While this may not find the longest matching substring in data with many duplicate \(n\)-grams, the advantage is that the \(n\)-gram in the reference string is retrieved from disk as part of a larger block, so it is efficient to extend the string comparison into the remainder of the block.

For very large data collections, the fingerprint index must be stored externally. One approach which explicitly addresses the limitations on searching large fingerprint indexes is proposed by Zhu et al. (2008) and is implemented in commercial de-duplication systems. These systems are intended to conserve storage space for backups and archiving as well as reduce the bandwidth needed for remote replication. The entire file system is broken into variable length blocks and hashed. Duplicate blocks are removed, and an index of the block fingerprints is stored on the file system. Efficient access to the fingerprint index is achieved using two techniques: to address the problem of false negative queries, a Bloom filter in memory is used to test the existence of a fingerprint in the index. By detecting a high proportion of the queries for fingerprints that are not in the index, the number of reads to external storage is greatly reduced. Also, indexes
are stored locally with the blocks they hashed. In typical usage, it is common for contiguous blocks to be access simultaneously, so loading a neighbourhood of fingerprints into cache along with the substrings they refer to reduces the number of disk accesses further. In combination, these techniques reduce the number of disk accesses for fingerprint lookups by up to 99% (Zhu et al., 2008).

2.4.4 Compressed Self-Indexes

The compressed self-indexes, and related data structures such as suffix trees, are potential alternatives for indexing the collection to support string search. We briefly review them here primarily to explain why we preferred fingerprint indexes for collection-based compression, and also to note recent advances which have addressed some of their previous shortcomings.

A suffix tree is a compressed trie containing all of the suffixes of the text $R$, with the start position of each suffix stored at the corresponding leaf (Weiner, 1973). All occurrences of a pattern $Q$ in the text are found in $O(N)$ time by traversing a path from the root of the tree to each successive symbol in $Q$. The subtrie at the end of the path contains all of the suffixes having $Q$ as a prefix. Suffix trees require $O(M)$ space, but with a high constant factor such that the size of the tree is several times greater than the original text, they are unsuitable for collection-based compression, which requires an index much smaller than the collection size.

A suffix array stores the position of each suffix of the text sorted in lexicographic order (Manber and Myers, 1993). It corresponds to the leaves of the suffix tree obtained from a depth-first traversal. The space complexity is $O(M)$ as the array contains $M$ integers, which is still several times larger than the text although smaller than the suffix tree. If the size of the integers is accounted
for, the space is $O(M \log M)$ bits compared to $O(M \log q)$ bits for the text. The search time for a pattern $Q$ is $O(N + \log M)$ when an additional LCP array (longest common prefix) is available.

The compressed suffix array (Grossi and Vitter, 2000, 2005) achieved a substantial reduction in memory usage by representing the suffix array in a compressed form. An improved structure by Grossi et al. (2003) required $MH_h + O(\log \log M/\log_q M)$ bits, with search time $O(N \log q + \text{polylog}(M))$. The term $H_h < \log q$ is the $h$-order empirical entropy of the text. In practice, the compressed suffix array requires less than half of the memory needed to store the original text.

The FM-index (Ferragina and Manzini, 2000) is a compressed index based on the Burrows-Wheeler transformed strings (Burrows and Wheeler, 1994) having many similarities to a suffix array. Search requires $O(N + \log 1+ \varepsilon M)$ time. As with compressed suffix arrays, the text can be reconstructed from the index and there is no need to store the text. A substantial reduction in memory is achieved by using the compressed suffix array or FM-index to represent the text. As a result, these data structures are known as compressed self-indexes. Navarro and Mäkinen (2007) review numerous variations of these data structures.

The effectiveness of collection-based compression depends on extensive repetition of data within the collection, and there has been some investigation of the performance of compressed self-indexes on highly compressible data. However, the entropy model of data similarity only considers the predictability of a symbol using its immediate neighbours, and is not able to exploit similarity over a large distance. Consider an incompressible substring which is repeated some billions of bytes further into the input file. Mäkinen et al. (2010) showed that an entropy-based compressed-self index does not exploit the duplication as the matching string is represented twice in the index. They referred to data
containing such long-range duplicates as *highly repetitive* data and designed a compressed self-index employing a run-length encoding to take advantage of long-range duplicates (Sirén et al., 2009; Mäkinen et al., 2010). Applying traditional data compression techniques to compressed-self indexes has since been the subject of substantial research interest. Claude and Navarro (2011) constructed a compressed self-index using context-free grammars, which can be considered as a general framework for many compression schemes (Charikar et al., 2005). Kreft and Navarro (2011) compressed the index using the LZ77 compression format (Ziv and Lempel, 1977), which is also able to encode long-range duplicates, and found that the index size was 2.5 to 4 times the compressed file size, when it was compressed with p7zip. Despite these recent advances, for the moment compressed self-indexes are too large for use in collection-based compression as it is desirable for the index to take less space than the compressed collection.

Index updates are another difficulty with suffix arrays that has seen recent advances. The traditional suffix array is a static data structure which must be rebuilt entirely to reflect any changes to the underlying text. Consequently, suffix trees are used for indexing dynamic texts. Sirén (2009) developed an algorithm for merging two compressed suffix arrays. While this does not allow edits within an existing file, it does support batch insertion of new files into the collection, which is sufficient for collection-based compression. Furthermore, it allows a parallel construction algorithm for the compressed suffix array by constructing suffix arrays for partitions of the text that are subsequently merged. Salson et al. (2010) developed a dynamic suffix array which updates the index to reflect the insertion, substitution or deletion of a single symbol or contiguous string in the text, although the update time is slow for large texts.

To summarise, since the commencement of this doctoral project, there have been significant advances in compressed self-indexes. It is a possibility that con-
continued progress will make compressed self-indexes the preferred data structure for collection-based compression. Nonetheless the fingerprint index adopted in this project is the best alternative at present, and has several advantages which suggest it will remain the preferred data structure. Ferragina and Manzini (2010) compared compressed self-indexes with two stage compression scheme using Bentley and McIlroy (2001) delta encoding followed by LZMA compression of the delta, finding that self-indexes achieved compression several times less than BM with LZMA, and also that decompression was much slower. Query speed is a key consideration, requiring constant time in the fingerprint index. Also, the index size can be tuned, at the cost of reduced search effectiveness, by increasing the sampling length. This is not a criticism of compress self-indexes, but rather a consequence of the fact that they are designed for a different problem. Suffix arrays index the entire text and support search for all substrings. Collection-based compression has a relaxed search requirement as significant compression can be achieved by searching only for matching substrings longer than a certain minimum size. The additional capability of compressed self-indexes necessarily comes at the cost of storing significant additional data. On the other hand, it can be expected that the compressed self-indexes would achieve better compression as all matching strings can be found, rather than just long ones.

2.5 Delta Encoding

Having constructed a fingerprint index to find substrings in the collection string that match substrings of the query string, the next step in collection-based compression is to encode the query string by replacing these matching substrings with references to their duplicates in the collection string. This process is known as delta encoding or data differencing. In the delta encoding literature, the query
string $Q$ of length $N$ is often known as the *target file*, and the collection string $R$ of length $M$ is known as the *source file* or *reference file*. The resulting file, which encodes the target using references to the source is variously called a *delta*, *diff*, or *patch*, file as it represents the differences between the two files.

The problem of finding a delta is an example of the *string-to-string correction* problem, first described by Wagner and Fischer (1974): given two strings, find a sequence of edit operations which transform one string into the other. The solution depends on the allowed edit operations on substrings, such as insertion, deletion, substitution of characters, relocation (referred to below as a *block move*), and reversal of substrings. This type of delta encoding is known as *COPY/ADD* encoding, where the delta contains these two types of instructions which are executed in sequence to generate the reconstruction, $Q'$, of the query file. A *COPY* instruction describes a substring of $R$ to be appended to $Q'$, and an *ADD* instruction describes a verbatim string to be appended to $Q'$. The delta encoding operation involves searching for matching substrings of $Q$ and $R$ that are encoded as *COPY* instructions. For example,

\[
R: \text{catdoggicowgoat} \\
Q: \text{pigxyzgoatcow} \\
\Delta: \text{COPY}(7, 3), \text{ADD}(xyz), \text{COPY}(13, 4), \text{COPY}(10, 3)
\]

where the *COPY*(position, length) instruction specifies the location and length of the matching substring in $R$.

Delta encoding is widely used as a compression technique to reduce storage space and network bandwidth use. For example, disaster recovery systems for large file systems typically store snapshots of the data as a delta with respect to the previous snapshot, requiring much less storage space than a full copy of the file system, and needing less time to transmit to a remote system for off-site storage. Other applications are efficient transmission of large files over the
Algorithm 6  Greedy algorithm for \textsc{Copy/Add} delta encoding. All symbols from the alphabet $\Sigma$ are assumed to occur in the reference string, so no \textsc{Add} instructions are needed.

Data:  $R$, the reference string.
       $Q$, the query string of length $N$.

Result:  $\Delta$, the optimal sequence of \textsc{Copy} instructions encoding $Q$.

1:  $i, j \leftarrow 0$
2:  \textbf{while} $i < N$  \textbf{do}
3:      $r, l \leftarrow$ the location $r$ in $R$ and length $l$ of the longest match with the substring at position $i$ in $Q$.
4:      $\delta_j \leftarrow (\textsc{Copy}, r, l)$, and append $\delta_j$ to $\Delta$.
5:      $i \leftarrow i + l$, $j \leftarrow j + 1$
6:  \textbf{Return} $\Delta$.

internet, synchronisation of data mirrors, efficient storage of multiple versions of dynamic files in version control systems, and efficient transmission of updated web pages.

2.5.1  Optimal Encoding

Tichy (1984) described several algorithms for \textsc{Copy/Add} encoding that find a minimal delta, meaning one which describes the target file using the least number of references to the source file. He called this the \textit{string-to-string correction with block moves} problem. The optimality of the greedy algorithm was also proved by Burns and Long (1997). Tichy defined a matching string reference (called a block move) as a triple $(p, q, l)$ where $R[p, p + l - 1] = Q[q, q + l - 1]$ is a substring of length $l$ which occurs in both the source file $R$, and target file $Q$. The delta is constructed as a covering set of block moves such that every element of $Q$ is referred to in $R$ in exactly one block move.

The algorithm is shown in Algorithm 6. The target file is encoded from beginning to end in a greedy manner, meaning that, at the current offset, the largest matching substring is used. The time complexity is $O(NM)$ when the
search for the largest matching substring uses a classical linear-time string search
algorithm such as Knuth-Morris-Pratt, and the space complexity is $O(N + M)$.  
Tichy improved on this by indexing the source file using a suffix tree, achieving
$O(N + M)$ time and space complexity. Tichy applied the algorithm in a line-
oriented fashion to create the RCS version control system. The \texttt{bdiff} program
(Hunt et al., 1998) implements Tichy’s algorithm using suffix arrays and extends
it to handle large files by sequentially processing the source and target files in
blocks. Another suffix array-based implementation is \texttt{bsdiff} (Percival, 2003).

An important observation is that, to achieve a compact encoding, the target
and source files do not need to be similar in size. Rather, the source file can
be much larger and contain any amount of non-matching data, provided it also
contains substrings that match the target file.

Tichy’s algorithm should not be confused with a related family of algorithms
known as insert/delete delta encoding, which is implemented in the \texttt{diff} utility on
UNIX. It performs a byte-by-byte, or line-by-line comparison of two files using an
algorithm discovered independently by Myers (1986) and Ukkonen (1985) solving
the longest common subsequence (LCS) problem. The LCS solution achieves good
compression when the source and target files are related by a sequence of insertion
and deletion edits, but it does not take advantage of block moves. Consequently,
the LCS solution is unable to use similar data from any location in the source
file, but is limited so that the matching data occurs in same order in both the
source and target files.

\footnote{This should not be confused with the \texttt{bdiff} program (‘big diff’) commonly installed on UNIX, which compares large files by partitioning them and then calling \texttt{diff} for each part.}
2.5.2 Suboptimal Encoding with Minimum Match Length

With time complexity $O(NM)$, the greedy algorithm is too slow for CBC. Furthermore, Tichy’s proof of optimality is valid only when matching strings as short as a single byte may be encoded as references. However, when using fingerprint-based search, matching strings shorter than $n$, the length of the $n$-grams on which the fingerprints were calculated, cannot be found and Tichy’s proof is not valid. To show this, consider the proof of optimality of the encoding generated by Algorithm 6.

**Theorem 4.** The greedy algorithm finds an encoding of the target string $Q$ in terms of the reference string $R$ using the minimum number of substring matches (COPY instructions).

**Proof.** (by induction from Burns and Long (1997)) Suppose there is a delta $G$ found by the greedy algorithm, and a shorter delta $P$ using fewer substring matches than $G$. Let $g_i$ and $p_i$ be the lengths of the $i$-th substring matches in $G$ and $P$. The first $j$ instructions from $G$ encode the first $\sum_{i=1}^{j} g_i$ symbols of the target string $Q$.

We assume that $P$ uses fewer instructions than $G$, so we look for the first $j$ instructions where $\sum_{i=1}^{j} g_i \leq \sum_{i=1}^{j} p_i$ as $P$ must encode more symbols than $G$ using the same number of instructions. Hence $p_j \geq \sum_{i=1}^{j-1} g_i - \sum_{i=1}^{j-1} p_i + g_j$.

However, at the beginning of the delta, $g_1 \geq p_1$ as the greedy algorithm selects the longest match. So, for some instructions at the beginning of the delta $\sum_{i=1}^{j-1} g_i - \sum_{i=1}^{j-1} p_i \geq 0$, and the assumption requires that, at some $j$, $p_j \geq g_j$. But $G$ always takes the longest match so there can be no string match starting at $\sum_{i=1}^{j} g_i$ and longer than $g_j$. Thus there can be no $P$ encoding the target string with fewer COPY instructions than $G$, and the greedy algorithm finds a delta using the minimum number of COPY instructions.
When there is a minimum match length $n$, it is possible that the longest match at the current position in $Q$ is shorter than $n$, so no match will be found. The next few characters will be encoded as ADD instructions until the start of the next matching substring.

The greedy algorithm using string search with a minimum length, which produces a suboptimal encoding, was implemented by Reichenberger (1991). He constructed a hash table referencing each $n$-gram in the source file, and then, to encode the target file, the fingerprint of the next $n$-gram was queried in the hash table, and all potential matches were then extended by a symbol-wise comparison to find the longest match. In the worst case, every $n$-gram could be a match or a collision, so the time complexity is $O(NM)$. The space needed is $O(M + N)$ as the hash table size is $O(M)$.

Burns and Long (1997) proposed a variation of the algorithm using constant space, which achieved a good encoding for files related by insertions and deletions, although it was unable to exploit block moves. As the algorithm only search forward in both files, it is unable to find string matches to locations earlier in file, and this algorithm finds a solution similar to the longest common subsequence. The target and source file were stepped through simultaneously and at each position, the $n$-grams in both files were fingerprinted and queried in a hash table. If a match was not found, then the $n$-gram references were inserted into the index and the procedure repeated at the next position. If a match was found, symbol-wise comparison was used to confirm the match, then the match was extended forward in the two files until the next non-matching symbol. Then, importantly, the hash table was flushed and the searching procedure was continued from the end of the match in both files with an empty hash table. Constant space for the hash table was achieved by discarding references to $n$-grams having a fingerprint that had previously been inserted. Flushing the hash table is important to en-
sure that $n$-gram fingerprints at the current location are not all discarded. The final innovation was to extend the symbol-wise comparison backwards as well as forwards.

Ajtai et al. (2002) made further enhancements while retaining the constant space use. As with Burns and Long (1997), only one $n$-gram reference was stored in a hash table for each fingerprint value; subsequent (or prior) fingerprint occurrences were discarded. Also, string matching was extended both backwards and forwards from each matching $n$-gram to determine the full extent of the match. This created the possibility that an extended match covered parts of the target file which had already been encoded, so a correction step was added to merge or replace overlapping matches. Of particular interest for large collections is their correcting 1.5-pass algorithm with checkpointing. The hash table for the source file was created first, and could be pre-computed. The default behaviour of 1.5-pass is to store the first occurrence of a fingerprint and discard subsequent instances. However, for large source collections there will be many fingerprint duplicates (from repeated $n$-grams as well as collisions) resulting in less coverage of the data toward the end of the source file. To achieve even, although sparse, coverage of the source file, a system of checkpointing was proposed such that in any region of the query file a subset of the fingerprints were replaced if a duplicate occurred, while outside of the region, duplicate fingerprints were discarded. For example, the fingerprints beginning with 0000 form a set containing $1/16$-th of the total fingerprint space. The target file was divided into blocks and a checkpoint (a particular value for the bottom 4 bits of the hash) was associated with each block such that only $n$-grams with hashes in the checkpoint are inserted into the index. Once the source file had been indexed, the target file was encoded by scanning from start to finish. But observe that this is not a greedy algorithm as the string search does not find all matches and select the largest. Moreover, all but one of
the references to the many occurrences of a repeated \( n \)-gram have been discarded from the index, so the string search can only find a single matching \( n \)-gram. While, the performance of this algorithm for very large files is not presented, it provides some scope for tuning the balance between compression performance and resource use.

Chedid and Mouawad (2006) investigated variations of the Ajtai et al. (2002) algorithms. Specifically, rather than only storing one string offset for each fingerprint and discarding subsequent occurrences, they stored up to \( l \) references in a linked list, only discarding occurrences when the list was full. Compared with the \( l = 1 \) case, there was a significant improvement in compression. It would be interesting to compare with the performance gain when the hash table size is increased by an equivalent amount.

Bentley and McIlroy (2001) used a sampled fingerprint index for the string search in their delta encoding scheme as described in Section 2.4.3. They used a sample period of \( k = n \), and found that \( n = 100 \) was the most effective \( n \)-gram length.

The \texttt{xdelta} program (MacDonald, 2000) constructs a hash table of the 16-grams of the reference file. As the file is indexed, a new fingerprint replaces an existing fingerprint in the hash table, thereby limiting memory use to a constant size. Consequently, like the Ajtai algorithm, it is not a greedy algorithm as only one match is found. This is then extended forwards and backwards using symbol-by-symbol comparison.

The \texttt{vdelta} delta encoding tool used hash-based search (Hunt et al., 1998). It is reported to be 3 to 6 times faster than \texttt{bdiff} and uses less space, although at a small cost in compression performance, as it does not always find the longest matching substring. It is included in the \texttt{vcdiff} tool set (Korn and Vo, 2002).

The \texttt{hsadelta} utility employs another differential encoding algorithm using
constant space (Agarwal et al., 2006). It uses a sampled fingerprint index in
which every $n$-th $n$-gram is indexed, and achieves constant space by varying $n$
with the input size so that the number of index entries is constant. Consequently,
worse compression is expected for larger collection sizes. Also, increasing the size
of the collection requires re-hashing all of the fingerprints with a larger $n$, which
is not practical so the size of the collection must be known in advance. However,
hsadelta includes features that are likely to be useful for CBC. It uses hash
sampling with a fixed sampling period in order to reduce the index size to a
fraction of the collection size. The problem of false negatives is addressed by
including several data structures intended to quickly answer most queries for
fingerprints which are not in the index. Another feature is that it uses a larger
fingerprint space of $2^{64} - 1$ to obtain a very small probability of hash collisions.
This index is sparsely populated even for multi-terabyte collections, so a tree
structure is used for the index, rather than a hash array. As hsadelta records
duplicate fingerprints, it is able to locate all instances of a repeated long substring.
Where multiple matches are found, it searches for the longest in a greedy manner.

2.6 Content-Defined Selection and Chunking

As described in the previous section, an index of every $k$-th $n$-gram fingerprint
may have up to $k - 1$ false negative index queries before a matching substring is
identified. This inefficiency can be avoided using content-defined selection meth-
ods, in which a set of locations in a string is deterministically selected based
solely on the symbols in some limited vicinity near each location. We refer to
the selected locations as selection points, although various names are used in the
literature, such as anchors (Manber, 1994), break points (Brin et al., 1995) and
cut points (Bjørner et al., 2010). By adding fingerprints to the index only at
selection points, we know that a fingerprint that is not at a selection point will not be in the index, and the false negative index query can be avoided. The set of selection points is denoted $S_P$. Location $i$ in string $Q$ is a selection point, $i \in S_P$, if the content of the substring $W_i = Q[i, \ldots, i + w - 1]$, called the local window, of length $w$ satisfies some criterion. For example, if $W_i$ is an element of a criterion set $C \subset \Sigma^w$, or $W_i$ satisfies a criterion function, such as having the lowest hash value in the local vicinity. Selection points have the useful property that, when changes are made at a distant location in the string, their position relative to their local neighbourhood is unchanged.

Chunking describes methods for partitioning a string at selection points, and the resulting substrings are known as chunks. Sometimes in the literature, chunking is used as a general term for partitioning a string, including at points selected independent of the string content, such as into $n$-grams at every $k$-th offset as described previously (Bjørner et al., 2010). In our terminology, “chunk” refers to a content-defined piece, and a fixed-offset substring is an $n$-gram. When it is necessary to use the more general sense, we qualify it such as “content-defined chunks” and “fixed-offset chunks”.

To understand the usefulness of content-defined selection, consider the phased offset problem, illustrated in in Figure 2.3. We wish to search for matching data in two long strings by comparing some selected substrings. When the second string is a copy of the first, except for an insertion or deletion, the symbols following the edit are shifted to the right or left. If we compare substrings selected at fixed offsets, the selected substrings will not match, and the similarity of the data will not be found. However, when we compare substrings selected according to their local content, which is unchanged following the edit, the matching data is detected.

We have already seen one solution to the phased offset problem: the Karp-
Input strings:
\[ R = \text{abcdefghijklmnopqrstuvwxyz} \]
\[ Q = \text{abcdefZghijklmnopqrstuvwxyz} \]

Fixed-offset selection (4-grams):
\[ \text{abcd efgh ijkl mnop qrst} \]
\[ \text{abcd efZg hijk lmmo pqrs} \]

Content-defined chunking (\( w = 2 \)):
\[ \text{abc defg hij klmno pqrst uv} \]
\[ \text{abc defZghij klmno pqrst uv} \]

Figure 2.3: An example of the phased offset problem. The new string \( Q \) is a copy of the original string \( R \) with a Z inserted near the start. When fixed-offset selection is used, none of the sampled 4-grams following the insertion in \( Q \) match the sampled 4-grams from \( R \), even though part of the strings match. With content-defined chunking, the chunks are terminated by the selection points (underlined), and the chunks following the insertion in \( Q \) match the chunks in \( R \).

Rabin string search algorithm. One of the strings is sampled at fixed offsets, then those selected \( n \)-grams are compared with every \( n \)-gram from the other string. However, there are situations when it is not practical to compare every \( n \)-gram, and content-defined selection is preferred. Applications include obtaining approximate measures of file similarity (Manber, 1994; Broder, 1997) reviewed in Section 2.8; plagiarism detection (Brin et al., 1995; Schleimer et al., 2003); identification of near-duplicate results for web searches (Broder, 2000; Potthast and Stein, 2008); remote differential file compression (Muthitacharoen et al., 2001; Bjørner et al., 2010); and file de-duplication (Quinlan and Dorward, 2002; Zhu et al., 2008). Also, content-defined selection is a key component of the data archive systems described in Section 2.9.

Several approaches towards content-defined selection have been proposed, each with its own selection point criterion. Their effectiveness is largely determined by the statistical properties of the distribution of selection points. Par-
particularly desirable qualities are a small variance in the distance between selection points (also referred to as the chunk length as chunks are terminated at the selection points), bounds on the smallest and largest distances, and the expected number of symbols before synchronised selection points occur following a transition from non-matching to matching data, a measure named slack (Bjørner et al., 2010). The slack quantifies susceptibility to the phased offset problem. In the case of fixed-offset selection, the selection points may never synchronise following a transition to matching data, and the slack is unbounded. The statistical properties of the chunk length for several content-defined selection algorithms are summarised in Table 2.1.

Manber (1994) proposed two methods. He first employed a dictionary of short (4 symbols) substrings named anchors, and defined selection points to be the locations where anchors occur in the input string. However, the set of anchors needed to be tuned to the bias of the input data, and he reported anchors occurred either too frequently or too rarely, so that it was difficult to select an anchor set obtaining well-distributed selection points. Longer anchors, and a larger dictionary, involved increasing computational cost, as each anchor is compared with each location in the string. Manber therefore preferred a second method in which \( n \)-gram hashes are calculated over the string with a rolling hash, and the selection points are locations of \( n \)-grams having a hash \( h \), such that \( h \equiv 0 \pmod{c} \). The parameter \( c \) determines the expected distance between selection points. For example, for \( c = 16 \), the criterion set \( C \) comprises the \( n \)-grams having hashes with 0 in the 4 least significant bits, and the expected distance between selection points is 16 symbols. An example is shown in Figure 2.4 (a). This scheme has two main advantages: the anchors are defined by the hash function, and do not need to be selected by hand; and the anchor comparison is performed in a single integer comparison of the hashes, which is faster than a symbol-by-symbol string
Mauber (1994)  Selection point at \(i\) when \(h(i) \equiv 0 \pmod{c}\).
Probability \(p\) of a selection point at \(i\) is \(p = 1/c\).
Chunk length \(L\) distribution is geometric, \(\Pr[L = j] = p(1-p)^{j-1}\).
Local algorithm.

| Method                        | Selection point at \(i\)                          | Probability \(p\)                                 | Statistical Properties                                      |
|-------------------------------|---------------------------------------------------|--------------------------------------------------|
| Manber (1994)                 | when \(h(i) \equiv 0 \pmod{c}\)                    |                                                  | \(\mathbb{E}(L) \approx c\), \(\text{Var}(L) \approx c^2 - c\) |
|                               |                                                  | \(\text{Min}(L) = 1\), \(\text{Max}(L) \to \infty\) | \(\mathbb{E}(\text{slack})/\mathbb{E}(L) \approx 1 - 1/c\) |

\[
\begin{align*}
\mathbb{E}(L) & \approx c \\
\text{Var}(L) & \approx c^2 - c
\end{align*}
\]

<table>
<thead>
<tr>
<th>Method</th>
<th>Selection point at (i)</th>
<th>Probability (p)</th>
<th>Statistical Properties</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bjørner et al. (2010)</td>
<td>when (h(i) \equiv 0 \pmod{c})</td>
<td></td>
<td>(\mathbb{E}(L) \approx c + m), (\text{Var}(L) \approx c^2 - c)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>(\text{Min}(L) = m), (\text{Max}(L) \to \infty)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>(\mathbb{E}(\text{slack})/\mathbb{E}(L) \approx 1.41)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>(\Pr[L &gt; K\mathbb{E}(L)] &gt; e^{-K}, k = m/c)</td>
</tr>
</tbody>
</table>

| Method                        | Selection point at \(i\)                          | Probability \(p\)                                 | Statistical Properties                                      |
|-------------------------------|---------------------------------------------------|--------------------------------------------------|
| LBFS method                   | when \(h(i) \equiv 0 \pmod{c}\)                    |                                                  | \(\mathbb{E}(L) \approx c + m\), \(\text{Var}(L) \approx c^2 - c\) |
|                               |                                                  |                                                  | \(\text{Min}(L) = m\), \(\text{Max}(L) \to \infty\)   |
|                               |                                                  |                                                  | \(\mathbb{E}(\text{slack})/\mathbb{E}(L) \approx 1.41\) |
|                               |                                                  |                                                  | \(\Pr[L > K\mathbb{E}(L)] > e^{-K}, k = m/c\)          |

| Method                        | Selection point at \(i\)                          | Probability \(p\)                                 | Statistical Properties                                      |
|-------------------------------|---------------------------------------------------|--------------------------------------------------|
| Interval filter method        | when \(h(i) \equiv 0 \pmod{c}\)                    |                                                  | \(\mathbb{E}(L) \approx c + m\), \(\text{Var}(L) \approx c^2 - c\) |
|                               |                                                  |                                                  | \(\text{Min}(L) = m\), \(\text{Max}(L) \to \infty\)   |
|                               |                                                  |                                                  | \(\mathbb{E}(\text{slack})/\mathbb{E}(L) \approx 1.41\) |
|                               |                                                  |                                                  | \(\Pr[L > K\mathbb{E}(L)] > e^{-K}, k = m/c\)          |

\[
\begin{align*}
\mathbb{E}(L) & \approx c \\
\text{Var}(L) & \approx c^2 - c
\end{align*}
\]

| Method                        | Selection point at \(i\)                          | Probability \(p\)                                 | Statistical Properties                                      |
|-------------------------------|---------------------------------------------------|--------------------------------------------------|
| Winnowing                     | when \(h(i)\) is minimum in window of width \(w\)|                                                  | \(\mathbb{E}(L) \approx w\), \(\text{Var}(L) \text{ not known}\) |
|                               |                                                  |                                                  | \(\text{Min}(L) = 1\), \(\text{Max}(L) = 2w - 1\)      |
|                               |                                                  |                                                  | \(\mathbb{E}(\text{slack})/\mathbb{E}(L) \text{ not known}\) |

| Method                        | Selection point at \(i\)                          | Probability \(p\)                                 | Statistical Properties                                      |
|-------------------------------|---------------------------------------------------|--------------------------------------------------|
| Bjørner et al. (2010)         | when \(h(i) \equiv 0 \pmod{c}\)                    |                                                  | \(\mathbb{E}(L) \approx 2w + 1\), \(\text{Var}(L) \text{ not known}\) |
|                               |                                                  |                                                  | \(\text{Min}(L) = w + 1\), \(\text{Max}(L) \to \infty\) |
|                               |                                                  |                                                  | \(\mathbb{E}(\text{slack})/\mathbb{E}(L) \text{ not known}\) |

| Method                        | Selection point at \(i\)                          | Probability \(p\)                                 | Statistical Properties                                      |
|-------------------------------|---------------------------------------------------|--------------------------------------------------|
| Local maximum                 | when \(h(i)\) is maximum in window of width \(i - w\) to \(i + w\)| | \(\mathbb{E}(L) \approx 2w + 1\), \(\text{Var}(L) \text{ not known}\) |
|                               |                                                  |                                                  | \(\text{Min}(L) = w + 1\), \(\text{Max}(L) \to \infty\) |
|                               |                                                  |                                                  | \(\mathbb{E}(\text{slack})/\mathbb{E}(L) \text{ not known}\) |

| Method                        | Selection point at \(i\)                          | Probability \(p\)                                 | Statistical Properties                                      |
|-------------------------------|---------------------------------------------------|--------------------------------------------------|
| Fixed offset                  | when \(i \equiv 0 \pmod{n}\).                     |                                                  | \(\mathbb{E}(L) = n\), \(\text{Var}(L) = 0\)           |
|                               |                                                  |                                                  | \(\text{Min}(L) = n\), \(\text{Max}(L) \to \infty\)   |
|                               |                                                  |                                                  | \(\mathbb{E}(\text{slack})/\mathbb{E}(L) \to \infty\)  |

Table 2.1: Comparison of the statistical properties of the distance between adjacent selection points (chunk length), \(L\), for several content-defined selection methods. Principally from Bjørner et al. (2010).
Brin et al. (1995) extended this approach to develop content-defined chunking, which they used to find blocks of matching data for plagiarism detection. When the hash of a 48-byte sliding window equalled a specified value, a break point was defined, the chunk terminated at the end of $W_i$, and a new chunk commenced at the next position. The expected chunk length is $c$, although they observed that care must be taken of particular situations, such as files containing long runs of zeroes that would contain either, depending on the hash function being used, no selection points, or a selection point at each location. Muthitacharoen et al. (2001) employed the content-defined chunking of Brin et al. (1995) in their low bandwidth file system (LBFS), using $c = 2^{13}$ for an expected chunk length of 8192 bytes, and a 48-byte window.

While the probability of very long chunks diminishes gradually, there is no upper bound on chunk length. Muthitacharoen et al. (2001) used a heuristic to limit the lengths of chunks: they implemented a maximum chunk length of $2^{16}$ by terminating the chunk and starting a new one. Also, a minimum chunk length of $m = 2^{11}$ bytes was imposed by ignoring any selection points occurring when the chunk was shorter. However this modification violates the locality of the algorithm: the location of a selection point can now be affected by symbols outside of the local window (Schleimer et al., 2003). The problem occurs when there is a concentration of potential selection points in close proximity, such that some of them are skipped over for being too close to their predecessor. An edit changing the first selection point in the chain will cause a different grouping of the following selection points to be skipped.

Muthitacharoen et al. (2001) observed a significant departure from randomness where some input files had a higher than expected proportion of excessively long chunks. They associated these with highly structured or stylised files. The
Lizards and frogs

Hashes (of 8-grams)
20  16  11  31  11  28  24  17  8  29

(a) Manber \( (c = 8) \)

(b) Winnowing (window has 4 8-grams)

Figure 2.4: Content-defined selection methods. Hashes of 8-grams are shown below the string “Lizards and frogs”. The selection points are shown as circled hash values. (a) Selection points from the Manber method are the positions of 8-grams with hash divisible by \( c = 8 \). (b) For winnowing, the hashes within each step of the sliding window are shown, and the position of the minimum hash is the selection point.
problem is that, in repetitive strings, there are fewer unique $n$-grams, which changes the probability distribution.

Bjørner et al. (2010) described interval filter chunking, where a selection point is defined at position $i$ if the $n$-gram at $i$ meets the criterion $S_i \in C$, and the $m$ preceding $n$-grams are not in $C$. This differs from the LBFS method by excluding all selection points in a close bunch. Interval filter chunking is claimed by US patent 7 555 531 (2009).

Schleimer et al. (2003) addressed the problem of excessive gaps with a method named winnowing, which guarantees an upper bound on the distance between selection points, shown in Figure 2.4 (b). In a sliding window containing $w$ adjacent $n$-grams, hence having length $w + n – 1$ symbols, the position of the $n$-gram having the minimum hash is chosen as the selection point. When the minimum hash occurs at multiple $n$-grams in the window, the right-most point is selected. Again, this method is vulnerable to input containing long sequences of repeated symbols. All $n$-grams in the local window are the same and have the same hash, so every position is a selection point. The authors introduced a modification that the selection point of the previous window is chosen if it is also a minimum in the current window, otherwise the right-most minimal hash is selected, although this violates the locality of the algorithm similarly to the heuristic of Muthitacharoen et al. (2001). As each window must contain a selection point, the distance between selection points is bounded by $w + n – 1$. It is possible, although unlikely, for selection points to be adjacent, and long runs of adjacent selection points due to repetitive input data are prevented by the select previous tie-breaking rule. Winnowing is claimed by US patent 6 757 675 (2004).

A team at Microsoft (Teodosiu et al., 2006) developed the local maximum chunking algorithm, which is closely related to winnowing, and presented a detailed mathematical analysis of the chunk length distribution (Bjørner et al.,
Chapter 2. Literature Review

A location $i$ is a selection point if the hash of the $n$-gram at $i$ is the maximum of the $n$-gram hashes of the preceding $w$ and following $w$ positions. The authors were unable to find approximations for the chunk length variance and expected slack. Experimental comparisons on several types of data showed that local maximum chunking had smaller variance than either the Manber method or interval filter chunking. The probability of long chunks decreases faster than exponentially, a significant advantage over the previous methods. In particular, the probability of chunks longer than 4 times the expected chunk length ($K = 4$) is smaller for the local maximum algorithm than the Manber method and the LBFS method with $k = 1/4$. When $K$ increases to 7, the probability is 1000 times less than LBFS with $k = 1/4$. There is an additional computational cost for finding the maximal hash in the local window, compared to the interval filter and Manber algorithms, which may all be processed in a single pass through the input. The authors give an algorithm requiring

$$1 + \frac{\ln w}{w} + O\left(\frac{1}{w}\right)$$

comparisons per symbol on average, which is an insignificant additional cost. Local maximum chunking is deployed in the remote differential compression (RDC) algorithm of the Microsoft Distributed File System Replication (DFSR) engine, a component of the Windows operating system. Local maximum chunking is claimed by US patent 7,555,531 (2009).

Another contribution of Teodosiu et al. (2006) is recursive chunking. After the initial chunking step, the individual chunks are hashed and concatenated into a string in the same order as they appear in the input string. The chunking algorithm can then be applied to this string to create chunks of chunks. The advantage is that very long substrings of matching data, spanning numerous
chunks, can be identified in a single comparison.

**Chunking biased data** The detailed statistical analyses of these algorithms by Bjørner et al. (2010), shown in Table 2.1, assume that the input string has maximum entropy: specifically that the probability of a symbol occurring at a location in the input string is independent of the symbols at other locations, and that the probabilities are uniformly distributed. However, this assumption does not generally apply to the biased data encountered in CBC, or the various applications for content-defined chunking noted above. Bjørner et al. (2010) reported results with non-random data that were similar to their statistical analysis, but they also noted specific cases that deviated from the uniform-derived values.

Real-world data departs from the randomness assumption in two distinct ways. The first involves data from a biased Markov source. That is, the probability model is stationary and the resulting string exhibits no periodicity, but the symbol frequencies are non-uniform and dependent on a finite window of prior symbols. Natural language data typically approximates these statistical properties. Bjørner et al. (2010) attempted to address this using a preliminary “randomisation” step: they replaced each symbol in the input $S$ with a hash value calculated over a small window, using a Rabin rolling algebraic hash function. The intention was that the string of hashes would conform more closely to the randomness assumptions required by their statistical analysis. They conceded that it is not possible to increase the randomness of a string using a deterministic hash function, and also observed that omitting the preliminary hashing step did not lead to poorer statistical properties of chunk length distribution. Essentially, they demonstrated that, for rolling hash functions, a hash of a hash is no more random than a hash from a single function. This observation agrees with the theoretical result from Lemire and Kaser (2010) that rolling hashes of overlapping
The occurrence of repeated \( n \)-grams in the input string. This has already been noted when describing the motivation for adding heuristics to the LBFS (Muthitacharoen et al., 2001) and local maximum algorithms (Björner et al., 2010): a long sequence of repeated symbols can lead to either selection points at every location, or no selection points until the conclusion of the sequence. The preliminary hash encoding described in the preceding paragraph is of no help as the repeated \( n \)-grams create repeated hash values. In fact, Björner et al. (2010) observe that the occurrence of long sequences of repeated symbols prevents the existence of a chunking method having bounds on both minimum and maximum chunk length. One remedy suggested by Björner et al. (2010) is to perform a preliminary run-length encoding step, in which sequences of repeated substrings are replaced with a single occurrence and the number of repeats, but no evaluation of this approach is presented.

In a chunking application, there is typically a cost relating to the number of chunks, and a benefit from avoiding many short chunks. The local maximum algorithm is preferred as it has a minimum chunk size, a rapidly diminishing probability of long chunks, and a smaller chunk length variance compared to the other methods which also have a minimum chunk length.

However, in an application involving selection of \( n \)-grams for indexing, the cost of short distances between selection points is merely some additional index entries. Alternatively, if some of the selection points are not indexed, the cost is the possibility of some false-negative index queries. In this situation, it may be better to limit the occurrence of very long distances between selection points, and the winnowing algorithm is preferred.

All methods are unable to handle long strings of repeated symbols in a prin-
cipled manner. In many application domains, however, this is not problematic, or else can be controlled by specific workarounds.

2.7 Delta Compression Systems

Delta encoding can achieve substantial reductions in file size, and, when combined with the techniques of data compression to achieve further reductions, is known as delta compression or differential compression. Generally, this involves further processing of the delta file to achieve a compact format. The principal techniques employed are efficient codes for the substring position and length values in the COPY instructions, storing the COPY and ADD instructions in separate segments of the delta file to enhance subsequence compression, and the use of standard data compression tools (gzip, 7-zip and bzip2) to compress the ADD instructions, or sometimes the whole delta file.

Hunt et al. (1998) made a significant contribution in vdelta by using COPY instructions to encode strings matching data earlier in the target file as well as in the reference file. Conceptually, the target file is appended to the reference file and they are both read by vdelta as a single input stream, which then emits the delta encoding for the portion of input from the target file. The encoding refers to matching strings occurring anywhere in the previous input, which is the reference file and that part of the target file that has already been read. This is a generalisation of the dictionary encoding of Ziv and Lempel (1978) to delta encoding. As with delta encoding, in Ziv-Lempel compression the input stream is encoded as references to matching strings occurring previously in the input. Vdelta can perform traditional file compression — create a Ziv-Lempel encoding of a file with respect to itself — by loading an empty source file. Hence LZ77 compression is a special case of delta compression.
Another contribution from \texttt{vdelta} is the VCDIFF file format for differentially compressed data, which supports several coding schemes for efficient storage of file offsets (Korn and Vo, 2002; Korn et al., 2002). This may be a suitable basis for a CBC file storage format, with the addition of capability to refer to multiple source files.

The \texttt{zdelta} utility (Trendafilov et al., 2002) builds on \texttt{vdelta}, achieving further compression by Huffman coding the references to matching strings in the source file. It is implemented by modification of the \texttt{zlib} library for Ziv-Lempel compression. Memory is limited for the hash index by imposing a sliding window on the input files.

An alternative to using a data compression library is to compress the delta file using a data compression tool. Bentley and McIlroy (2001) used a two-stage pipeline in which the delta created in their first stage was compressed with \texttt{gzip} in the second stage.

**Delta compression with multiple reference files** Delta encoding with respect to multiple reference files is a simple extension of single-file delta encoding by concatenating the reference files before encoding. However, few systems have been implemented, and some technical enhancements to single-reference-file delta encoders are needed before it is practical.

A rudimentary system can be implemented using existing tools. For example, the reference files could be stored in concatenated form using the \texttt{tar} utility, then encode the query file with respect to the tar file with \texttt{xdelta} or \texttt{vcdiff}.

The \texttt{Zdelta} utility (Trendafilov et al., 2002) is the only delta-encoder that explicitly supports multiple reference files. However, it was developed for encoding HTML files in web caches, and is quite limited for the purposes of CBC. It only allows four reference files, and does not accept large input files.
2.8 Resemblance Detection

An alternative approach to CBC is to identify the file in a collection which is most similar to the query file, then use that as the reference file to create a delta encoding of the query file. Techniques for identifying similar files have been investigated for other applications unrelated to CBC, such as plagiarism detection, removal of near-duplicates from web search queries, content-addressable file systems, distributed file systems, and peer-to-peer file systems. Many approaches have been proposed, such as heuristics to identify related file versions based on file names, or using domain-specific knowledge such as the similar content between an MSword file and its PDF exported version. These are reviewed in detail in Kumar and Govindarajulu (2009). Here, we review the techniques known as resemblance detection that have been use in the collection compression systems considered in Section 2.9.

What does it mean to say that two files are similar? The notion of similarity or resemblance varies according to the needs of the application domain. In revision control, similar files are related by a small number of edits. In plagiarism detection, similar files are considered to have blocks of identical text in a similar order. For CBC, a looser notion of similarity is needed: that the similar files share some common blocks of data. The shared substrings do not have to be in sequential order, but can be in arbitrary locations as though the files have been shuffled.

Similarity detection involves extraction of features from a file that summarise its contents, then searching for files with some common features, and finally selecting the file with the features having the closest fit to the original file. The features may be shared substrings, or more typically smaller features such as fingerprints of blocks of data.
Broder (1997) proposed two measures to describe the degree of similarity between two files, A and B,

the resemblance of A and B \( r_w(A, B) = \frac{|S(A, w) \cap S(B, w)|}{|S(A, w) \cup S(B, w)|} \) (2.16)

the containment of A in B \( c_w(A, B) = \frac{|S(A, w) \cap S(B, w)|}{|S(A, w)|} \) (2.17)

where \( S(A, w) \) is the set of features in file A. To construct the features, Broder tokenised the reference string using word or line boundaries, then defined a shingle of length \( w \) to be the substring containing \( w \) contiguous tokens. Fingerprints of the shingles, calculated using a recursive hash function, were stored as the set of features. The distance between \( A \) and \( B \) may be defined as \( d(A, B) = 1 - r(A, B) \), which obeys the triangle inequality \( d(A, C) \leq d(A, B) + d(B, C) \), and is a metric. This allows file similarity to be extended beyond pair-wise comparisons, such as the detection of clusters of similar files. For CBC, this is a more appropriate measure of similarity than previous metrics such as edit distance and Lewenstein distance.

An alternative estimate resemblance and containment is obtained by selecting the \( s \) smallest fingerprints to form a sketch of the file. The common regions in two files will generate the same shingles and fingerprints, and, depending on how many non-matching shingles are in the two files, the sketch is expected to contain some of the matching fingerprints.

Manber (1994) proposed the approximate fingerprinting technique for evaluating file similarity, and developed the SIF program to find all similar files in a file system. An approximate fingerprint of a file is a subset of the \( n \)-gram fingerprints from the file, which are calculated using a recursive hash function as described in Section 2.3. The \( n \)-gram fingerprints in the approximate fingerprint are selected...
using the anchor method described in Section 2.6. The approximate fingerprint is a set $F = \{x_1, x_2, \ldots x_m\}$ where $x_i$ is the fingerprint of the $n$-gram at the $i$-th anchor in the file. Files having approximate fingerprints which match for some threshold proportion $p = (F_A \cap F_B)/(F_A \cup F_B)$, such as 50%, are reported as being similar. Using an anchor density to generate approximately 4–5 fingerprints per kByte of data, the sorted list of anchors was stored on disk using approximately 5% of the space of the files. The algorithm was effective at identifying similar files, able to reliably match a file with a version having been approximately 50% changed with random edits.

## 2.9 Collection Compression

We now turn to the problem of compressing entire collections. Specifically, we are interested in schemes which allow random access to data in the collection, or at least pseudo-random access where the desired data is efficiently accessed by decompressing a larger block. This is an essential requirement if the collection is to be used for CBC, which requires the retrieval of strings from all parts of the collection. Random access is also necessary for many other applications, such as with text or genomic data, that retrieve files or records without decompressing the entire collection. Compressed collections may also have improved access times, compared to uncompressed storage of the data, as fewer reads from external storage are needed, and decompression is often faster than the additional external reads of uncompressed data (Yiannis and Zobel, 2007; Hoobin et al., 2011).

Furthermore, CBC can be used to compress a collection by successive steps of compression and insertion. To illustrate, consider an empty collection. The first file is compressed with respect to itself, then the compressed file is inserted into the collection. The next file is compressed relative to the file in the collec-
tion, then its delta file is inserted, and so on. The complexity of this and related schemes was investigated by Storer and Szymanski (1982). They assumed that compression is achieved by replacing a string with a pointer to a duplicate string elsewhere in the collection. When the pointers refer to the original (uncompressed) files, there is an $O(n)$ greedy algorithm, where $n$ is the total length of the collection files, to find the most compact compressed representation of the collection. However, as each file contains pointers to other uncompressed files, many other files may need to be decompressed in order to decompress a single file. This is avoided when the pointers refer to the compressed representations of the collection files, as is the case when using the iterative CBC procedure described above. However, finding the most compact representation of the collection using compressed pointers is NP-complete. Storer and Szymanski (1982) also consider several other restrictions on the pointers, including the use of recursive pointers that refer to compressed data which itself contains a pointer. Decompression with recursive pointers require multiple passes to resolve all of the pointers. They found that, when recursion is prohibited, the most compact representation of the collection is $O(\sqrt{n})$.

Consequently, solutions to the collection compression problem must be heuristic. One approach is to find a representative subset of the collection, from which an index is constructed and the remainder of the collection is compressed relative to the subset. Examples of this approach are Cannane and Williams (2002) for general-purpose data, and Kuruppu et al. (2012) for genomic data using a reference genome. However, our definition of CBC uses data from throughout the collection, so this approach is not within our scope. We now describe several systems which do employ techniques similar to CBC.

Douglis and Iyengar (2003) evaluated the effectiveness of delta encoding via resemblance detection (DERD) for compression of data collections. They com-
pressed collections of HTML files from web site crawls, and email collections of a number of individual people, using resemblance detection to identify similar files, then delta encoded pairs of similar files. First, duplicate files were eliminated from the collection. Duplicate files were identified by comparing their MD5 hashes. Next, compressed files were decompressed. Then resemblance detection and delta encoding was attempted. The resemblance detection followed the approach of Broder (1997) whereby features were generated by calculating fingerprints of the shingles (approximately 30 bytes long) from the files. These features were sorted, and the lowest 30 were retained as the sketch of the file. Similar files for delta encoding were identified as having a minimum number of common features in their approximate fingerprints. When there were multiple files sharing the same number of features, several pairs were delta encoded and the smallest delta was used, but an exhaustive search of all such pairs of files was not performed. Delta encoding was performed using vcdiff (Korn and Vo, 2002). Finally, files for which a similar file could not be found were self-compressed, again using vcdiff by delta encoding with respect to an empty file.

They tested the compression performance for all minimum common feature thresholds from 1 to 30, and observed a monotonic increase in compression as the minimum number increased, although the improvement beyond approximately 20 features was minimal. They also evaluated retention of 100 features in the sketches and found that the compression improvement was marginal compared to that achieved with 30 features. Overall, the compression achieved on the HTML collections was 2 to 3 times more than using self-compression alone. The results were not as dramatic for the email collections, and they observed that most of the compression achieved was due to a small number of large files.

The principal shortcomings of DERD were that (i) the number of pairs of files sharing a minimum number of features increases faster than $O(n)$, so the
approach of comparing the sizes of delta encoded pairs does not scale well; (ii) as each file had a fixed number of features in its sketch, larger files were less likely to be identified as being similar. Essentially, the feature density decreased as the file size increased, making it less likely that similar data could be identified. Also, DERD is unable to identify a file with high similarity to a fractional part of a larger file; it is only able to detect file similarity, and not file containment.

These disadvantages motivated the development of *redundancy elimination at the block level* (REBL) (Kulkarni et al., 2004). Each collection file is divided into content-defined chunks of approximately similar size, then it proceeds in a manner similar to DERD. First duplicate chunks are identified by comparing their SHA hashes, and eliminated. Then resemblance detection and delta encoding is attempted on the chunks. Finally, the remaining chunks were self-compressed. A further enhancement was the evaluation of two alternative procedures for selecting the pairs of similar chunks for delta encoding. In *FirstFit*, a chunk was arbitrarily selected as a reference chunk, then all other chunks having at least one matching feature were delta encoded respect to it. This is a fast process, although it is unlikely that it will achieve the best pairings of files. The alternative procedure, *BestFit*, sorted the chunks according to the greatest number of matching features with other chunks. These are the candidate reference chunks, and they are then evaluated by finding the number of other chunks which share a minimum number of features with them. These are then delta encoded with respect to the reference chunks. *BestFit* achieves better compression than *FirstFit*, and also uses a smaller number of reference chunks, however it has quadratic time complexity. As with DERD, they used collections of HTML files from crawling a number of web sites, and also email collections.

PRESIDIO is a system for efficient compression of data for archival storage (You et al., 2011; You, 2006). In several aspects, it follows a similar approach to
REBL, but it uses a larger range of features for identifying similarity, and provides a framework for adding new types of features in the future. The features used in PRESIDIO are:

1. **Digests**: strong hashes (SHA-1 or MD5) of whole files.

2. **Chunk digests**: strong hashes (SHA-1 or MD5) of file chunks. The chunks are generated using a weak hash (Rabin fingerprints) using the method of Manber.

3. **Chunk lists**: concatenation of the chunk digests from a file.

4. **Sketches**: Using the method of Broder, these contain a deterministic sample of fingerprints of shingles of chunks. The shingle fingerprints are generated using a weak hash function (32 bits).

5. **super-fingerprints**: fingerprints of permutations of shingles.

Each of these feature types is stored in a single virtual content-addressable store (VCAS) in which objects are retrieved by reference to their fingerprint. One consequence is that hash collisions are ignored, although the probability of a collision on the strong hash functions is insignificant compared to the probability of a disk error.

To add a new file to the archive, PRESIDIO attempts to match each of the feature types in the order of least to most computational effort. First, the digest for the entire file is calculated. If it already exists in the VCAS, then the file is a duplicate and no further compression needs to be performed. If not, the file is chunked, the chunk digests are calculated, and duplicates of chunks already existing in the VCAS are removed. The remaining chunks are shingled and their sketches calculated. Similar chunks are identified in the VCAS, and the new chunks are delta encoded with respect to these.
To summarize, in many circumstances, PRESIDIO achieves excellent compression. However there are two key aspects of the design which may limit its effectiveness and speed. First, de-duplication of chunks ignores potential matching data before and after the matching chunk. Finding the full extent of the matching data surrounding a matching chunk would allow the entire match to be encoded using less bits. Also, consecutive matching chunks could be joined into a single encoded reference.

Second, retrieving a file from the collection, which has been delta encoded, requires that its reference file is fully restored, which in turn requires that all predecessor reference files are fully restored. PRESIDIO takes some measures to limit this by selecting reference files with a short dependency graph where possible, and by only delta encoding using a single reference file. However, this retrieval process performs substantial effort which does not contribute to the end result; parts of the reference files are restored which are not needed for the requested file, or data is retrieved multiple times where it occurs in several reference files.

Bigtable by Google Inc (Chang et al., 2008) is an alternative approach which used a sampled hash index and delta encoding similar to Bentley and McIlroy (2001) as a preliminary filter before compression with an LZ-style compressor. Only a brief sketch of the algorithm is provided in Chang et al. (2008), but the approach was investigated in detail by Ferragina and Manzini (2010). They compressed two large collections (one was 440 Gbytes) of web pages with a variety of existing compression tools. They found that Bentley-McIlroy (BM) compression followed by LZMA compression of the delta file was the best combination, achieving a compression ratio of $<5\%$. They concatenated the collection files into a single long file in order to compress the collection as a whole, rather than as individual files. Nonetheless, the BM delta encoding step followed by stan-
standard compression was clearly superior to compression without it. They also achieved compression improvement by changing the order in which files were appended to the concatenation. Both of these observations suggest there were many long range matching substrings within the single file which standard compressor were unable to find.

2.10 Summary

We have reviewed string hashing and several algorithms which employ it for the purpose of collection-based compression. A recurring theme to emerge is that the behaviour of particular CBC implementation will substantially depend upon the type of bias in the input. The behaviour of the hash function for generating fingerprints depends on the short-range bias over the length of $n$-grams or chunks, while the existence of long-range similarity greatly influences the effectiveness of the delta encoding and compression algorithms. Much of the design process will involve validation against representative data.

Another theme is the poor worst-case bounds, and the use of heuristic algorithms. This is a necessity for the string hash function due to the impracticality of rehashing the collection. Similarly for delta encoding, the impracticality of the $O(N + M)$ algorithm for large collections and the use of sampling to reduce the fingerprint index size have led to the development of several heuristic algorithms. Finally, the need for random access into a compressed collection leads to the use of pointers to the compressed files, rather than the original files, where the compression problem is $\text{NP}$-complete.

Nonetheless, from the many algorithms described here, and the experience of the collection compression systems, it is clear that there are many heuristic $O(N)$ CBC algorithms which can be expected to achieve good compression.
Chapter 3

Collection-Based Compression

In Chapter 2 we described several algorithms and systems that can be adapted for CBC, or employed as components of a CBC system. However, as we describe in the following paragraphs, these schemes were designed for other purposes, and each has limitations in some way when used for CBC. In this chapter we implement and evaluate a new method of CBC that avoids these problems. It employs many elements from the existing approaches combined with a new procedure of delta encoding with respect to multiple reference files, and a fingerprint index maintained on external storage.

To motivate the design of our new scheme, we briefly recount how the existing approaches described in Chapter 2 are restricted in their ability to detect long-range matching substrings. Collection compression systems employ a CBC step to compress new files as they are added to the collection. Some of these systems, such as de-duplication systems (Zhu et al., 2008), partition the collection files into chunks, and do not extend matching substrings into adjacent chunks unless that entire chunk also matches. These systems identify matching strings using compare-by-hash, meaning that they do not retrieve the collection string to resolve collisions, instead relying on very long fingerprints to reduce the possibility
of collision to a negligible probability. As de-duplication systems do not retrieve
the data from the collection file, they are unable to extend a match forward or
backward, and consequently cannot find the full extent of a string match extend-
ing into neighbouring chunks. Each long match is encoded as multiple chunk
references, rather than as a single long reference, resulting in a less compact
encoding.

Collection compression schemes employing file similarity detection, such as
DERD (Douglis and Iyengar, 2003), encode the input with respect to the most
similar collection file, ignoring additional matches in other files. This was partially
remedied by first partitioning the collection files into smaller chunks (Kulkarni
et al., 2004, REBL) and then encoding with respect to the most similar chunk,
but there are still potentially many matching substrings from other chunks that
are not exploited, as well as matching substrings that span chunk boundaries.
PRESIDIO (You et al., 2011) employed sketches and super-fingerprints to address
the problem of matches spanning multiple chunks, but it restricted the number
of files referenced by the encoding in order to limit the dependency graph of new
files added to the collection, thereby ignoring additional matching substrings in
other files.

Differential encoding and compression utilities, such as xdelta or vcdiff, are
unsuitable as they do not maintain an index of the collection, instead reading the
entire collection to construct the index before compressing a file. Furthermore,
they view the collection as a single large file, so storing the index is only a partial
solution as the index needs to be entirely reconstructed when any collection file is
modified. A similar problem occurs using archive mode of standard compression
utilities, such as gzip and 7-zip: multiple files added to the archive at the same
time are compressed simultaneously, exploiting matching strings shared between
the files, but a file added to the archive at a later time is compressed only with
respect to itself, making no use of string matches elsewhere in the archive. In order to exploit matching data in the previous files, the entire archive must be decompressed, then re-compressed together with the new file.

In this chapter we present a new CBC scheme employing several techniques to overcome these problems. We investigate the effectiveness of our scheme using several algorithmic variations and parameter values, and identify configurations obtaining significant compression improvement. The investigation uses data from two application domains, web sites and genomics, where the benefit from CBC is particularly pronounced.

3.1 Overview of the CBC Scheme

The components of our CBC scheme are illustrated in Figure 3.1. Compression of the input file is via a two-step pipeline. The first step is a delta encoding, which produces a delta file that refers to matching strings in multiple collection files. The matching strings are located using the fingerprint index in memory, then the strings are retrieved from the externally stored collection files for collision resolution. The second step compresses the delta file with a conventional single-file compression utility, such as gzip, bzip2, or 7-zip. Decompression is achieved by reversing the pipeline: the compressed file is decompressed by a conventional utility to obtain the delta; then the COPY commands in the delta file are replaced by their original strings, which are retrieved from the collection files, producing the original input file. The implementation of our scheme is named cobald, a novel word beginning with “coba” from “COllection-BAseD compression”. When compared to existing delta compression schemes, cobald is most similar to the system of Bentley and McIlroy (2001), with the addition of a delta encoder able to refer to multiple reference files.
3.1. Overview of the CBC Scheme

Figure 3.1: Functional diagram of the collection-based compression scheme implemented by cobald. The pipeline has two stages: the encode process performs a delta encoding of the input with references to multiple collection files, then the resulting delta file is compressed using a standard compression utility. The encode process queries the fingerprint index in memory to find matching strings in the collection files, then retrieves the matching strings from the externally stored collection files to resolve fingerprint collisions.

The index is a sampled fingerprint index of \( n \)-gram references. Fingerprints are calculated using a rolling hash function. Index entries identify the location of the \( n \)-gram in the collection using two coordinates: the file identifier and the position within the file. The \( n \)-gram references inserted in the index are selected at fixed offsets in the reference file, at every \( n \)-th position, similar to the method of Bentley and McIlroy (2001). In a later section we evaluate content-defined selection of \( n \)-gram references. Fingerprint collision resolution is performed during index construction. Distinct \( n \)-grams with the same fingerprint (collisions) are added to the index using a linked list, while re-occurrences of an \( n \)-gram that is already in the index are discarded. Note that the use of the linked list at each index entry incurs a substantial memory cost. It is included in cobald so that the effect of various fingerprint lengths can be evaluated, and should not be employed in an optimised CBC system. The fingerprint length would be chosen so that collisions were rare, and we expect that discarding these few collisions
Chapter 3. Collection-Based Compression 95

from the index would not result in a noticeable decrease in compression.

The encoding operation is suboptimal, generating a larger delta than the greedy algorithm of Tichy (1984), for several reasons. As the index contains fingerprints of \( n \)-grams, matching substrings shorter than \( n \) bytes long cannot be found. Fingerprint sampling further increases the length matching substrings which are guaranteed to be found. Finally, discarding references to duplicate strings from the index in memory eliminates the capability of searching for the longest match in a greedy manner.

The encoder fingerprints each \( n \)-gram of the input file using a rolling hash function, and queries the fingerprint in the index. When a matching \( n \)-gram is found, the encoder inserts a COPY instruction into the delta. Variations of this “select first match” encoding algorithm were employed by Burns and Long (1997), Ajtai et al. (2002) and MacDonald (2000). It is faster than finding all of the matches at the current location in the input file and selecting the longest in a greedy manner, and there is high likelihood that, if a much longer alternative match exists in the collection, it will be found when the encoder moves beyond the current match and replace the shorter match by backward extension. A consequence of using select-first matching is that only the first instance of an \( n \)-gram in the index will ever be used, and there is no advantage in storing references to duplicates. That is why duplicate \( n \)-grams are discarded during index construction.

During encoding, identifying matching \( n \)-grams involves first finding the matching fingerprint in the index, then resolving collisions by retrieving the \( n \)-gram from the collection file and performing a byte-by-byte comparison. Then the match is extended backwards and forwards by further byte-wise comparison of the input file and collection file. The COPY instruction emitted by the encoder contains the full extent of the string match. This method was also employed by
### 3.1. Overview of the CBC Scheme

| collection file1: | ABCDEFGHIJK |
| collection file2: | ABCDLMNOPQRST |
| input file: | ABCDEFGHIUVWXYZOPQRSTCDEFGH |

**delta:**
- COPY(file1, position=0, length=9)
- ADD("UVWXY")
- COPY(file2, position=7, length=6)
- COPY(input file, position=2, length=7)

Table: Example of encoding a file with two collection files, file1 and file2. The index is constructed using fixed-offset sampling, so that every n-th n-gram of the collection files is inserted in the index. The example uses n = 4, and the indexed n-grams are indicated with square brackets, and also listed in the right-hand column. The third n-gram is a duplicate of the first, so it is discarded from the index in memory, which is indicated by the line through it. The third occurrence of **ABCD** is from the input file, so it is not discarded from the index as matches in the input file are preferred to matches in the collection. As input file is encoded, its n-grams are progressively indexed, again using fixed-offset sampling. Encoding proceeds by searching for each n-gram from input file in the index. The n-grams which are successfully matched are shown in bold. These matches are extended backwards and forwards then encoded in the delta as COPY commands. Note that the final COPY refers to a string earlier in the input file, even though **EFGH** is also indexed from file1, as the encoding operation prefers matches in the input file when there is a choice.

Bentley and McIlroy (2001) and Ajtai et al. (2002).

The n-grams from the input file are indexed as it is encoded, allowing matching strings later in the input to be encoded as COPY instructions referring to the input file, as well as to the collection files. This is similar to the delta compression approach of vdelta (Hunt et al., 1998). When the index contains n-gram references to both the input file and a collection file, the encoding uses the reference to the input file, in order to reduce the number of COPY instructions referring to the collection.
The delta file contains ADD instructions of verbatim data that were not matched with any collection data, and COPY instructions containing references to strings in the collection. The COPY references contain a collection file identifier and the position in the file of the first byte of the matching string, similar to the $n$-gram references in the index, as well as the length of the string match. Using two-dimensional references (file identifier and start position) to locate a string in the COPY instructions and the index allows the addition or modification of collection files without needing to reconstruct the entire index, or to re-encode all existing delta files. The encoding operation is illustrated in Figure 3.2.

### 3.2 Detailed Description

We now describe the remaining details of the CBC scheme, along with some implementation details of cobald. Note that cobald contains several elements to support comparison of algorithms, and parameters that impose some performance cost, both in speed and memory use. The previously described use of linked lists in the index for fingerprint collisions, which would be removed in an optimised system, is an instance of this. Also, there are many aspects of cobald that could be made more efficient but are beyond the scope of this investigation, such as compact coding of integers storing string locations and lengths. Those aspects that affect the compression results are noted in the following description. Figure 3.3 shows the worked example from Figure 3.2, with example values for the data structures used in cobald.

**Collection files** The collection files are stored individually on external disk. They are stored uncompressed as the encoder requires random access to the files to make comparisons with matching strings in the input.
Figure 3.3: Example of encoding showing the values written by cobald in the index, match list, and delta file. The example uses the same collection files and input file as Figure 3.2. The n-gram index uses fixed-offset sampling with $n = 4$. The n-grams inserted into the index are underlined with square brackets, and are also listed in the right-hand column. The third n-gram, indicated with a line through it, is a duplicate of the first, and is discarded from the index in memory. There is a third occurrence of the n-gram ABCD, but it is not discarded as it occurs in the input file, which is preferred over n-grams from the collection files, and is inserted into $I_{\text{input}}$. There are two distinct n-grams having fingerprint 3123, so the second one has collision ID = 1. During encoding, three matching n-grams are found in the index, indicated using bold text. The full matches, after backward and forward extension, are shown in the match list. The delta contains these matches as COPY blocks, as well as an ADD block for the unmatched text UVWXY between the first and second COPY blocks. The mapping of the COPY and ADD blocks to the original files is indicated by the pink arrows.
**Fingerprint index file**  An index file stores the fingerprints and locations of the indexed *n*-grams. The index file contains an entry for each collection file, comprised of the file name as a null-terminated string, followed by a sequence of triples, \((\text{fingerprint}, \text{collision ID}, \text{location})\), where *fingerprint* is the hash value of the *n*-gram beginning at *location* in the file. The *collision ID* is an integer in the range 0 to 254 assigned to different *n*-grams having the same fingerprint (collisions). The triples are fixed-length records. For most experiments, the lengths used were 8 bytes for the location, 8 bytes for the fingerprint, and 1 byte for the collision ID. The end of the list of triples is indicated by an additional triple containing the special value of 255 for the collision ID.

The index file is constructed in a single pass by appending the fingerprints and position references as each collection file is read. Adding a file to the collection simply involves appending its *n*-gram references to the existing index file. Collision resolution is performed as the index file is constructed, which requires that the index resides in memory as construction proceeds. Before appending a new *n*-gram reference, the existing index entries with a matching fingerprint are read from their collection files and compared with the new *n*-gram. If the new *n*-gram matches one from the index, it is assigned the same collision ID. If the new *n*-gram does not match any of the existing entries, it is a collision and is assigned a new collision ID. Consequently, all occurrences of an *n*-gram in the index file will have the same collision ID.

The index file format was chosen for simplicity and flexibility, and we have not sought a more efficient coding, although the size of the index file is a consideration when evaluating the compression effectiveness. Furthermore, some entries in the file are redundant in certain circumstances, but are retained to allow comparison with other situations. For instance, when *n*-grams are indexed at fixed offsets from the beginning of the collection file, the position can be inferred from the
sequence of entries, so there is no need to record it. However, with content-defined selection of \( n \)-grams, the position cannot be calculated.

**Fingerprint index in memory** Before encoding a file, the fingerprint index is constructed in memory by reading the index file. A subset of \( n \)-gram references in the index file are loaded as references to duplicate \( n \)-grams are discarded. When a duplicate \( n \)-gram reference, which is identified by matching fingerprint and collision ID with an existing entry, is read from the index file during construction, it can either be discarded or be used to replace the previous \( n \)-gram reference in the index. In all of our experiments, duplicate \( n \)-gram references were discarded, as we expected this to produce delta files having references to fewer collection files. However, the choice does not arise in Chapter 4 where duplicate strings have been removed from the collection files. The structure of the index, illustrated in Figure 3.4, is a chained hash table of pointers to linked lists of \( n \)-gram references. The index size is linear with the number of \( n \)-grams, and the index has constant average access time when the hashes are uniformly distributed. To allow the encoding algorithm to find matches earlier in the input file, a second index, having the same structure as the collection index, stores the \( n \)-gram references from the file being encoded. When encoding, the input index is queried first, and the collection index is only queried when the input index query was unsuccessful, so that matches to the input file have priority, and the number of COPY instructions referencing the collection are reduced. Note that an alternative could be to insert the input \( n \)-gram references into the collection index during encoding, replacing any duplicate entries from the collection, however these references would need to be removed from the collection index before encoding a second file.

The index is implemented in **cobald** using the `std::unordered_multimap` data structure from the GNU C++ standard library. As elements are inserted
Chapter 3. Collection-Based Compression

Figure 3.4: Illustration of the index data structure. $f_p$ is the n-gram fingerprint, and $h(f_p)$ is the hash table index derived from the fingerprint. Typically, $h$ is the least significant bits of $f_p$ such that the table has the desired occupancy. A table entry contains a pointer to a linked list whose elements contain the fingerprints giving the value $h(f_p)$. Each of these entries contain pointers to a list of references to n-grams having that fingerprint.

and the table becomes full, the array is dynamically re-sized by copying it to an array double in size. The table elements are fixed-sized records having 1 byte for the collision ID, 4 bytes for the file ID, and 8 bytes for the position. The fingerprints are stored in 8 bytes of memory.

**Compress operation**  In the compress operation, a standard compression utility such as gzip, 7-zip, or bzip2 can be used to compress the delta file to produce the final compressed version of the input file.

**Delta file:** The output from the encoding operation is a delta file having the following format. First is an array of the names of all collection files referred to by the COPY commands in the delta. The delta commands follow as a sequence of blocks. A *COPY block* describes a COPY instruction to locate the matching
string in a collection file and consists of a triple \((\text{file id}, \text{length}, \text{offset})\) where \text{file id} is the index of the file name array in the header, \text{length} is the number of bytes of the matching string and \text{offset} is the position in the file of the first byte. An \textit{ADD block} describes an ADD instruction, and consists of a marker to distinguish it from a \textit{COPY block}, followed by an integer indicating the length of the ADD instruction, followed by \text{length} bytes of verbatim data. The first two fields of the \textit{COPY} and \textit{ADD} blocks are the same length, with \text{file id} = -1 marking the start of an \textit{ADD} block, and \text{file id} = -2 indicating a \textit{COPY} block which refers to earlier in the input file. \text{File id} is a 4 byte integer, and \text{length} and \text{offset} are 8 bytes to accommodate very large files.

The \texttt{cobald} delta file format is designed for simplicity rather than efficiency. When an input file is encoded with respect to a single reference file, we are able to convert it to the standard VCDIFF format (Korn et al., 2002). We found that the VCDIFF format delta was up to 5\% smaller than the \texttt{cobald} delta file. The principal aspects of VCDIFF format enabling improved compression are byte coding of match reference offsets and lengths, and grouping the verbatim data for ADD commands into a contiguous block. VCDIFF-ed delta files were used for the trials in Section 3.10.

**Encoding algorithm:** The encoding operation is described in Algorithm 7. Encoding is done in a single pass through the input file, searching for matching fingerprints in the index. The fingerprint of the \textit{n}-gram at the current position is found using a rolling hash function, and queried in the index. When a fingerprint match is found, collision resolution is performed: the \textit{n}-gram reference in the index is retrieved from the collection file, or earlier in the input file, and compared byte for byte with the \textit{n}-gram from the input file. A successful match is extended forwards and backwards to find the complete matching substring, which is then
appended to the *match list*. As a result of backward extension, it is possible that the new match overlaps matches found previously, and their entries in the match list are amended. Any previous match which is entirely contained within the new match in the input string is removed from the match list. If the new match begins part-way through a previous match, then the overlapping part is removed from the new match so that it begins immediately following the previous match. Finally, if the length of the new match less than a minimum value (8 bytes in `cobald`) it is discarded as COPY block would be longer than the substring which it is encoding. Then the search resumes in the input file from the end of the match. As the search proceeds through the input file, every *n*-th *n*-gram is inserted in input file index. Once the search of the input file is complete, the delta file is created by writing the references from the match list as COPY blocks and inserting ADD blocks in the gaps between matches.

The match reference record in the match list is of fixed length containing four items: the position of the first matching byte in the input file (8 bytes); the position of the first matching byte in the collection file, or earlier in the input (8 bytes); the length of the match (8 bytes); and an identifier indicating which collection file contains the match (4 bytes). When encoding large files, the match list can become a large data structure.

**Hash function** As each *n*-gram from the input file may have a matching reference in the index, the encoder calculates the *n*-gram fingerprint and performs an index query at each position in the input file, unless it is extending forward. A rolling hash function is most suitable in this situation. `Cobald` uses the division by irreducible polynomial hash algorithm of Cohen (1997). Unless stated otherwise, the hash length is 62 bits, using the irreducible polynomial \(x^{62} + x^{48} + x^{18} + x^{16} + x^9 + x^2 + 1\) (Živković, 1994).
Algorithm 7 Encoding with fixed-offset sampled fingerprint index

**Data:** \( R \), the set of collection files;

- \( Q \), the input file of length \( N \) (bytes);
- Index file of sampled \( n \)-gram fingerprints and references from files in \( R \).

**Parameters:**
- \( n \), the length of \( n \)-grams (bytes);
- \( b \), the maximum backward extension distance (bytes).

**Result:** A delta file encoding \( Q \) in terms of \( R \).

1: Construct \( I_R \), the index of fingerprints from \( R \) in memory.

Each \( n \)-gram reference in the index file is inserted into \( I_R \) if an entry with the same fingerprint and collision ID has not previously been inserted.

2: Create \( I_Q \), the empty index of fingerprints from \( Q \) in memory.

3: Create \( L \), the empty list of string match references found during encoding.

4: \( i \leftarrow 0 \).

Set the position, \( i \), of the encoding operation in \( Q \) to the beginning.

5: while \( i \leq N - n \) do

6: \( f_p \leftarrow \text{Hash}(Q[i, i + n - 1]) \).

Find \( f_p \), the fingerprint of the \( n \)-gram at position \( i \) in \( Q \).

7: \( M, j \leftarrow \text{QUERYINDEX}(f_p, i) \).

Query the index for the \( n \)-gram. When a matching \( n \)-gram is found, \( M \) is a reference to the file containing the match, and \( j \) contains the offset of the first byte of the match in \( M \).

8: if a match is found then

9: \( i_b \leftarrow \text{EXTENDBACKWARD}(i, M, j) \), and

\( j_b \leftarrow j - (i - i_b) \).

Set the positions \( i_b \) in \( Q \), and \( j_b \) in \( M \), to the beginning of the matching string.

10: \( i_e \leftarrow \text{EXTENDFORWARD}(i + n, M, j + n) \), and

\( l \leftarrow i_e - i_b \).

Find the position, \( i_e \), in \( Q \) of the non-matching byte immediately following the matching string, and the length, \( l \), of the match.

11: \( \text{APPENDMATCH}(i_b, M, j_b, l) \).

Append the matching string to \( L \).

12: \( i \leftarrow i_e \).

Move the encoding position past the end of the match.

13: else

14: if \( i \mod n \equiv 0 \) then

15: \( \text{INSERTINDEX}(f_p, i) \).

Insert the fingerprint of every \( n \)-th \( n \)-gram from \( Q \) into \( I_Q \).

16: \( i \leftarrow i + 1 \).

17: \( \text{GENERATEDelta}( ) \).
Algorithm 7 ...continued.

18: function ExtendBackward\((i, M, j)\)

Find the beginning of the matching string.

19: Starting at positions \(x = i\) in \(Q\) and \(y = j\) in collection file \(M\), decrement \(x\) and \(y\) until either \(Q[x - 1] \neq M[y - 1]\), the beginning of one of the files is reached, or until \(i - x > b\), the limit on backward extension.

20: Return \(x\), the position in \(Q\) of the first symbol of the matching string.

21: function ExtendForward\((i, M, j)\)

Find the end of the matching string, while inserting \(n\)-grams into \(I_Q\).

22: Starting at positions \(x = i\) in \(Q\) and \(y = j\) in collection file \(M\), increment \(x\) and \(y\) until either \(Q[x] \neq M[y]\), or the end of \(Q\) or \(M\) is reached.

23: At each position, if \(x + 1 \pmod{n} \equiv 0\) then

24: \(f_p \leftarrow \text{Hash}(Q[x-n+1, x])\), and

\(\text{InsertIndex}(f_p, x - n + 1)\).

25: Insert every \(n\)-th \(n\)-gram fingerprint into \(I_Q\). Note that \(x\) is the position of the last symbol in the \(n\)-gram.

26: Return \(x\), the position in \(Q\) following the end of the matching string.

27: function QueryIndex\((f_p, i)\)

Search for the fingerprint, \(f_p\), in the input index, \(I_Q\). Then if a matching \(n\)-gram is not found, search for \(f_p\) in the collection index, \(I_R\). When an entry for \(f_p\) is found, confirm that it is not a hash collision by retrieving the indexed \(n\)-gram \(M[j, j + n - 1]\) from the collection file \(M\) (or from \(Q\) if the match is in \(I_Q\), in which case \(M = Q\)) and perform a byte-wise comparison with \(Q[i, i + n - 1]\).

28: Return \(M, j\), or FALSE if a matching string is not found.

29: function AppendMatch\((i, M, j, l)\)

Resolve overlaps with previous string matches resulting from extending backwards, then append the adjusted string match to \(L\). The matching string of length \(l\) starts at position \(i\) in \(Q\) and position \(j\) in collection file \(M\). Let \(i_z\) and \(l_z\) refer to the starting position in \(Q\) and the length of the last matching string in \(L\).

30: while \(i_z \geq i\) do

31: if \(i < i_z + l_z\) then

32: \(j \leftarrow j - i + (i_z + l_z)\),

\(l \leftarrow l + i - (i_z + l_z)\),

\(i \leftarrow i_z + l_z\).

33: Append the adjusted matching string reference \((i, M, j, l)\) to \(L\).
3.3 Independence of Recursive Hash Functions

Recall that Lemire and Kaser (2010) showed that rolling hash functions are pairwise independent at best. Consequently, it would seem that a rolling hash function is a poor choice for a fingerprint index. For an index containing millions of fingerprints, a pairwise independent function is expected to have significantly more fingerprint collisions than would be obtained from a stronger hash function.

However, we now show that the pairwise independence result of Lemire and Kaser (2010) only applies to $n$-grams that substantially overlap. This situation can be readily avoided by ensuring that fingerprints are sampled from $n$-grams separated in the input file by a minimum distance, and there is no reason to conclude that they have reduced independence.

**Theorem 5.** The theorem of Lemire and Kaser (2010), that hashes from a rolling hash function can be pairwise independent at best, is not valid when the $n$-grams overlap in fewer than $n/2$ positions.

*Proof.* We begin by restating the proof of Lemire and Kaser (2010). Let $h$ be a 3-wise independent rolling hash function onto $\{0, 1, \ldots, M - 1\}$ chosen randomly from a function family, and having the iterative relation $h(u_i = s_is_{i+1}\ldots s_{i+n-1}) = F(u_{i-1}, s_{i-1}, s_{i+n-1})$. For 3-wise independence, $\Pr(h(u_1) = v_1, h(u_2) = v_2, h(u_3) = v_3) = 1/M^3$. LK then showed that 3-wise independence was not satisfied for a class of input strings containing repeated bytes. Let the input be $a^nbb$. The 3-way independence calculation for this input string such that...
the adjacent $n$-grams have the same hash values $v_1 = v_2 = v_3 = v$ is

$$\Pr \left( h(a^n) = v \land h(a^{n-1}b) = v \land h(a^{n-2}bb) = v \right)$$

$$= \Pr \left( h(a^n) = v \land F(v, a, b) = v \land F(v, a, b) = v \right)$$

$$= \Pr \left( h(a^n) = v \land F(v, a, b) = v \right)$$

$$= 1/M^2$$

$$\neq 1/M^3 \quad \text{as required for 3-wise independence.}$$

So, when $h(a^n) = h(a^{n-1}b)$, then $h(a^{n-2}bb)$ has no independence.

This argument is also valid if the input is extended to $a^nbbbb$, and we select every second $n$-gram. To obtain the next selected hash value, the iteration relation twice is applied twice. When the first and third $n$-gram hashes are equal, $h(a^n) = v = h(a^{n-2}bb) = F(F(v, a, b), a, b)$, then the fifth $n$-gram hash has the same iterative expression as the third, $h(a^{n-4}bbbb) = F(F(h(a^{n-2}bb), a, b), a, b) = F(F(F(v, a, b), a, b) = v$.

This holds true for selection separation distances up to $n/2$, but, for larger separations, the first symbol of the final $n$-gram is past the end of the first $n$-gram so the iterative expressions for the middle and final $n$-gram hashes must differ. To express this mathematically, let $F_j(x, y) = F_j(F_{j-1}(x, y), x, y)$, $F_0 = v$ represent the iterative application of the rolling hash function $j$ times. Then, with input $a^nbbbc$ and $n$ even, when selecting every $(n/2+1)$-th $n$-gram the hash of the middle $n$-gram is $F_{n/2+1}(a, b)$. The next $n$-gram selected will be $n + 2$ bytes from the start of the first $n$-gram. The hash of this final $n$-gram is $F(F(F_n(a, b), b, c), b, c), b, c)$, which differs from the expression for the hash of the middle $n$-gram.

As we described in Section 2.1, both pseudo-randomness of the hash function and randomness in the key set contribute to the uniformity of the hash distri-
bution. The theorem of Lemire and Kaser (2010) is a succinct illustration of this observation. When the keys are biased by the condition that they overlap, the uniformity of the hash distribution can be no better than that obtained by pairwise independence. But when the bias of the keys is reduced by removing the condition that they overlap, the uniformity of the hash values is no longer limited.

3.4 Complexity of the Encoding Algorithm

In Chapter 1 we stated a requirement that the compression time complexity should be linear with the size of the input file. This can be readily obtained by an algorithm that makes a constant number of passes over the input, and performs a constant time operation on each input byte. However, the coballd encoding algorithm performs backward extension of string matches during which some input bytes are read multiple times.

It is clear that the encoding algorithm is linear if it did not extend backwards. There will be at most $2N$ reads from the input file $Q$, first to find the matching strings, and the second time to insert the ADD instructions in the delta file. There will be at most $N - n - 1$ constant-access-time index queries. Finally, there will be at most $N$ bytes read from the externally stored collection files to resolve collisions and extend forwards, assuming that the $n$-gram fingerprint length is sufficiently long to ensure an insignificant number of fingerprint collisions.

However, the inclusion of backwards extension in the encoding algorithm allows the possibility that multiple comparisons will be performed on input bytes, and the algorithm is quadratic in the worst case.

**Theorem 6.** Encoding the input file $Q$ of length $N$ using backward extension requires $O(N^2)$ byte comparisons on the input file with $O(N^2)$ bytes read from the
collection. The bounds are tight.

Proof. We construct a collection such that an encoding requires $O(N^2)$ comparisons. Let the input file be the string be $G_1G_2G_3\ldots G_{N/n}$, where $G_i$ are unique $n$-grams. The collection files containing these $G_i$ are

- file 1: $*G_1*$
- file 2: $*G_1G_2*$
- file 3: $*G_1G_2G_3*$
- \[\ldots\]
- file $N/n$: $*G_1G_2G_3\ldots G_{N/n}*$

The * indicate data that is not in the input file. The files were added to the index in the order file 1, file 2 \ldots file $N/n$ so that each file contains a prefix of the input file, and that prefix contains the prefix in the previously indexed collection file. Finally, file $N/n$ contains the entire input file. The $n$-gram references inserted into the index are $(G_1$ in file 1), $(G_2$ in file 2), $\ldots (G_{N/n}$ in file $N/n)$. Note that, during index construction, references to $G_1$ in file 2, file 3 $\ldots$ are discarded as the index already contains a reference to $G_1$ in file 1.

The encoding operation proceeds by finding $(G_1$ in file 1), then $(G_2$ in file 2) and extending backwards through the $G_1$ in file 2, then finding $(G_3$ in file 3) and extending backwards through $G_2$ then $G_1$ in file 3, and so on. The sequence of comparisons on the input file are

$G_1, G_2, G_1, G_3, G_2, G_1, \ldots, G_{N/n}, G_{N/n-1}, \ldots, G_2, G_1$.

The total number of byte comparisons is a multiple of a triangular number, $(1 + 2 + 3 + \ldots + N/n)n = (N(N/n + 1))/2$, which is $\Theta(N^2)$ as $n$ is constant. This is also the number of bytes read from the collection files.
Finally, observe that the number of index queries is $O(N)$. Also note that this proof ignores the possibility that, with very large $N$, the number of unique $n$-grams will be exhausted, and it will not be possible to construct the collection files as described. However, the size of such a collection is far beyond practical consideration.

The encoding-time asymptotic cost cannot be improved by changing the index insertion policy, such as replacing an $n$-gram reference in the index when a duplicate is found, rather than discarding it. The encoding in the proof will again result in quadratic comparison when the collection files are inserted in the reverse order.

An encoding algorithm with linear complexity can be obtained by simply imposing a limit on the length of backward extension.

**Theorem 7.** Encoding the input file $Q$ of length $N$ using backward extension no longer than $b$ bytes requires $O(N)$ byte comparisons on the input file with $O(N)$ bytes read from the collection.

**Proof.** The highest number of comparisons will occur when no $n$-grams are able to be extended forward, but all are extended backwards to the maximum extent. The number of comparisons, $c$, for each such $n$-gram is $c < n + b$, with backward extension limited to less than $b$ bytes at the beginning of the input file. The total number of such $n$-grams is $N/n$, so the total number of comparisons is $cN/n < (n + b)N/n$ which is $O(N)$.

Note that the proof does not establish a limit on the maximum backward extension $b$. It can be arbitrarily large, although, to have a practical effect, it must be less than the collection size. Also, we observe that the proof invokes a collection having a pathological structure unlikely to occur in practice. In **cobald**, the maximum length of backwards extension is $10^7$ bytes.
The memory required by the encoding algorithm is $O(M + N)$, where $M$ is the total length of the collection files. The largest data structure is the fingerprint index. An $n$-gram reference in the index requires constant space. Specifically, it uses 1 byte for the collision ID, 4 for the file ID, 8 for the position and 8 for the pointer to next list item, for a total of 21 bytes. Additionally, each $n$-gram reference list is pointed to by a list item having 8 bytes for the fingerprint, 8 bytes for a pointer to the $n$-gram reference list, and 8 bytes for a pointer to the next fingerprint item. Finally, there are 8 bytes for the entry in the table of pointers to the fingerprint lists. The overall total is 53 bytes per $n$-gram reference. References to $n$-grams are stored from the collection files and from the input, so the maximum number of $n$-gram references stored in the index is $(M + N)/n$, and the total memory for the index is $53(M + N)/n$.

The other data structure of significant size is the list of matches found during encoding. It is a linked list using $O(N)$ memory. Each match reference is a constant size of 28 bytes, and the maximum number of matches is $N/n$.

The $O(M + N)$ memory use exceeds the requirement we stated in Chapter 1 that the space complexity should be sublinear with the collection size. The requirement is achieved, for suitable data having high similarity, in Chapter 4 by indexing delta files instead of the original collection files. This ensures that each byte in a long string that is duplicated in the collection is represented in the index by a single $n$-gram reference. Discarding duplicate $n$-grams from the index, as we do in this chapter, is not sufficient to achieve this reduction in index size because the duplicate string is likely to be sampled at different offsets. As the index contains $n$-gram references located at fixed multiples of $n$, when the duplicate string is shifted relative to the start of the file, the $n$-grams inserted from the duplicate will not match those from the first occurrence, and will be inserted into the index as well. In the worst case, a byte in a string duplicated numerous times in the
3.5 Performance Evaluation

To evaluate the performance of cobald, we use two measures to quantify the compression achieved when encoding a file. The effectiveness of the encoding operation is evaluated using the percent not matched, that is, the number of bytes in delta file ADD blocks as a percentage of the un-encoded input file size. This is the proportion of the file that was not matched with other locations in the collection, or some earlier position in the input file. Note that the size of the unmatched data is less than the size of the delta file as it does not include the additional data to encode the COPY blocks. It allows comparison of the performance of the encoding algorithm independent of the efficiency of the delta file format.

The other performance measure used is comparisons of compressed delta sizes. Several file types are used for comparison: the uncompressed delta file, the gzip-ed delta file, and the 7-zip-ed delta file. These are compared to the un-encoded uncompressed input file, the gzip-ed input file and the 7-zip-ed input file. Comparisons are expressed as a percentage of the un-encoded uncompressed input file size. We can then obtain a measure of the compression improvement achieved by using cobald as a preliminary step before conventional compression: the difference between the sizes of the compressed (by gzip or 7-zip) delta files compared to the size of the compressed input files. When reporting the compression of a data set containing multiple files, the sizes of all files in it are summed to provide a single total.
The compressors were configured with their best practical options to achieve high compression:

```
7z  -m0=lzma  -mx=9  -mfb=64  -md=32m  -t7z  -l 
gzip --best 
bzip2 --best 
xz  --best.
```

The trials were run on a Dell Optiplex 755 desktop computer, having a dual-core Intel Core(tm)2 Duo E6850 3.00 GHz Processor, which is a 32 bit architecture, and 4 Gbytes of RAM. The system had two ATA hard disks of 1 Tbytes and 250 Gbytes size. For all of the experiments, the collection files were stored on the larger drive and the input files were read from, and output files written to, the other drive. The operating system was the Ubuntu 12.04 distribution of the Gnu/Linux kernel version 3.2.0-60-generic-pae. The systems has two hard disk drives: the collection and input files were stored on one, while the index file and output files were written to the other.

### 3.6 Data Collections

The experiments, and those in Chapter 4, used two data collections: *Web snapshots* and *Genome*. Each collection contains several *data sets*, each containing multiple files. Each data set was created sequentially as a revision of the previous data set. Therefore a substantial proportion of the data is unchanged between data sets, similar to what might be expected from a sequence of archival snapshots. Table 3.1 summarizes the properties of the data sets.

**Web snapshots** The *web snapshots* collection is eight web crawls of a media web site (abc.net.au) taken one week apart over a two-month period (Ali, 2008). It contains only the (untruncated) html files; media files such as images have been
Table 3.1: Summary of data collections

<table>
<thead>
<tr>
<th>Collection</th>
<th>Number of data sets</th>
<th>Data set size (Mbytes)</th>
<th>Collection size (Mbytes)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Web snap-shots</td>
<td>8</td>
<td>6,450–7,850</td>
<td>55,714</td>
</tr>
<tr>
<td>Genome</td>
<td>21</td>
<td>1,100–1,400</td>
<td>26,141</td>
</tr>
</tbody>
</table>

Genomes The Genome data set comprises releases 36 to 56 of the Drosophila melanogaster genome from the Ensembl project (ftp.ensembl.org). The data consists of nucleotide sequences with annotations describing aspects such as protein and RNA expression. Each release is an update of the data from the previous month incorporating new results from the genomic research community. The files are individually gzip-ed on the Ensembl ftp server. The collection is chosen as it has multiple versions of the genome of a single organism, which is a common scenario in research and clinical situations. Also, the variation between releases could be viewed as a proxy for the genomes of multiple individuals from the same species.

The files are in three common genome file formats embl, genbank, and fasta. These are plain text files formatted with white space and offset counts to facilitate human readability. Unfortunately, the visual formatting confounds delta encoding: for example, the insertion of an additional nucleotide near the start of a sequence causes changes to the text of the remainder of the sequence as the position of white space or offset counts is adjusted. We therefore strip the files of their visual formatting before analysis, by transforming each nucleotide sequence into a single long line of text without any interleaved space or newline characters. The files were restored to their original format by re-inserting the stripped formatting after retrieving files from the collection. For comparison of compression performance, the gzip and 7-zip file sizes reported are for the stripped input discarded.
files. Interestingly, stripping the genome files improved the compression achieved by gzip and 7-zip by approximately 15%.

### 3.7 Results: Fixed-Offset Selection

We evaluated our CBC scheme with fixed-offset selection of $n$-grams by encoding and compressing some data sets from the Web snap-shots and Genome collections, in a range of conditions.

**Encoding test:** First, we evaluated the effectiveness of encoding with a range of $n$-gram lengths, $n = 64, 256, 1024, 4096, 16384$ and $65536$. The *release-56* data set from the Genome collection was encoded by cobald with *release-55* in the index, and the *week2* data set from Web snap-shots with *week1* in the index. By populating the index with the preceding data set, we ensure that there are numerous string matches in the input files, and the test explores how much of this matching data is found by the encoder. As a baseline, we also encoded the files with an empty index. In this situation, the encoder may still find string matches to earlier in the input, but significantly less data should be matched. By comparing the outcomes from empty and filled indexes, we are able to evaluate the effectiveness of encoding with the index.

The results, shown in Table 3.2 and Figure 3.5, are presented as the amount of data that was not matched, which is found by summing the lengths of the ADD instructions in the delta files. Each data set contains several files that were encoded individually. The ADD lengths from each delta file were summed to produce a single total for the whole data set. These are expressed as a proportion of the total size of the un-encoded input files. The percent not matched is displayed in Figure 3.5, rather than the percent-matched, as the values close to zero can
3.7. Results: Fixed-Offset Selection

Table 3.2: Percentage of input files that were not matched by the encoding operation using empty and filled indexes for a range of \(n\)-gram lengths (left-hand column). The remaining columns are the sum of the ADD instruction lengths in the delta files, expressed as a percentage of the total input file sizes. A percent not matched of 100% indicates that no compression was achieved. The files from Web snap-shots collection, week2 were encoded with an empty index, and then with the files from week1 in the index. The files from Genome collection, release-56 were encoded with an empty index, and then with the files from release-55 in the index.

<table>
<thead>
<tr>
<th>(n)</th>
<th>Web snap-shots, week2</th>
<th>Genome, release-56</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>empty</td>
<td>week1</td>
</tr>
<tr>
<td>65 536</td>
<td>99.98</td>
<td>98.07</td>
</tr>
<tr>
<td>16 384</td>
<td>96.07</td>
<td>71.65</td>
</tr>
<tr>
<td>4 096</td>
<td>43.03</td>
<td>9.44</td>
</tr>
<tr>
<td>1 024</td>
<td>36.99</td>
<td>3.16</td>
</tr>
<tr>
<td>256</td>
<td>24.89</td>
<td>1.51</td>
</tr>
<tr>
<td>64</td>
<td>16.58</td>
<td>0.68</td>
</tr>
<tr>
<td>16</td>
<td>6.29</td>
<td></td>
</tr>
</tbody>
</table>

be clearly shown on a logarithmic scale. Some initial trials were performed with \(n = 16\) but we considered it to be impractical as the resulting indexes were very large and encoding was much slower than \(n = 64\).

We can see from Figure 3.5 that the encoding improves monotonically as the \(n\)-gram length decreases. This is to be expected as encoding with a shorter \(n\)-gram will find all of the strings matches found using a longer \(n\)-gram, as well as some additional string matches too short to be found with the longer \(n\). The longer \(n\)-gram lengths were ineffective. Web snap-shots week2 found almost no matching data with \(n = 65 536\), and Genome release-56 saw insignificant improvement between \(n = 65 536\) and 4 096. The shorter \(n\)-gram lengths are increasingly effective, and achieve a large compression improvement compared to compressing with an empty index.
(i) Encoding Web snapshots, week 2.

![Figure 3.5: Encoding effectiveness with empty and filled indexes using a range of n-gram lengths. The percent not matched is the sum of the lengths of ADD instructions in the delta files, expressed as a percentage of the total input file sizes. A low percent not matched indicates high compression. (i) Encoding files from the Web-snap shots collection, week 2 and (ii) encoding files from the Genome collection, release-56. The blue line in each chart is the result from encoding with an initially empty index. The pink line is encoding after the preceding data set, (i) Web snapshot, week 1 and (ii) Genome release-55, were added to the index. A logarithmic scale is used on the y-axis to clearly show the compression improvement with shorter n-grams. The data is listed in Table 3.2.](image)

(ii) Encoding Genome, release-56.
3.7. Results: Fixed-Offset Selection

**Compression test:** We evaluated the performance of the full encoding and compression pipeline by compressing the delta files obtained by encoding Web snap-shots week2 using **cobald** with week1 in the index, and by encoding Genome release-56 with release-55 in the index. The compressed file sizes are shown in Figure 3.6 and Table 3.3. The overall compression is every effective. With $n = 256$, Web snap-shots week2 is compressed to 0.2% of the input file size by **cobald** and **7-zip** together. The Genome release-56 files were compressed to 1.5% of the input file size. The contribution of **cobald** is the gap between the input file + compressed (dashed line), and the corresponding delta + compressed (solid line). We can see that this increases as the $n$-gram length decreases, indicating that **cobald** encoded more matching strings that would not have otherwise been found by the second-stage compressor. Using **cobald** with $n = 256$ improved the compression obtained by **7-zip** 22.9 times for the Web snap-shots collection, and 11.5 times for the Genome collection.

The contribution of the compression second stage can be seen as the gap between the uncompressed delta, shown as the solid yellow line, and the compressed delta (blue or red line). This gap is largely constant, indicating that the effectiveness of the second-stage compressor is independent of the $n$-gram length used to obtain the delta, and of the amount of compression achieved by **cobald** in the first stage. In particular, when **cobald** achieves a high compression, the effectiveness of the second stage is not adversely affected, and we can conclude that **cobald** is finding string matches that a conventional compression utility does not. For example, for Web snap-shots week 2 encoded using $n = 256$ bytes, the delta file is 2% of the input file size, which is reduced a further 10 times to 0.2% when compressed with **7-zip**. We also performed some tests using **bzip2 --best**, which achieved better compression than **gzip** but slightly less than **7-zip**, and **xv --best**, which achieved compression very close that achieved by **7-zip**.
(i) Encoding Web snap-shots, week 2

![Graph showing compressed file sizes for Web snap-shots week 2](image)

(ii) Encoding Genome, release-56

![Graph showing compressed file sizes for Genome release-56](image)

Figure 3.6: Compressed file sizes, expressed as a percentage of the input file size, using a range of n-gram lengths. The dashed lines represent the sizes of the input files compressed without using cobald. The solid lines are the cobald delta file sizes when uncompressed, and compressed with gzip and 7-zip. (i) Web snap-shots week2 was encoded with week1 in the index. (ii) Genome release-56 was encoded with release-55 in the index. This data is listed in Table 3.3
(i) Encoding Web snap-shots, week 2

<table>
<thead>
<tr>
<th>Input</th>
<th>gzip</th>
<th>7-zip</th>
</tr>
</thead>
<tbody>
<tr>
<td>no cobald</td>
<td>100.0</td>
<td>11.93</td>
</tr>
</tbody>
</table>

\[
\begin{array}{cccc}
\hline
n & Delta & \% & \times \\
\hline
65 536 & 98.07 & 11.63 & 1.0 \\
16 384 & 71.68 & 7.65 & 1.6 \\
4 096 & 9.62 & 1.20 & 9.9 \\
1 024 & 3.41 & 0.46 & 26.0 \\
256 & 1.87 & 0.31 & 38.6 \\
64 & 1.33 & 0.32 & 37.4 \\
\hline
\end{array}
\]

(ii) Encoding Genome, release-56

<table>
<thead>
<tr>
<th>Input</th>
<th>gzip</th>
<th>7-zip</th>
</tr>
</thead>
<tbody>
<tr>
<td>no cobald</td>
<td>100.0</td>
<td>23.25</td>
</tr>
</tbody>
</table>

\[
\begin{array}{cccc}
\hline
n & Delta & \% & \times \\
\hline
65 536 & 46.09 & 9.32 & 2.5 \\
16 384 & 45.21 & 9.26 & 2.5 \\
4 096 & 36.24 & 8.15 & 2.9 \\
1 024 & 18.07 & 4.58 & 5.1 \\
256 & 7.00 & 2.01 & 11.6 \\
64 & 5.02 & 1.60 & 14.5 \\
\hline
\end{array}
\]

Table 3.3: Compressed file sizes using a range of n-gram lengths. The percentages are the file size as a proportion of the uncompressed input file size. The multiples (in columns labelled ×) are the improvement by cobald compared to compression of the files individually without cobald. This is calculated as the compressed input file size divided by the compressed delta size using gzip and 7-zip. (i) Web snap-shots week2 was encoded with week1 in the index. (ii) Genome release-56 was encoded with release-55 in the index. This data is shown in Figure 3.6.
While the general trend is for monotonic improvement in compression as the $n$-gram length decreases, this is not the case for $n = 64$, where the Web snap-shots compressed deltas and slightly larger than than for $n = 256$, and only marginally smaller for the Genome compressed deltas. As the uncompressed deltas are clearly smaller, the reduced compression can be attributed to reduce effectiveness of the second-stage compression by gzip and 7-zip. One partial explanation for this observation is the diminishing effectiveness of encoding shorter matching strings. In addition to the unmatched bytes in each ADD block, the delta file includes data describing each COPY block, as well as the length of each ADD instruction. These references are encoded in 20 bytes, so the gain from encoding short strings is less significant.

Table 3.4 shows the index file size, memory use, and time taken to insert and encode by cobald. The index file size is inversely related to the $n$-gram length as halving the $n$-gram length result in twice as many $n$-grams. This relationship does not hold exactly when the index is created in memory as references to duplicate $n$-grams are discarded. The other large data structures are the index for $n$-grams from earlier in the input file, which has size proportional to the input file length, and the list of matches which is used to construct the delta file, which is limited to a size also proportional to the input file length. Consequently, the memory used by cobald is proportional to the combined size of the indexed files and the input file being encoded. The memory use is shown as a percentage of this total file size. We can see that the memory use increases at slightly less than the inverse of the $n$-gram length.

The insertion operation consists of two main elements: reading the file and calculating the fingerprints with the rolling hash function, which relates to the file size and is the same for all $n$-gram lengths; and inserting the $n$-gram references in the index, which increases with the number of $n$-grams. These two components
can be clearly identified. When inserting Web snap-shots, approximately 230 seconds is required to read and fingerprint the files, and approximately 50 seconds for Genome release-55. The remaining time is inversely related to the \( n \)-gram length.

The encoding time is more complicated. The encode operation has two modes when processing the input file, which run at different speeds. When the current position has no matching string in the collection, the \( n \)-gram at every position of the input file is queried in the index, which is expensive. When the current position is part of a match, the encoder is extending the match forward by direct comparison of the strings in the input and collection files, which is a faster operation. Consequently, files that \texttt{cobald} compresses well are encoded faster. Counteracting this is the high cost of starting a new matching string, which involves reading from the collection file on external storage, so a high number of short matches is expensive. The Web snap-shots data set achieved higher compression than the Genome data set, and this is reflected in the reduced encoding time for Web snap-shots as the \( n \)-gram length decreases. However, when \( n = 64 \) the Web snap-shot encoding time increases, largely because of the time penalty from external reads for small matches.

The key conclusions are that encoding with \( n = 64 \) is not worthwhile. It does not achieve better compression than \( n = 256 \), and the memory usage and index file size are a significant fraction of the collection size. At the other extreme, \( n = 65536 \) and 16384 do not achieve significant compression. The most practical choices of \( n \)-gram length are \( n = 1024 \) and 256, which achieve good compression while using resources less than 10% of the collection size.
## Table 3.4: Index file size, memory and time use by cobald during the compression test.

(i) Web snap-shots week2 data set was encoded with week1 in the index. (ii) Genome release-56 data set was encoded with release-55 in the index. The index file size is shown as a percentage of the input files which were inserted in the index, (i) Web snap-shots week1, and (ii) Genome release-55. The memory column shows the maximum memory used when encoding an input file, and expressed as a percentage of the sum of the file sizes of the indexed files and the input file being encoded. The insert time is the total time to insert the files into the index. The encode time is the total time to encode all of the files in the input data sets.

### (i) Encoding Web snap shots, week2

<table>
<thead>
<tr>
<th>n</th>
<th>Index file size</th>
<th>Memory</th>
<th>Time (seconds)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Mbytes</td>
<td>%</td>
<td>Mbytes</td>
</tr>
<tr>
<td>65536</td>
<td>1.7</td>
<td>0.03</td>
<td>6.3</td>
</tr>
<tr>
<td>16384</td>
<td>6.8</td>
<td>0.10</td>
<td>21.2</td>
</tr>
<tr>
<td>4096</td>
<td>27.2</td>
<td>0.42</td>
<td>77.3</td>
</tr>
<tr>
<td>1024</td>
<td>108.6</td>
<td>1.66</td>
<td>228.6</td>
</tr>
<tr>
<td>256</td>
<td>434.5</td>
<td>6.64</td>
<td>600.0</td>
</tr>
<tr>
<td>64</td>
<td>1,738.1</td>
<td>26.56</td>
<td>1,544.0</td>
</tr>
</tbody>
</table>

### (ii) Encoding Genome, release-56

<table>
<thead>
<tr>
<th>n</th>
<th>Index file size</th>
<th>Memory</th>
<th>Time (seconds)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Mbytes</td>
<td>%</td>
<td>Mbytes</td>
</tr>
<tr>
<td>65536</td>
<td>0.3</td>
<td>0.03</td>
<td>2.5</td>
</tr>
<tr>
<td>16384</td>
<td>1.2</td>
<td>0.10</td>
<td>4.5</td>
</tr>
<tr>
<td>4096</td>
<td>4.7</td>
<td>0.42</td>
<td>14.0</td>
</tr>
<tr>
<td>1024</td>
<td>18.9</td>
<td>1.66</td>
<td>51.3</td>
</tr>
<tr>
<td>256</td>
<td>75.5</td>
<td>6.64</td>
<td>179.4</td>
</tr>
<tr>
<td>64</td>
<td>302.0</td>
<td>26.56</td>
<td>666.0</td>
</tr>
</tbody>
</table>
3.8 Content-Defined Chunk Selection

Fixed-offset chunk selection, where the chunk references inserted in the index are selected at every $n$-th position in the file, leads to some inefficiency for subsequent searches. When there is a long matching substring in the collection, up to $n - 1$ index queries may be performed before a matching $n$-gram is found. We refer to these index queries for chunks that exist in the collection but which were not indexed as false negative queries.

Several alternative chunk selection algorithms, reduce the rate of false negative queries, are described in Section 2.6. These algorithms define some property for each chunk determined solely by the content of the chunk or its local neighbourhood. If only those chunks having the property are inserted in the index, there is no need to search for chunks not having the property, thus eliminating false negative queries. In this section we investigate whether delta encoding is affected by eliminating false negative index queries using content-defined selection. While a reduction in the number of index queries is expected to increase the speed of the encoding step, content-defined selection involves some additional costs. One key effect is that the compression achieved will be reduced for a given index size. If the content-defined parameters are selected so that expected inter-chunk distance is equal to $n$, so that the index size is the same as a fixed-offset index, there will be some adjacent chunks with large gaps between them. Data in these long chunks is not represented in the index, and a string match in these chunks will not be found. Another cost is the additional computation required to calculate the selection property. In some content-defined selection algorithms, this is significant and the algorithm is not useful if it exceeds the time saved by eliminating the false-negative queries.
Selection algorithm  The various content-defined selection algorithms are summarised in Table 2.1. The choice of algorithm depends largely on the relative cost of very short chunks and very long chunks. For our hash index in memory, the cost of short chunks is a larger index, while the cost of very long chunks is that some matching strings will not be found, potentially leading to reduced compression. A larger index will not have a significant impact on query speed, although it will reduce the limit on the collection size before measures must be taken to cap the memory use. Here, we investigate a method to achieve good compression by reducing the occurrence of long chunks even at the cost of a larger index.

A related consideration is that long periodic strings (strings containing repeated short substrings, such as $\texttt{abababab\ldots}$) should be handled gracefully. Again, the cost of long strings that do not contain selection points is a potential loss of compression, while the cost of long strings having a selection point at every position must be limited. In cobald, all of the chunks will be appended to the index file, but only a single reference to each unique $n$-gram is loaded into the index in memory. By careful choice of the selection function so that known periodic strings in the input (such as a string of zeroes) do not contain selection points, the occurrence of adjacent $n$-gram references can be minimised.

Two content-defined selection algorithms meet these criteria: (i) the method of Manber (1994), where the selection point occurs when the $n$-gram hash $h = c \pmod{d}$, with a maximum distance between selection points; and (ii) winnowing of Schleimer et al. (2003). In cobald, we employed the method of Manber. The chunking algorithm, which is defined in in the \texttt{GETCHUNK()} function of Algorithm 8, has three parameters: the $n$-gram length $n$, the number of bits in the chunk selection mask $s$, and the maximum chunk length $m$. Each $n$-gram of the input is hashed, and if the least significant $s$ bits of the hash equal 1, the
3.8. Content-Defined Chunk Selection

![Diagram showing chunks and indexed n-grams]

Figure 3.7: Content-defined chunk and n-gram positions. Each chunk begins at the byte following an indexed n-gram, and ends with the last byte of n-gram which terminates the chunk. In a short chunk, the n-gram may begin before the first byte of the chunk.

An n-gram terminates the chunk. Also, if m bytes of the input have been read since the end of the previous chunk, the chunk is terminated. The relative positions of the terminating n-grams and the chunk boundaries is illustrated in Figure 3.7.

Assuming a uniform hash distribution, each n-gram has a probability of $1/2^s$ of satisfying the termination condition, $h \pmod{d}$. As described in Section 2.6, the theoretical distribution of chunk lengths without a maximum length is geometric, with a mean chunk length of $2^s$. The actual expected chunk length will be slightly less when a maximum chunk length is imposed.

The algorithm for encoding with content-defined selection is described in Algorithm 8. Like fixed-offset encoding, every n-gram of the input is hashed, however only the last n-gram of each chunk is queried in the index as we know that n-grams which did not satisfy the chunking condition were not inserted. However this is not the case when the chunk is terminated when it reached the maximum length. Then the final n-gram did not satisfy the chunking condition. If the maximal chunk is part of a matching string in the collection, we do not know which n-gram was inserted in the index, so when during a maximal chunk, the encoding algorithm reverts back to the fixed-offset behaviour, and every n-gram is queried in the index.
Algorithm 8 Encoding with content-defined sampled fingerprint index

Data: \( R \), the set of collection files;
\( Q \), the input file of length \( N \) (bytes);
Index file of sampled \( n \)-gram fingerprints and references from files in \( R \).

Parameters: \( n \), the length of \( n \)-grams (bytes);
\( b \), the maximum backward extension distance (bytes);
\( s \), the number of bits in the chunk selection mask;
\( c \), the maximum chunk length (bytes).

Result: A delta file encoding \( Q \) in terms of \( R \).

1: Construct \( I_R \), the index of fingerprints from \( R \) in memory.
   Each \( n \)-gram reference in the index file is inserted into \( I_R \) if an entry with the same fingerprint and collision ID has not previously been inserted. The lower \( s \) bits of the fingerprints are removed (using right-shift) before insertion into the index as these bits are used to define the chunk boundaries.

2: Create \( I_Q \), the empty index of fingerprints from \( Q \) in memory.

3: Create \( L \), the empty list of string match references found during encoding.

4: \( mode \leftarrow \text{Searching} \).
   The algorithm uses two modes as it processes \( Q \): in \text{Searching} mode the index is queried for matching \( n \)-grams; and in \text{Extending} mode a string match is extended forward.

5: \( i \leftarrow n - 1 \).
   Position \( i \) in \( Q \) is the last byte of the previously encoded chunk. It is initialised to the last byte of the first \( n \)-gram.

6: while \( i < N \) do
7: \( j, l_c, F \leftarrow \text{GETCHUNK}(i) \).
   Read the next chunk from \( Q \), and set \( j \), the position in \( Q \) of the first byte of the last \( n \)-gram in the chunk; the chunk length, \( l_c \); and the array, \( F \), containing the fingerprints of all \( n \)-grams in the chunk.
8: if \( mode = \text{Extending} \) or \( l_c = c \) then
9: \( k \leftarrow i - n + 1 \).
   Set \( k \), the position of the next operation on this chunk, to the start of the first \( n \)-gram in the chunk.
10: else
11: \( k \leftarrow j \).
   For short chunks, only the last \( n \)-gram is queried in the index.

continued ...
Algorithm 8 . . . continued.

12: while $k \leq j$ do
13:     if mode = Extending then
14:         $i_e \leftarrow$ EXTENDFORWARD($k + n - 1, M, a + n, j + n - 1$).
15:         if the end of the match is found then
16:             $l = i_e - i_b$, the length of the match.
17:             mode $\leftarrow$ Searching.
18:             APPENDMATCH($i_b, M, a_b, l$).
19:             Resolve overlapping matches.
20:     if $l_c = c$ then
21:         $k \leftarrow i_e - n + 1$.
22:     else
23:         $k \leftarrow j$
24:     end
25:     if mode = Searching then
26:         $M, a \leftarrow$ INDEXQUERY($F[k - i + n + 1]$).
27:         Query the index for the $n$-gram. When a match is found, $M$ is
28:         a reference to the file containing the matching $n$-gram, and $a$
29:         is the position of the $n$-gram in $M$.
30:         if a match is found then
31:             $i_b \leftarrow$ EXTENDBACKWARD($k, M, a$), and
32:             $a_b \leftarrow a - (k - i_b)$.
33:             mode $\leftarrow$ Extending.
34:     end
35:     $k \leftarrow k + 1$.
36: end
37: INDEXINSERT($Q, F[0], j$)
38: $i \leftarrow j + n$
39: GENERATEDELTA(

continued . . .
Algorithm 8 . . . continued.

35: function ExtendForward($i, M, j, i_{\text{max}}$)

Find the end of the matching string, while inserting $n$-grams into $I_Q$.

36: Starting at positions $x = i$ in $Q$ and $y = j$ in collection file $M$, increment $x$ and $y$ until either $Q[x] \neq M[y]$, the end of $Q$ or $M$ is reached, or the end of the chunk, $i_{\text{max}}$, is reached.

37: At each position, if $x + 1 \mod n \equiv 0$ then

38: $f_p \leftarrow \text{Hash}(Q[x - n + 1, x])$, and

39: $\text{InsertIndex}(f_p, x - n + 1)$.

        Insert every $n$-th $n$-gram fingerprint into $I_Q$. Note that $x$ is the position of the last symbol in the $n$-gram.

40: Return $x$, the position in $Q$ following the end of the matching string, or FALSE if the end of the match is not found before reaching the end of the chunk.

41: function GetChunk($i$)

Find the next chunk in $Q$ after position $i$. The chunk ends when the lower $s$ bits of the $n$-gram fingerprint are set, or the chunk reaches the maximum length, $c$.

42: $j \leftarrow 0$

43: repeat

44: $f_p \leftarrow \text{Hash}(Q[i + j - n + 1, i + j])$

        Find the fingerprint of the $n$-gram ending at position $i + j$ in $Q$.

45: $F[j] \leftarrow f_p$ right-shifted by $s$ bits.

        The lower $s$ bits are removed from the fingerprint used in the index.

46: $j \leftarrow j + 1$.

47: until $j = c$ or $j = N - i$ or the lower $s$ bits of $f_p$ are set.

48: Return $i + j, j, F$.

49: function AppendMatch($i, M, j, l$)

50: function QueryIndex($f_p, i$)

These functions are defined in Algorithm 7.
3.9 Results: Content-Defined Selection

The effectiveness of content-defined selection was evaluated by encoding the same files used in Section 3.7 using a range of maximum chunk lengths. The results are presented in Table 3.5 and Figure 3.8. With the data from both collections, we see that the encoding effectiveness decreases with longer maximum lengths, which is expected as there are larger gaps between indexed \(n\)-grams and a greater chance that a matching string in collection files does not contain an indexed \(n\)-gram. The reduction in compression effectiveness is greater for the Genome release-56 files than for the Web snap-shots data set. We attribute this to the characteristics of the data: Genome release-55 appears to contain a high number of matches close to 1024 bytes in length that are missed by the content-defined selection process. This is supported by the results in Figure 3.5: there is little improvement in compression between \(n = 4096\) and 65536, indicating that there are few string matches of length 4096 that are not part of matches longer than 65536.

The broad effect of maximum chunk length on the chunk length distribution can be seen in the average chunk length, listed in Table 3.6, which increases gradually with the maximum length. Also, the index file size decreases slightly as there are fewer chunks, and the memory used when encoding decreases slightly as the index is smaller. The chunk length distribution is shown in detail in Figure 3.9. Without a maximum chunk length, the theoretical distribution is geometric, as described in Section 2.6. It can be seen that the imposition of a maximum chunk length does not significantly change the distribution of lengths that are less than the maximum length.

A key result is that the memory used when encoding is reduced by over 20% for Web snap-shots and 50% for the Genome data set compared to encoding with fixed-offset \(n\)-gram selection. This is because most false-negative \(n\)-grams have
Figure 3.8: Percent matched by the encoding operation using a range of maximum chunk lengths. This data is presented in Table 3.5. Web snap-shots week2 was encoded with week1 in the index, and Genome release-56 was encoded with release-55 in the index. The encoding was performed with \( n \)-gram length \( n = 1024 \), and the chunk selection mask having 10 bits. Each data point represents the total length of data in COPY blocks in the delta files, as a proportion of the total input files size. The dashed line is the result achieved when encoding with fixed-offset selection from Section 3.7.

been eliminated from the index. The encoding algorithm only inserts \( n \)-grams that meet the chunk selection condition, so, when indexing a duplicate string, the same \( n \)-grams will be inserted from both strings, and the second one will be discarded. However, with fixed-offset selection different \( n \)-grams are likely to be indexed from a duplicate string, and both sets of \( n \)-grams will be inserted into the index in memory.

The encoding times are shown in Table 3.6 and Figure 3.10. The encode time decreases as the maximum chunk length increases. The main reason is that, with
3.9. Results: Content-Defined Selection

<table>
<thead>
<tr>
<th>m</th>
<th>Web snap-shots</th>
<th>Genome</th>
</tr>
</thead>
<tbody>
<tr>
<td>2048</td>
<td>95.92</td>
<td>78.62</td>
</tr>
<tr>
<td>3072</td>
<td>95.42</td>
<td>77.56</td>
</tr>
<tr>
<td>4096</td>
<td>95.19</td>
<td>77.08</td>
</tr>
<tr>
<td>5120</td>
<td>95.05</td>
<td>76.82</td>
</tr>
<tr>
<td>6144</td>
<td>95.01</td>
<td>76.66</td>
</tr>
<tr>
<td>7168</td>
<td>94.98</td>
<td>76.53</td>
</tr>
<tr>
<td>8192</td>
<td>94.96</td>
<td>76.42</td>
</tr>
<tr>
<td>fixed-offset, $n=1024$</td>
<td>96.84</td>
<td>82.22</td>
</tr>
</tbody>
</table>

Table 3.5: Percentage of input files that were matched by the encoding operation using a range of maximum chunk lengths, m. Web snap-shots week2 was encoded with week1 in the index, and Genome release-56 was encoded with release-55 in the index. The encoding was performed with n-gram length $n=1024$, and the chunk selection mask having 10 bits so that each n-gram has a 1/1024 probability of being selected. The second and third columns show the total length of data represented in COPY blocks in the delta files, as a proportion of the total input files size. The last line is the result achieved when encoding with fixed-offset selection from Section 3.7.

When the maximum chunk length is short, there is a higher proportion of chunks having the maximum length. These maximum length chunks are treated as a special case by the encoding algorithm: the n-grams at each position in the chunk are queried in the index, in a similar manner to fixed-offset encoding. This takes substantially longer than a shorter chunk where only the last n-gram is queried. As the maximum chunk length increases, there are fewer chunks having the maximum length and therefore fewer index queries. A lesser contribution to the reduced encoding time is that, as the maximum chunk length increases, the average chunk length increases and there are fewer n-grams to insert in the input file index.

The insertion times are much slower for content-defined selection compared to fixed-offset selection, being nearly twice as long for the Genome data, and four times longer for Web snap-shots. This is because, when inserting data that matches previously inserted data, the matching fingerprint will be found in the index and the inserter will perform a collision resolution step, which involves an
external read from the collection file. With fixed-offset selection, it is unlikely that the same \( n \)-gram references will be selected for both occurrences of the matching data. Consequently, the reduced memory for content-defined selection comes at the cost of longer insertion time.

The key conclusions are that content-defined selection provides a significant benefit by reducing the memory needed to encode, and is also faster for longer maximum chunk lengths. The best choice depends on the data: for the Web snapshots collection, a maximum of 5120 (5 times the expected chunk length) achieves fast encoding; while for the Genome collection, there is little speed improvement with a maximum length beyond 3 times the expected chunk length. Even a maximum chunk length of 2048 (2 times the expected chunk length) achieved a large reduction in memory use, compared to fixed-offset encoding, with only a small decrease in compression.

### 3.10 Comparison With Other Systems

We were unable to run performance comparisons with other systems on a CBC task involving multiple collection files. Of the collection compression systems described in Section 2.9, PRESIDIO (You et al., 2011) is the most developed, and has the capability to perform CBC with multiple files. However, we were unable to obtain the software from the authors.

Instead, we compared the performance of \texttt{cobald} against some of the single reference file delta encoders and compressors described in Section 2.5.2. Specifically, we made comparisons with \texttt{xdelta3} and \texttt{open-vcdiff}, which are available as open source software, and \texttt{hsadelta}, which is provided by IBM under an ‘early release’ licence. Our tests involve encoding and compression of a single input file with respect to a single reference file, which is done in \texttt{cobald} by inserting the
### 3.10. Comparison With Other Systems

(i) Encoding Web snap-shots week 2

<table>
<thead>
<tr>
<th>Maximum chunk length</th>
<th>Average chunk length</th>
<th>Index file size</th>
<th>Memory</th>
<th>Time (seconds)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bytes</td>
<td>Bytes</td>
<td>Mbytes</td>
<td>%</td>
<td>Mbytes</td>
</tr>
<tr>
<td>8 192</td>
<td>1 059.3</td>
<td>105.0</td>
<td>1.60</td>
<td>159.2</td>
</tr>
<tr>
<td>7 168</td>
<td>1 058.7</td>
<td>105.1</td>
<td>1.61</td>
<td>159.2</td>
</tr>
<tr>
<td>6 144</td>
<td>1 056.7</td>
<td>105.3</td>
<td>1.61</td>
<td>159.5</td>
</tr>
<tr>
<td>5 120</td>
<td>1 051.2</td>
<td>105.8</td>
<td>1.62</td>
<td>160.1</td>
</tr>
<tr>
<td>4 096</td>
<td>1 035.4</td>
<td>107.4</td>
<td>1.64</td>
<td>161.5</td>
</tr>
<tr>
<td>3 072</td>
<td>998.8</td>
<td>111.4</td>
<td>1.70</td>
<td>166.3</td>
</tr>
<tr>
<td>2 048</td>
<td>902.8</td>
<td>123.2</td>
<td>1.88</td>
<td>181.0</td>
</tr>
<tr>
<td>fixed</td>
<td>1 024</td>
<td>108.6</td>
<td>1.66</td>
<td>228.6</td>
</tr>
</tbody>
</table>

(ii) Encoding Genome release-56

<table>
<thead>
<tr>
<th>Maximum chunk length</th>
<th>Average chunk length</th>
<th>Index file size</th>
<th>Memory</th>
<th>Time (seconds)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bytes</td>
<td>Bytes</td>
<td>Mbytes</td>
<td>%</td>
<td>Mbytes</td>
</tr>
<tr>
<td>8 192</td>
<td>1 060.8</td>
<td>18.2</td>
<td>1.60</td>
<td>23.0</td>
</tr>
<tr>
<td>7 168</td>
<td>1 059.4</td>
<td>18.3</td>
<td>1.61</td>
<td>23.0</td>
</tr>
<tr>
<td>6 144</td>
<td>1 056.6</td>
<td>18.3</td>
<td>1.61</td>
<td>23.0</td>
</tr>
<tr>
<td>5 120</td>
<td>1 050.7</td>
<td>18.4</td>
<td>1.62</td>
<td>23.0</td>
</tr>
<tr>
<td>4 096</td>
<td>1 036.1</td>
<td>18.7</td>
<td>1.64</td>
<td>23.3</td>
</tr>
<tr>
<td>3 072</td>
<td>999.3</td>
<td>19.3</td>
<td>1.70</td>
<td>24.1</td>
</tr>
<tr>
<td>2 048</td>
<td>905.1</td>
<td>21.4</td>
<td>1.88</td>
<td>26.1</td>
</tr>
<tr>
<td>fixed</td>
<td>1 024</td>
<td>18.9</td>
<td>1.66</td>
<td>51.3</td>
</tr>
</tbody>
</table>

Table 3.6: Index file sizes, memory use, inserting and encoding times for encoding with content-defined n-gram selection. (i) Web snap-shots week 2 data set was encoded with week1 files in the index. (ii) Genome release-56 files were encoded with release-55 in the index. The index file size is shown as a percentage of the total file sizes inserted in the index. The memory use is shown as a percentage of the total index file sizes plus the input file size for which the memory result was obtained. The insert time is the total time to insert the indexed files. The encode time is the total time to encode all files from the input data set.
(i) Web snap-shots week1 inserted in the index

![Graph showing the distribution of chunk lengths for Web snap-shots week1.]

(ii) Genome release-55 inserted in the index

![Graph showing the distribution of chunk lengths for Genome release-55.]

Figure 3.9: Distribution of chunk lengths from a range of maximum chunk lengths. Each point represents the mean probability of chunk lengths in bins of size 32 bytes. The chunks lengths were from the inserted files from (i) Web snap shots week1, and (ii) Genome release-55.
reference file into the index, then encoding the input file and compressing the delta.

The results are presented in Tables 3.7 to 3.10. In addition to some data from the Web snap-shots and Genome collections, these trials use some files from commonly available source and binary software distributions which were used by Agarwal et al. (2006) when evaluating the performance of hsadelta. All of these files are aggregated into the combine set. The files used are listed in the Appendix.

Hsadelta includes a compression stage using bzip2, so the delta files from cobald and open-vcdiff were compressed with bzip2 --best to ensure a fair comparison. Xdelta3 uses zlib to compress its delta, so it was run with the
<table>
<thead>
<tr>
<th>File pair</th>
<th>Input files</th>
<th>cobald $F,n_{256}$</th>
<th>cobald $CD,n_{256}$</th>
<th>cobald $F,n_{1024}$</th>
<th>cobald $CD,n_{1024}$</th>
<th>hsadelta</th>
<th>xdelta3</th>
<th>vdiff</th>
<th>bzip2</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>(%)</td>
<td>35.94</td>
<td>35.91</td>
<td>36.80</td>
<td>36.93</td>
<td>36.88</td>
<td>36.55</td>
<td>35.68</td>
<td>38.69</td>
</tr>
<tr>
<td>gentoo1</td>
<td>Mbytes</td>
<td>14.56</td>
<td>14.86</td>
<td>16.21</td>
<td>16.63</td>
<td>12.27</td>
<td>18.87</td>
<td>12.08</td>
<td>40.47</td>
</tr>
<tr>
<td></td>
<td>(%)</td>
<td>10.37</td>
<td>10.58</td>
<td>11.54</td>
<td>11.84</td>
<td>8.73</td>
<td>13.43</td>
<td>8.60</td>
<td>28.82</td>
</tr>
<tr>
<td>gcc2</td>
<td>Mbytes</td>
<td>1.93</td>
<td>2.06</td>
<td>2.92</td>
<td>3.27</td>
<td>1.36</td>
<td>1.49</td>
<td>1.71</td>
<td>25.92</td>
</tr>
<tr>
<td></td>
<td>(%)</td>
<td>1.07</td>
<td>1.14</td>
<td>1.61</td>
<td>1.80</td>
<td>0.75</td>
<td>0.82</td>
<td>0.94</td>
<td>14.31</td>
</tr>
<tr>
<td>gcc1</td>
<td>Mbytes</td>
<td>9.31</td>
<td>9.68</td>
<td>12.99</td>
<td>13.53</td>
<td>8.05</td>
<td>8.96</td>
<td>8.52</td>
<td>26.00</td>
</tr>
<tr>
<td></td>
<td>(%)</td>
<td>5.10</td>
<td>5.30</td>
<td>7.11</td>
<td>7.41</td>
<td>4.41</td>
<td>4.91</td>
<td>4.66</td>
<td>14.24</td>
</tr>
<tr>
<td>linux1</td>
<td>Mbytes</td>
<td>14.63</td>
<td>15.10</td>
<td>19.32</td>
<td>20.01</td>
<td>12.33</td>
<td>26.88</td>
<td>12.65</td>
<td>33.47</td>
</tr>
<tr>
<td></td>
<td>(%)</td>
<td>7.75</td>
<td>8.00</td>
<td>10.24</td>
<td>10.60</td>
<td>6.53</td>
<td>14.25</td>
<td>6.70</td>
<td>17.73</td>
</tr>
<tr>
<td>linux2</td>
<td>Mbytes</td>
<td>2.23</td>
<td>2.36</td>
<td>3.64</td>
<td>4.07</td>
<td>1.79</td>
<td>6.15</td>
<td>1.79</td>
<td>33.98</td>
</tr>
<tr>
<td></td>
<td>(%)</td>
<td>1.16</td>
<td>1.23</td>
<td>1.90</td>
<td>2.12</td>
<td>0.93</td>
<td>3.21</td>
<td>0.94</td>
<td>17.74</td>
</tr>
<tr>
<td>linux3</td>
<td>Mbytes</td>
<td>2.15</td>
<td>2.34</td>
<td>3.73</td>
<td>4.17</td>
<td>1.75</td>
<td>31.16</td>
<td>1.77</td>
<td>34.58</td>
</tr>
<tr>
<td></td>
<td>(%)</td>
<td>1.10</td>
<td>1.20</td>
<td>1.91</td>
<td>2.14</td>
<td>0.90</td>
<td>15.97</td>
<td>0.91</td>
<td>17.72</td>
</tr>
<tr>
<td>gentoo2</td>
<td>Mbytes</td>
<td>25.67</td>
<td>26.47</td>
<td>29.96</td>
<td>31.20</td>
<td>23.32</td>
<td>50.79</td>
<td>21.24</td>
<td>89.39</td>
</tr>
<tr>
<td></td>
<td>(%)</td>
<td>9.16</td>
<td>9.44</td>
<td>10.69</td>
<td>11.13</td>
<td>8.32</td>
<td>18.12</td>
<td>7.58</td>
<td>31.89</td>
</tr>
<tr>
<td>install</td>
<td>Mbytes</td>
<td>574.23</td>
<td>574.26</td>
<td>574.49</td>
<td>574.53</td>
<td>627.21</td>
<td>649.90</td>
<td>579.70</td>
<td>655.63</td>
</tr>
<tr>
<td></td>
<td>(%)</td>
<td>85.22</td>
<td>85.23</td>
<td>85.26</td>
<td>85.27</td>
<td>93.09</td>
<td>96.46</td>
<td>86.04</td>
<td>97.31</td>
</tr>
<tr>
<td>genome</td>
<td>Mbytes</td>
<td>17.59</td>
<td>20.56</td>
<td>40.94</td>
<td>48.27</td>
<td>78.76</td>
<td>330.40</td>
<td>na</td>
<td>269.89</td>
</tr>
<tr>
<td></td>
<td>(%)</td>
<td>1.47</td>
<td>1.72</td>
<td>3.42</td>
<td>4.03</td>
<td>6.58</td>
<td>27.61</td>
<td>22.56</td>
<td>57.22</td>
</tr>
<tr>
<td>combine</td>
<td>Mbytes</td>
<td>624.94</td>
<td>626.23</td>
<td>637.39</td>
<td>640.18</td>
<td>725.33</td>
<td>797.13</td>
<td>na</td>
<td>858.27</td>
</tr>
<tr>
<td></td>
<td>(%)</td>
<td>41.67</td>
<td>41.75</td>
<td>42.50</td>
<td>42.68</td>
<td>48.36</td>
<td>53.15</td>
<td>57.22</td>
<td>57.22</td>
</tr>
<tr>
<td>web</td>
<td>Mbytes</td>
<td>5.57</td>
<td>5.78</td>
<td>6.83</td>
<td>7.59</td>
<td>7.54</td>
<td>84.98</td>
<td>na</td>
<td>131.45</td>
</tr>
<tr>
<td></td>
<td>(%)</td>
<td>0.29</td>
<td>0.30</td>
<td>0.36</td>
<td>0.40</td>
<td>0.40</td>
<td>4.46</td>
<td>6.89</td>
<td>6.89</td>
</tr>
</tbody>
</table>

Table 3.7: Comparison of compression achieved by cobald, hsadelta, xdelta3 and open-vcdiff. For each trial, the top number is the size of the compressed delta file in megabytes, and the bottom number is the compressed delta file size as a percentage of the query file size.
option to achieve best compression. The \texttt{xdelta3} delta is encoded using the VCDIFF format using an improved variable bit-length encoding which is more compact than the original byte code. \texttt{Xdelta3} version 3.0.0 was used in the tests with the command line \texttt{xdelta3 encode -9 -S djw -s REFERENCEFILE INPUTFILE OUTPUTFILE}. The delta files from \texttt{cobald} were converted to VCDIFF format prior to compression with \texttt{bzip2}.

The first observation from the compression results in Table 3.7 is that, for files less than 300 Mbytes, \texttt{vcdiff} and \texttt{hsadelta} are most effective. \texttt{Cobald} is more effective than \texttt{xdelta3} on some of the files. All, except \texttt{xdelta3} on some files, show substantially greater compression than \texttt{bzip2} compressing the input file, demonstrating that there was substantial similarity in the file pairs to be exploited by delta encoding.

An important observation from the compression results is that the relative compression achieved by \texttt{cobald} improves for larger input and reference files. Or perhaps it may be more accurate to say that the performance of the other tools degrades as their mechanisms for limiting memory take effect. \texttt{xdelta3} uses 16-byte \textit{n}-grams and discards earlier \textit{n}-grams from its index to limit its size. As the file sizes increase, the indexed \textit{n}-grams will cover less of the input and reference file, and we expect less matches to be found, leading to degraded compression. We observe this effect as \texttt{xdelta3} achieves significantly less compression that the other tools on the largest files. \texttt{hsadelta} uses a different mechanism to achieve constant memory use. It increases \textit{n} as the input and reference file sizes increase so that approximately the same number of \textit{n}-grams are inserted in the index. This allows the index to cover all of the input and reference files, but at a lower sampling frequency. Again, we observe that the performance of \texttt{hsadelta} declines as the file sizes increase. It was the best compressor for many of the small files, but was surpassed by \texttt{vcdiff} and \texttt{cobald} for the larger ones. Finally,
<table>
<thead>
<tr>
<th>File pair</th>
<th>Input files</th>
<th></th>
<th>cobald F, n256</th>
<th>cobald F, n1024</th>
<th>cobald hsadelta</th>
<th>cobald CD, n256</th>
<th>cobald CD, n1024</th>
<th>xdelta3</th>
<th>xdelta3</th>
<th>vcdiff</th>
<th>vcdiff</th>
<th>bzip2</th>
<th>bzip2</th>
</tr>
</thead>
<tbody>
<tr>
<td>samba (Mbytes)</td>
<td>34 22</td>
<td>9.9</td>
<td>10.7</td>
<td>6.7</td>
<td>6.7</td>
<td>90.1</td>
<td>201.7</td>
<td>93.3</td>
<td>6.7</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(%)</td>
<td>17.8</td>
<td>19.1</td>
<td>12.1</td>
<td>12.1</td>
<td>161.9</td>
<td>362.3</td>
<td>167.6</td>
<td>12.1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>genool (Mbytes)</td>
<td>140 138</td>
<td>36.9</td>
<td>35.2</td>
<td>12.1</td>
<td>10.8</td>
<td>165.7</td>
<td>239.2</td>
<td>608.0</td>
<td>6.7</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(%)</td>
<td>13.2</td>
<td>12.6</td>
<td>4.3</td>
<td>3.9</td>
<td>59.4</td>
<td>85.8</td>
<td>218.0</td>
<td>2.4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>gcc2 (Mbytes)</td>
<td>181 183</td>
<td>49.9</td>
<td>53.8</td>
<td>14.7</td>
<td>16.0</td>
<td>163.6</td>
<td>235.5</td>
<td>719.0</td>
<td>7.5</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(%)</td>
<td>13.7</td>
<td>14.8</td>
<td>4.0</td>
<td>4.4</td>
<td>45.0</td>
<td>64.7</td>
<td>197.6</td>
<td>2.1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>gcc1 (Mbytes)</td>
<td>183 146</td>
<td>48.9</td>
<td>51.8</td>
<td>13.9</td>
<td>15.0</td>
<td>162.8</td>
<td>235.9</td>
<td>626.0</td>
<td>7.5</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(%)</td>
<td>14.9</td>
<td>15.8</td>
<td>4.2</td>
<td>4.6</td>
<td>49.6</td>
<td>71.8</td>
<td>190.7</td>
<td>2.3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>linux1 (Mbytes)</td>
<td>189 164</td>
<td>53.8</td>
<td>58.2</td>
<td>14.7</td>
<td>16.2</td>
<td>165.0</td>
<td>237.7</td>
<td>672.0</td>
<td>7.5</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(%)</td>
<td>15.2</td>
<td>16.5</td>
<td>4.2</td>
<td>4.6</td>
<td>46.8</td>
<td>67.4</td>
<td>190.5</td>
<td>2.1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>linux2 (Mbytes)</td>
<td>192 189</td>
<td>55.4</td>
<td>61.4</td>
<td>15.5</td>
<td>17.3</td>
<td>163.8</td>
<td>237.5</td>
<td>733.0</td>
<td>7.5</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(%)</td>
<td>14.6</td>
<td>16.2</td>
<td>4.1</td>
<td>4.5</td>
<td>43.1</td>
<td>62.5</td>
<td>192.8</td>
<td>2.0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>linux3 (Mbytes)</td>
<td>195 192</td>
<td>56.1</td>
<td>62.4</td>
<td>15.7</td>
<td>17.5</td>
<td>162.3</td>
<td>237.8</td>
<td>740.0</td>
<td>7.5</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(%)</td>
<td>14.5</td>
<td>16.1</td>
<td>4.1</td>
<td>4.5</td>
<td>42.0</td>
<td>61.5</td>
<td>191.4</td>
<td>1.9</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>gentoo2 (Mbytes)</td>
<td>280 273</td>
<td>71.6</td>
<td>72.2</td>
<td>22.8</td>
<td>22.8</td>
<td>163.1</td>
<td>239.9</td>
<td>1201.0</td>
<td>7.5</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(%)</td>
<td>12.9</td>
<td>13.0</td>
<td>4.1</td>
<td>4.1</td>
<td>29.5</td>
<td>43.3</td>
<td>215.9</td>
<td>1.4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>install (Mbytes)</td>
<td>674 688</td>
<td>202.2</td>
<td>226.7</td>
<td>51.7</td>
<td>58.0</td>
<td>160.9</td>
<td>242.4</td>
<td>2750.0</td>
<td>7.5</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(%)</td>
<td>14.9</td>
<td>16.6</td>
<td>3.8</td>
<td>4.3</td>
<td>11.8</td>
<td>17.8</td>
<td>202.0</td>
<td>0.6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>genome (Mbytes)</td>
<td>1 196 1 137</td>
<td>339.0</td>
<td>120.0</td>
<td>90.4</td>
<td>41.4</td>
<td>161.0</td>
<td>240.3</td>
<td>na</td>
<td>7.5</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(%)</td>
<td>14.5</td>
<td>5.1</td>
<td>3.9</td>
<td>1.8</td>
<td>6.9</td>
<td>10.3</td>
<td>na</td>
<td>0.3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>combine (Mbytes)</td>
<td>1 500 1 443</td>
<td>391.0</td>
<td>423.1</td>
<td>105.6</td>
<td>110.2</td>
<td>161.3</td>
<td>242.4</td>
<td>na</td>
<td>7.5</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(%)</td>
<td>13.3</td>
<td>14.4</td>
<td>3.6</td>
<td>3.7</td>
<td>5.5</td>
<td>8.2</td>
<td>na</td>
<td>0.3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>web (Mbytes)</td>
<td>1 907 1 907</td>
<td>272.0</td>
<td>185.1</td>
<td>105.5</td>
<td>66.6</td>
<td>163.8</td>
<td>236.9</td>
<td>na</td>
<td>7.5</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(%)</td>
<td>7.1</td>
<td>4.9</td>
<td>2.8</td>
<td>1.7</td>
<td>4.3</td>
<td>6.2</td>
<td>na</td>
<td>0.3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 3.8: Maximum memory used in kB, and as a percentage of the sum of the input and reference file sizes.
### Table 3.9: Compression time (seconds) for encoding and compressing

<table>
<thead>
<tr>
<th>File pair</th>
<th>input</th>
<th>reference</th>
<th>bzip2</th>
<th>samba</th>
<th>gentoo1</th>
<th>gcc2</th>
<th>linux1</th>
<th>linux2</th>
<th>linux3</th>
<th>gentoo2</th>
<th>install</th>
<th>genome</th>
<th>combine</th>
<th>web</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>n</td>
<td>1024</td>
<td>0.634</td>
<td>0.517</td>
<td>0.517</td>
<td>0.517</td>
<td>0.517</td>
<td>0.517</td>
<td>0.517</td>
<td>0.517</td>
<td>0.517</td>
<td>0.517</td>
<td>0.517</td>
<td>0.517</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>File pair</th>
<th>input</th>
<th>reference</th>
<th>bzip2</th>
<th>samba</th>
<th>gentoo1</th>
<th>gcc2</th>
<th>linux1</th>
<th>linux2</th>
<th>linux3</th>
<th>gentoo2</th>
<th>install</th>
<th>genome</th>
<th>combine</th>
<th>web</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>n</td>
<td>1024</td>
<td>0.634</td>
<td>0.517</td>
<td>0.517</td>
<td>0.517</td>
<td>0.517</td>
<td>0.517</td>
<td>0.517</td>
<td>0.517</td>
<td>0.517</td>
<td>0.517</td>
<td>0.517</td>
<td>0.517</td>
</tr>
</tbody>
</table>

#### 3.10. Comparison With Other Systems
<table>
<thead>
<tr>
<th>File pair</th>
<th>File size (Mbytes)</th>
<th>Compression time (seconds)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Cobald 1</td>
<td>Cobald 2</td>
</tr>
<tr>
<td>samba</td>
<td>34</td>
<td>1.47</td>
</tr>
<tr>
<td>gentoo1</td>
<td>140</td>
<td>2.34</td>
</tr>
<tr>
<td>gcc2</td>
<td>181</td>
<td>1.54</td>
</tr>
<tr>
<td>gcc1</td>
<td>183</td>
<td>1.55</td>
</tr>
<tr>
<td>linux1</td>
<td>189</td>
<td>2.78</td>
</tr>
<tr>
<td>linux2</td>
<td>192</td>
<td>0.45</td>
</tr>
<tr>
<td>linux3</td>
<td>195</td>
<td>1.46</td>
</tr>
<tr>
<td>gentoo2</td>
<td>280</td>
<td>1.55</td>
</tr>
<tr>
<td>install</td>
<td>674</td>
<td>27.62</td>
</tr>
</tbody>
</table>

**Table 3.10:** Decompression time (seconds) for decompressing and decoding.
only cobald was able to compress files larger than 4 Gbytes.

The memory used by the compressors is summarised in Table 3.8. Xdelta3, hsadelta and bzip2 reached their memory limit for all but the smallest input files. For the four largest pairs of files, Cobald with \( n = 1024 \) used less memory than the other delta compressors, while also obtaining better compression. Cobald with \( n = 256 \) used three to four times more memory than when \( n = 1024 \), which was expected as the index contains approximately four times more \( n \)-gram references. Also, it used more memory than the other delta compressors.

The compression times are shown in Table 3.9, which was measured as the sum of the user time and system time of each process. While xdelta3 is always the fastest, often by several times, it never achieved the best compression. cobald is consistently slower than the other tools, typically two to four times slower. We expect this is partly due to the additional features of the data structures included for evaluation and comparison, and the time would be improved in an optimized implementation. The results show that cobald scales in a similar manner to the other tools, and is broadly competitive with them for delta encoding. Finally, the decompression times, shown in Table 3.10, are similar for all tools except for xdelta3, which is significantly faster on many pairs of files.

3.11 Discussion

From the comparison with other delta compression systems, it can be seen that cobald is an effective delta encoding system achieving compression close to or better than existing systems, especially for files larger than 600 Mbytes. The decompression times are similar, and with content-defined selection, cobald used substantially less memory than the other systems. The encoding time for cobald is two to three times slower, which is primarily due to the number of external
reads of collection data during encoding, although some attention to optimization should yield a substantial improvement.

On the data collections tested, we found that \( n \)-gram lengths of 256 and 1024 gave the best trade-off between good compression and resource use. When \texttt{cobald} is used with \texttt{7-zip} with these \( n \)-gram lengths, the compression was improved between 5 and 20 times. Further work to find a more compact representation of \( n \)-gram references in the index and index file is needed for \( n = 64 \) to be practical. Furthermore, content-defined selection of fingerprints provides a large reduction in memory use and a some speed improvement compared to fixed-offset selection, at the cost of a larger index file as more fingerprints need to be added to achieve the same compression.

With fixed-offset selection, memory use and encoding time are linear with the collection size. Using content-defined selection improves this as \( n \)-grams from duplicate data can be identified and discarded. This process is investigated in detail in Section 4.
3.11. Discussion
Chapter 4

CBC with Compressed Collections

In Chapter 3, we compressed files with respect to an uncompressed collection. However, the storage space needed for large collections could become a significant impediment, and it would be advantageous if CBC could be performed when the collection is compressed. We have already seen how this may be practical using cobald. Specifically, when using content-defined selection, the cobald algorithm inserts only the first occurrence of a duplicated chunk into the index in memory, and later occurrences are never referred to in delta file COPY blocks. Consequently, the effectiveness of CBC would not be reduced if the subsequent chunk occurrences were removed from the collection. This is easily achieved using cobald by storing the delta files instead of the original collection files.

In this chapter, we investigate using cobald to compress the collection, then performing CBC with respect to this compressed collection of delta files. The scheme follows the general approach outlined in Section 2.9: an iterative process of encoding a file, inserting the delta into the collection index, encoding the next file with respect to the first delta, inserting the new delta into the index, and
so on. The result is a compressed representation of the collection as delta files that contain references to matching strings in other deltas. This process can be thought of as an alternative method of de-duplication: the long duplicate strings in the collection files are replaced with references to a single occurrence elsewhere in the collection. But there are key differences from conventional de-duplication algorithms: the files are not partitioned into separate chunks prior to the matching process, and the string matches are confirmed with a byte-by-byte comparison, rather than assumed to match on the basis of matching fingerprints.

The encoding and compression pipeline for compressing a collection using cobald is shown in Figure 4.1. It is similar to the CBC pipeline in Figure 3.1, with the addition of the insertion process for adding the delta file to the fingerprint index and a link indicating that the delta file is stored in the collection. This highlights a key difference from the work described in Chapter 3: CBC involves encoding and compression with respect to a static collection that has been constructed and indexed at some prior time, whereas here we are concerned with the collection construction process, and are investigating how the delta encoding algorithms result in a more compact and efficient representation of the collection.

Retrieval of files from the collection is the same as decoding of files compressed using CBC, as described in Chapter 3: the delta file COPY blocks are replaced by the strings they refer to, which are retrieved from the collection files. As the delta files are added to the index, rather than the input file, each COPY block can be decoded by reading matching string directly from the collection without needing to decode delta file. To understand why this approach is preferred, consider the alternative of inserting the input files into the index before they are encoded. To retrieve a string match, the delta file stored in the collection must be decoded to retrieve the original input file. But part of the matching string may have been encoded as a COPY block referring to another input files, which must also be
Chapter 4. CBC with Compressed Collections

Figure 4.1: Functional diagram of the collection compression scheme implemented by cobald. The pipeline contains three stages: the encode process performs a delta encoding of the input using references to multiple collection files. The insert operation inserts references to n-grams from the delta file into the fingerprint index and stores the delta in the collection. The compress operation compresses the delta using a conventional compression utility.

decoded. The result may be a long cascade of COPY blocks from multiple delta files which are decoded in order to retrieve the original string. Clearly, decoding will be more efficient from a collection where the index contains references to delta files.

The encoding algorithms from Chapter 3 could be used without modification to compress collections. However, many of the n-grams selected for index insertion would contain data from COPY blocks. Inserting COPY block data into the index would allow the possibility of a COPY block referring to a string containing another COPY block, and consequently decoding a delta file may required decoding of a cascade of other delta files. COPY blocks are eliminated from the index by modifying the cobald insertion algorithm so that only n-grams in delta file ADD blocks are indexed, achieving a further reduction in the index size. A worked example of compressing a collection is shown in Figure 4.2.

There are additional benefits from indexing the delta files, compared to indexing the input files. As the delta files are smaller and the index file size is linear
Figure 4.2: Example of encoding and inserting two files into a collection. The shaded arrows indicate the data referred to by COPY and ADD blocks in the delta files. The square brackets indicate n-grams whose fingerprints are appended to the index file during insertion. Each of the indexed n-grams is also listed next to its entry in the index file.

File1 contains a duplicate string, CDEFGHI, and the second occurrence is encoded in file1.delta as a COPY block. The remainder of file1.delta is ADD blocks encoding the data that could not be matched. file1.delta is indexed by appending its fingerprints to the index file.

File2 has two substrings that also occur in file1.delta. GHIKLM is matched with IJL, the third entry in the index file. The full match is found by extending forwards and backwards. The other match, QRSTU, is not found during encoding as it does not contain a complete n-gram listed in the index, but instead spans the boundary between two indexed n-grams. File2.delta is then indexed. Note that the second n-gram has a duplicate fingerprint with the second n-gram from file1.delta. This is a hash collision as the two underlying strings differ, so a new collision set identifier is assigned.
with the quantity of data being indexed, the index file is expected to be significantly smaller, resulting in a smaller index in memory when using fixed-offset \( n \)-gram selection. However, this does not result in a corresponding reduction in the size of the index in memory when content-defined selection is used as references to duplicate chunks are already detected and eliminated. Note that this is not a criticism of content-defined selection, but rather highlights that indexing the deltas is another method for removing references to duplicate strings from the index, and allows fixed-offset selection to obtain a similar memory reduction to content-defined selection. Overall, this is perhaps the key insight of our compression scheme: as files are encoded using the first string match found, any additional references to the matching string in the index will never be used, and the memory used by the algorithms can be reduced by eliminating references to duplicates without significantly affecting the compression achieved. Furthermore, while encoding with the first match might miss other much longer matches in the collection, there is a high probability that the longer match will be found later in the encoding process and then fully recovered by extending backwards. So, while we did not explicitly search for the longest match in a greedy manner, the delta file is likely to be encoded using long matches where they exist.

Collection compression in this manner, in the situation when the alphabet size is unlimited, was analysed by Storer and Szymanski (1982). When the delta employs pointers to a compressed representation of the collection files, which in our case are the deltas of files previously added to the collection, the problem of finding the most compact representation of the collection is \( \text{NP-complete} \). Furthermore, when the matching strings referred to by pointers in the delta files do not themselves contain pointers, referred to as ‘recursive pointers’ in the terminology of Storer, the compressed collection will be at least \( O(\sqrt{R}) \), where \( R \) is the total length of the uncompressed collection files.
The encoding algorithm is necessarily heuristic and, moreover, we are unable to give an improved approximation bound on how closely the compressed collection approaches the most compact representation, beyond the $O(\sqrt{R})$ of Storer and Szymanski (1982). There are two main aspects of the encoding algorithm that resist analysis: the sequence in which files are added to the collection affects the compactness of subsequent delta files; and we have already seen in Section 2.5.2 that delta encoding with a minimum match length is sub-optimal.

### 4.1 Evaluation

The remainder of this chapter is an empirical investigation of the effectiveness of CBC with respect to a collection of deltas. There are two key questions we seek to answer: is storing the delta files an effective method of collection compression, and how effective is CBC with respect to this compressed collection? These questions are investigated separately in two sets of experiments, using several cobald configurations that performed well in Chapter 3. In both experiments, the Genome and Web-snapshots data collections from Chapter 3 are used, and the trials are run on the same computer system.

The first experiment investigates collection compression. Compression effectiveness is evaluated by comparing the storage space, computing time and memory required by cobald with the resources required to compress the collection using a single-file compressor. Four compressors were used in the trials, gzip, bzip2, 7-zip and xz. Detailed results are provided using gzip as it is the established practice for several large genome data repositories, and also using 7-zip as it achieved similar compression to xz and was superior to gzip and bzip2. Additionally, the collections were compressed using 7-zip in archive mode, in which inter-file similarity is used to compress files which are passed to it in a single
Compression was measured using the total storage space needed to store the compressed collections. For the compression achieved by cobald, this was the sum of the delta files sizes and the index file size. The compressed storage space was expressed as a percentage of the total size of the uncompressed collection files, to obtain the measure of the compression.

Results are also presented for the sizes of the compressed delta files following the second-stage compression. This allows us to quantify the compression improvement achieved by cobald as a pre-filter to single-file compression. The compression improvement is expressed as the ratio of the storage space needed by single-file compression without cobald, divided by the storage space needed after the deltas were compressed by the second stage. A compression improvement ratio of 1 indicates that cobald did not achieve any improvement, and higher numbers show an increasing benefit obtained by cobald.

The second experiment evaluated CBC with respect to a collection of deltas. The aim of this experiment is not to evaluate the compression achieved, which was thoroughly investigated in the first experiment, but to test whether the CBC requirements stated in Chapter 1 are satisfied. The results are presented as charts of compression time versus input file size, and memory use versus collection size. The results are evaluated by inspecting the linearity of the observed trends.

### 4.2 Results: Compressed Collections

The Web snapshots and Genome collections were compressed using cobald with several combinations of parameters that were found to be effective in Chapter 3. The configurations are listed in 4.1. Two content-defined configurations are tested for each $n$-gram length, having maximum chunk lengths two and five times the
4.2. Results: Compressed Collections

Table 4.1: Configurations of cobald for the collection compression experiment. The parameters are the $n$-gram length, $n$, the number of 1 bits in the chunk selection mask, $s$, and the maximum chunk length, $m$. Note that $2^s = n$ for each of the content-defined configurations. The maximum chunk lengths are 2 and 5 times the $n$-gram length, as indicated in the configuration label.

<table>
<thead>
<tr>
<th>Test Label</th>
<th>Selection method</th>
<th>Parameters $n$</th>
<th>$s$</th>
<th>$m$</th>
<th>Mean separation</th>
</tr>
</thead>
<tbody>
<tr>
<td>F8</td>
<td>fixed-offset</td>
<td>256</td>
<td>-</td>
<td>-</td>
<td>256</td>
</tr>
<tr>
<td>CD8.2</td>
<td>content-defined</td>
<td>256</td>
<td>8</td>
<td>512</td>
<td>≈ 256</td>
</tr>
<tr>
<td>CD8.5</td>
<td>content-defined</td>
<td>256</td>
<td>8</td>
<td>1280</td>
<td>≈ 256</td>
</tr>
<tr>
<td>F10</td>
<td>fixed-offset</td>
<td>1024</td>
<td>-</td>
<td>-</td>
<td>1024</td>
</tr>
<tr>
<td>CD10.2</td>
<td>content-defined</td>
<td>1024</td>
<td>10</td>
<td>2048</td>
<td>≈ 1024</td>
</tr>
<tr>
<td>CD10.5</td>
<td>content-defined</td>
<td>1024</td>
<td>10</td>
<td>5120</td>
<td>≈ 1024</td>
</tr>
</tbody>
</table>

Table 4.1: Configurations of cobald for the collection compression experiment. The parameters are the $n$-gram length, $n$, the number of 1 bits in the chunk selection mask, $s$, and the maximum chunk length, $m$. Note that $2^s = n$ for each of the content-defined configurations. The maximum chunk lengths are 2 and 5 times the $n$-gram length, as indicated in the configuration label.

expected chunk length. The delta files were added to the collection one at a time, by encoding the input file then inserting the delta into the hash index, in the sequence listed in the appendix. After the collections were completely encoded in this manner, the delta files were compressed by several standard compression utilities to obtain the compressed collection.

Table 4.2 shows the total storage space used for the compressed collections, comprising the compressed delta files and the index file. These results are summarised in Table 4.3 showing the compression improvement achieved when cobald is used compared to compression without the cobald first stage. Compressing with cobald is significantly more effective in all cases. With the best cobald configuration (F8), four times more compression was achieved using cobald than using the best single-file compressors (xz, 7-zip) alone for the Web snap-shots collection, and more than 17 times for the Genome collection. While the other cobald configurations achieve a little less improvement, the least effective configuration (CD10.5) is still worthwhile, achieving 3.2 times more compression on the Web snap-shots collection, and 10.2 times for the Genome collection. The larger improvement observed with the Genome collection is primarily because it
<table>
<thead>
<tr>
<th>Compression method</th>
<th>Input files</th>
<th>cobald encoding method</th>
<th>F8</th>
<th>CD8.2</th>
<th>CD8.5</th>
<th>F10</th>
<th>CD10.2</th>
<th>CD10.5</th>
</tr>
</thead>
<tbody>
<tr>
<td>Web snap-shots:</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>uncompressed</td>
<td>(Mbytes)</td>
<td></td>
<td>55</td>
<td>344</td>
<td>3778</td>
<td>4594</td>
<td>4872</td>
<td>6097</td>
</tr>
<tr>
<td></td>
<td>(%)</td>
<td></td>
<td>100.00</td>
<td>6.83</td>
<td>8.30</td>
<td>8.80</td>
<td>11.02</td>
<td>13.03</td>
</tr>
<tr>
<td>gzip</td>
<td>(Mbytes)</td>
<td></td>
<td>6539</td>
<td>933</td>
<td>1067</td>
<td>1058</td>
<td>1047</td>
<td>1199</td>
</tr>
<tr>
<td></td>
<td>(%)</td>
<td></td>
<td>11.82</td>
<td>1.69</td>
<td>1.93</td>
<td>1.91</td>
<td>1.89</td>
<td>2.17</td>
</tr>
<tr>
<td>bzip2</td>
<td>(Mbytes)</td>
<td></td>
<td>3955</td>
<td>711</td>
<td>802</td>
<td>794</td>
<td>756</td>
<td>859</td>
</tr>
<tr>
<td></td>
<td>(%)</td>
<td></td>
<td>7.15</td>
<td>1.29</td>
<td>1.45</td>
<td>1.44</td>
<td>1.37</td>
<td>1.55</td>
</tr>
<tr>
<td>7-zip</td>
<td>(Mbytes)</td>
<td></td>
<td>2742</td>
<td>672</td>
<td>753</td>
<td>739</td>
<td>683</td>
<td>761</td>
</tr>
<tr>
<td></td>
<td>(%)</td>
<td></td>
<td>4.95</td>
<td>1.21</td>
<td>1.36</td>
<td>1.34</td>
<td>1.23</td>
<td>1.38</td>
</tr>
<tr>
<td>xz</td>
<td>(Mbytes)</td>
<td></td>
<td>2672</td>
<td>667</td>
<td>747</td>
<td>732</td>
<td>674</td>
<td>751</td>
</tr>
<tr>
<td></td>
<td>(%)</td>
<td></td>
<td>4.83</td>
<td>1.21</td>
<td>1.35</td>
<td>1.32</td>
<td>1.22</td>
<td>1.36</td>
</tr>
<tr>
<td>7-zip archive</td>
<td>(Mbytes)</td>
<td></td>
<td>2741</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>(%)</td>
<td></td>
<td>4.95</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Genome:</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>uncompressed</td>
<td>(Mbytes)</td>
<td></td>
<td>23902</td>
<td>1319</td>
<td>1731</td>
<td>1832</td>
<td>1592</td>
<td>2158</td>
</tr>
<tr>
<td></td>
<td>(%)</td>
<td></td>
<td>100.00</td>
<td>5.52</td>
<td>7.24</td>
<td>7.66</td>
<td>6.66</td>
<td>9.03</td>
</tr>
<tr>
<td>gzip</td>
<td>(Mbytes)</td>
<td></td>
<td>5794</td>
<td>347</td>
<td>458</td>
<td>473</td>
<td>401</td>
<td>540</td>
</tr>
<tr>
<td></td>
<td>(%)</td>
<td></td>
<td>24.24</td>
<td>1.45</td>
<td>1.92</td>
<td>1.98</td>
<td>1.68</td>
<td>2.26</td>
</tr>
<tr>
<td>bzip2</td>
<td>(Mbytes)</td>
<td></td>
<td>5662</td>
<td>292</td>
<td>389</td>
<td>404</td>
<td>369</td>
<td>501</td>
</tr>
<tr>
<td></td>
<td>(%)</td>
<td></td>
<td>23.69</td>
<td>1.23</td>
<td>1.63</td>
<td>1.69</td>
<td>1.54</td>
<td>2.10</td>
</tr>
<tr>
<td>7-zip</td>
<td>(Mbytes)</td>
<td></td>
<td>4843</td>
<td>279</td>
<td>369</td>
<td>381</td>
<td>321</td>
<td>441</td>
</tr>
<tr>
<td></td>
<td>(%)</td>
<td></td>
<td>20.26</td>
<td>1.17</td>
<td>1.55</td>
<td>1.60</td>
<td>1.34</td>
<td>1.85</td>
</tr>
<tr>
<td>xz</td>
<td>(Mbytes)</td>
<td></td>
<td>4818</td>
<td>278</td>
<td>369</td>
<td>381</td>
<td>320</td>
<td>440</td>
</tr>
<tr>
<td></td>
<td>(%)</td>
<td></td>
<td>20.16</td>
<td>1.17</td>
<td>1.55</td>
<td>1.59</td>
<td>1.34</td>
<td>1.84</td>
</tr>
<tr>
<td>7-zip archive</td>
<td>(Mbytes)</td>
<td></td>
<td>4576</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>(%)</td>
<td></td>
<td>19.14</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Table 4.2:** Total size of the compressed collections, including the index file, using the various cobald test configurations. The inputs column is the sizes of the collection files compressed individually without using cobald as a preliminary stage. The remaining columns are the total size of the delta files and the index file for each combination of cobald test configuration and second-stage compression utility. The percentage rows show the compressed collection size as a proportion of the uncompressed collection size.
4.2. Results: Compressed Collections

<table>
<thead>
<tr>
<th>Compression method</th>
<th>cobald encoding method</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>F8</td>
</tr>
<tr>
<td>Web snap-shots:</td>
<td></td>
</tr>
<tr>
<td>gzip</td>
<td>7.0</td>
</tr>
<tr>
<td>bzip2</td>
<td>5.6</td>
</tr>
<tr>
<td>7-zip</td>
<td>4.1</td>
</tr>
<tr>
<td>xz</td>
<td>4.0</td>
</tr>
<tr>
<td>Genome:</td>
<td></td>
</tr>
<tr>
<td>gzip</td>
<td>16.7</td>
</tr>
<tr>
<td>bzip2</td>
<td>19.3</td>
</tr>
<tr>
<td>7-zip</td>
<td>17.4</td>
</tr>
<tr>
<td>xz</td>
<td>17.3</td>
</tr>
</tbody>
</table>

Table 4.3: Compression improvement from using cobald. The numbers shown are the compression improvement ratio: the size of the collection compressed without cobald (the third column in Table 4.2) divided by the size of the delta files compressed with cobald followed by the second stage (columns 4 to 9 in Table 4.2).

contains more data sets: there are 20 data sets in Genome, and only 8 in Web snap-shots. As data sets are added to a collection, the compression improvement increases as there are more opportunities to matching data.

In most cases, the various cobald configurations perform as expected, with the compressed delta sizes in the order F8 < CD8.2 < CD8.5 < F10 < CD10.2 < CD10.5. However, an interesting anomaly occurred when the compressed deltas for Web snap-shots CD8.5 < CD8.2. At first this appeared to be an erroneous result as the uncompressed delta file sizes were in the correct order CD8.2 < CD8.5. Detailed inspection of the individual files sizes showed that, after the first data set was encoded, subsequent data sets generated larger delta files using CD8.5, but these were compressed to smaller sizes compared to the CD8.2 compressed deltas. These delta files share some common characteristics that suggest an explanation. The CD8.5 deltas had fewer COPY blocks than the CD8.2 deltas, and also had more data encoded as references into the collection files. The CD8.2 deltas were
smaller overall as they had less data encoded as ADD blocks, and also had more
data encoded as references to earlier in the input file. We hypothesise that the
second stage was able to compress ADD blocks better than COPY blocks, which
we consider to be a reasonable expectation as the COPY blocks contain integers
that are likely to be unique to each block. It is not clear why the CD8.2 deltas
had substantially more COPY blocks than the CD8.5 delta in this case, but it
is not surprising that less compression was obtained from files with more COPY
blocks.

Also shown in Table 4.2 are the results of compressing the collections using
7-zip in archive mode. Rather than compressing the files individually, they were
passed to 7-zip in a single batch, allowing 7-zip to find inter-file matching
data, to the extent that the sliding window included multiple files. With the
Web snap-shots collection, the compressed deltas were the same size as when
they were compressed individually, achieving no compression improvement, while
for the Genome collection compression improved by approximately 5%. This
is likely to be because the files in Web snap-shots are mostly 2 Gbytes in size,
which is greater than the 7-zip sliding window, while there are many smaller files
in the Genome collection. Nonetheless the deltas compressed using the cobald
first stage are 17 times smaller, and we conclude that, for large collections, a
compressor based on a sliding window over the concatenated collection files is
inferior to our system, which can find duplicate data from any location in the
collection.

We now look at the compression of each data set in the collections. Figure 4.3
shows the sizes of the compressed delta files as each data set was added to the
cobald index. The files were compressed using the best cobald configuration,
F8. For both collections, the compressed deltas from the first data set were much
larger than subsequent data sets as the index was empty. Once the index was
4.2. Results: Compressed Collections

(i) Web snap-shots

![Graph showing relative data set sizes for web snap-shots]

(ii) Genome

![Graph showing relative data set sizes for genome]

Figure 4.3: Relative sizes of the compressed data sets in the collection compressed with test configuration F8. ‘Un-encoded, gzip’ and ‘un-encoded, 7-zip’ are the sizes of the data set files compressed individually using gzip and 7-zip. ‘Cobald, uncompressed’ indicates the sizes of the delta files for each data set, which were encoded after the deltas from the previous data sets were inserted into the cobald index. ‘Cobald, gzip’ and ‘cobald, 7-zip’ indicate the sizes of the delta files following the second-stage compression of the deltas by gzip and 7-zip.
Chapter 4. CBC with Compressed Collections

populated with the first data set, the delta sizes of the following data sets were fairly stable. The delta files for Genome release-49 and release-56 were noticeably larger than the other data sets because more new data was added to the collection in these releases, while the other releases have very little change from their predecessors. Figure 4.3 neatly demonstrates the advantage of cobald over the single-file compressors. After the first data set, cobald achieved a large improvement in compression, while the single-file compressors compressed each data set similarly to the first. The contribution of cobald to the total compression is the difference between the compressed delta size and the individually compressed data set sizes. For the second and following data sets, using 7-zip as the second stage, the improvement with the Web snap-shots collection was 5 to 10 times, and with the Genome collection was 10 to 25 times. Note that the index file size is not included in the compressed delta sizes, so the overall advantage from using cobald is slightly less. The compressed delta sizes are shown cumulatively in Figure 4.4, which clearly shows the increasing advantage of using cobald as additional data sets are added to the collection.

The cumulative size of the compressed delta for each of the cobald test configurations is shown in Figure 4.5. The results for the Web snap-shots collection conform to the expected ranking of F8 < CD8.2 < CD8.5 < F10 < CD10.2 < CD10.5, with CD10.5 deltas approximately two times larger than the F8 deltas. However, the results for the Genome data set were more interesting. They conformed to the expected order for the first 3 data sets, but then the F10, and to a lesser extent CD10.2 and CD10.5, increased with a lower slope as these configurations generated smaller deltas. This is unexpected: the configurations using a shorter $n$-gram length (F8, CD8.2 and CD8.5 have $n = 256$) are able to find more matching strings than the configurations using longer $n$-grams (F10, CD10.2 and CD10.5 have $n = 1024$). The effect emerges as a result of adding
4.2. Results: Compressed Collections

(i) Web snap-shots

![Graph showing cumulative size of compressed collection with test configuration F8. 'Un-encoded, gzip' and 'un-encoded, 7-zip' are the sizes of files compressed individually using gzip and 7-zip. 'Cobald, uncompressed' indicates the cumulative size of the delta files in the collection, which were encoded after the deltas from the previous data sets were inserted into the cobald index. 'Cobald, gzip' and 'cobald, 7-zip' indicate the cumulative sizes of the delta files following the second-stage compression by gzip and 7-zip.](image)

(ii) Genome

![Graph showing cumulative size of compressed collection with test configuration F8. 'Un-encoded, gzip' and 'un-encoded, 7-zip' are the sizes of files compressed individually using gzip and 7-zip. 'Cobald, uncompressed' indicates the cumulative size of the delta files in the collection, which were encoded after the deltas from the previous data sets were inserted into the cobald index. 'Cobald, gzip' and 'cobald, 7-zip' indicate the cumulative sizes of the delta files following the second-stage compression by gzip and 7-zip.](image)
Figure 4.5: Cumulative sizes of the delta files as data sets are added to the collection for the various test configurations.
delta files to the index. There are some strings in the early input files that were not found to contain matches using \( n = 1024 \), but which were partially matched using \( n = 256 \). These strings then occurred unchanged in the following data sets. The \( n = 1024 \) configurations were able to match these subsequent occurrences with the first occurrence as it was encoded entirely in a ADD block. However the \( n = 256 \) configurations could only match the smaller substring which was not matched in the earlier delta. This seemingly un-intuitive outcome reminds us that the compression achieved by the algorithm depends on the sequence in which files are added to the collection, and also shows that poor initial compression can lead to better compression of subsequent data.

**Computation time** The computation time is shown in Figure 4.6, for the cobald configuration F8, compared with compression by the single-file compressors 7-zip and gzip. For both collections, the cobald encoding time and the second-stage compression time of the first data set was longer than for subsequent data sets as less compression was obtained and the delta files were larger. For the second and subsequent data sets, most of the time was for cobald encoding, and the second-stage compression of the delta files took less than 10% additional time. In each collection, compression of the second and subsequent data sets was significantly faster than compressing the input files with 7-zip, and two-stage compression was even faster than gzip for the Genome collection, although not for the Web snap-shots collection. This is an important result, showing that two-stage compression can use less computing resources than compression of individual files, while also achieving much greater compression.

Figure 4.7 shows the times to encode and insert the data sets using each cobald configuration. Fixed-offset selection with \( n = 1024 \) (F10) was the slowest for both collections, while content-defined selection with a long maximum chunk
Chapter 4. CBC with Compressed Collections

(i) Web snap-shots

![Graph](image1)

(ii) Genome

![Graph](image2)

**Figure 4.6:** Computation time for each data set of the compressed collection with test configuration F8. ‘7-zip of input’ and ‘gzip of input’ are the times to compress the files individually using 7-zip and gzip. ‘Cobald’ indicates the time to encode the delta files, and insert them into the index. The delta files were encoded after the deltas from the previous data sets were inserted into the cobald index. ‘Cobald + gzip’ and ‘cobald + 7-zip’ indicate the total time to create the deltas and compress them in the second-stage compression by gzip and 7-zip. The time for the second-stage compression is the difference between these lines and the ‘cobald’ encode line.
4.2. Results: Compressed Collections

(i) Web snap-shots

![Web snap-shots graph]

(ii) Genome

![Genome graph]

Figure 4.7: Time to encode and insert the data sets for each cobald configuration. The times are shown cumulatively so that each point shows the time to encode and insert all of the preceding data sets.
lunk length (CD8.5 and CD10.5) were the fastest.

**Memory**  The maximum memory used to compress the collection is shown in Figure 4.8. The memory used by the cobald encoding stage gradually increased as files were added to the collection, which is due to the growth of the index. As delta files are added, rather than the input files, the index grows slowly when the new files are similar to the files already in the collection. This effect is seen with each collection, where the increase in memory use was much slower than the increase in collection size. However, there was a sudden increase for the Genome collection when the collection size was approximately 15 Gbytes, which is caused by the large proportion of new data in release-49.

The memory used by the second-stage compression using 7-zip is also shown in Figure 4.8. In the configuration used in these experiments, the memory used by 7-zip is limited to approximately 380 Mbytes. The maximum memory was required for all data sets in the Web snap-shot collection, and the two data sets having the largest deltas in the Genome collection. Compared to single-file compression using 7-zip, cobald encoding used an additional 20% memory for the Web snap-shots collection, and less than one third of the memory for the Genome collection. This difference is mainly because the index for the Web snap-shots collection is larger than for the Genome collection. The results with gzip are not shown as its memory use is limited to less than 1 Mbyte.

Figure 4.9 compares the maximum memory used to encode with the various cobald configurations. There was a large difference between the two $n$-gram lengths, with $n = 256$ using approximately 2.5 times the memory used with $n = 1024$. There was a much smaller difference in the memory used between fixed-offset and content-defined selection. As stated earlier, the large difference observed in Chapter 3 has largely been eliminated by inserting the delta files into
4.2. Results: Compressed Collections

(i) Web snapshots

![Graph showing memory usage for Web snapshots.]

(ii) Genome

![Graph showing memory usage for Genome.]

**Figure 4.8:** Maximum memory used to compress each data set of the collection with test configuration F8. ‘7-zip of input’ is the maximum memory used by 7-zip when compressing the input files individually. Note that in chart (i), the green line is overlaid by the purple line as 7-zip reached its memory limit when compressing both the input and delta files. ‘Cobald’ indicates the maximum memory used by cobald when encoding the files in the data set. Insertion of the delta files into the collection is not shown as it always uses less memory than encoding. ‘7-zip of deltas’ shows the memory used by 7-zip when compressing the deltas in the second stage. Memory use for gzip is not shown as it always used less memory than cobald.
the index. This is because the duplicate strings in the input files, that were likely to be indexed multiple times by fixed-offset selection starting at different offsets, have been removed from the delta files.

4.3 Results: CBC on a Compressed Collection

We now return to the collection-based compression problem, and review the effectiveness of CBC with respect to collections containing delta files. Recall the requirements of a successful CBC system from Chapter 1: the compression time should increase linearly with the input file size, and the memory required should increase less than linearly with the collection size. The following experiments investigate whether these criteria are met when encoding with respect to the Web snapshots and Genome collections that have been indexed as delta files.

To evaluate the relationship between encoding time and input file size, input files were created from the last data set from each collection, week8 in Web snapshots and release-56 in Genome, then encoded using cobald with respect to collections containing the preceding data sets, week1 to week7 for Web snapshots, and release-36 to release-55 for Genome. These collection files had previously been encoded, then their delta files inserted into the index in the manner described earlier in this chapter. To obtain a wide range of input file sizes, files were created by taking a prefix of each desired size from the largest file in the last data set.

The results for encoding time versus input file size are shown in Figure 4.10. Broadly, the encoding time appears to increase linearly with the input file size, although some aspects of the charts suggest some caution against a definitive conclusion. The ratio of largest to smallest input files sizes is only 8 times, and the results of encoding input files an order of magnitude larger are needed to be
4.3. Results: CBC on a Compressed Collection

(i) Web snap-shots

![Graph showing memory usage for web snap-shots](image)

(ii) Genome

![Graph showing memory usage for genome](image)

Figure 4.9: Maximum memory used to encode each data set for each cobald configuration.
(i) Web snap-shots

![Graph showing encode time by cobald versus input file size for different configurations of the cobald encoding parameters for Web snap-shots.]

(ii) Genome

![Graph showing encode time by cobald versus input file size for different configurations of the cobald encoding parameters for Genome.]

**Figure 4.10:** CBC encode time by cobald versus input file size for the different configurations of the cobald encoding parameters.
confident there is no super-linear contribution. As it is, a slight increase in the
slope can be observed for F10 (fixed-offset selection with $n = 1024$) for the larger
files from Web snap-shots. There is also some small variation about a linear trend,
which is expected from our understanding of the encoding algorithm in Chapter 3.
The encoding operation has three modes which are each linear: extending a match
forward, $M_{\text{ext}}$; searching for a match, $M_{\text{search}}$; and backward extension, $M_{\text{back}}$.
Each of these modes has different speeds, with $M_{\text{ext}}$ being faster than $M_{\text{search}}$
and $M_{\text{back}}$. Consequently, the encoding speed depends on the input data, with
highly similar data being encoded faster than dissimilar data. Also, there will be
some departure from linearity due to the specific operating environment of the
computer, such as the increasing number of cache misses for larger files.

The relationship between the encoding times of the various co Baldwin configurations is also complex. In particular, F8 (fixed-offset selection and $n = 256$) is the
fastest algorithm for Web snap-shots, but the slowest for Genome. It is not clear
why this occurred. There are aspects of the F8 algorithm that are both faster and
slower than the other configurations. F8 has the largest index, and also inserts the
most $n$-gram references in the input index, which requires more time. F8 will also
take more time when searching for a match than encoding with content-defined
selection which does not need to query the index at every position. On the other
hand, F8 will find the most matches and perhaps take less time, as extending
forwards is faster than searching for a match. The balance between these trade-
offs will depend on the degree of similarity in the data. The other configurations
are ranked in the expected order for both Genome and Web snap-shots: CD8.5 is
faster than CD8.2, and CD10.5 is faster than CD10.2, for both collections as the
number of maximal-length $n$-grams is lower, and these are searched by querying
each $n$-gram in similar fashion to fixed-offset encoding. Overall, we conclude that
the results are consistent with a linear relationship between encoding time and
input file size, although this is not a conclusive demonstration.

The relation between memory use and collection size was evaluated by encoding a file with cobald using a range of subsets of each collection. The input file was the largest file in the last data from each collection, week8 for Web snapshots, and release-56 for Genome. The collection subsets available to the encoder were created by sequentially encoding each data set and inserting the delta files into the index, following the method described earlier in this chapter. For example, the Web snapshots input file was first encoded with the week1 delta files in the index, then encoded again with the week1 and week2 delta files in the index, and so on until the final encoding used an index containing the deltas of week1, week2, ..., week7.

The results for memory use versus the collection size are shown in Figure 4.11. The broad relationship appears to be sublinear for the Genome collection, and linear for Web snapshots. Again, we need to be cautious as the ratio of the largest to the smallest collection sizes is only 8 (times) for Web snapshots and 20 (times) for the Genome data, and we consider that measurements over a larger range are needed to be confident that the relationship is sublinear. As we described in Chapter 3, there are three large data structures in cobald. For a large collection, the largest structure is the collection index, with size proportional to the total size of the indexed files. The other two data structures, the input index and the match list, are related to the input file size and will be effectively identical for each data point in this experiment. As the collection delta files were inserted into the index, the memory needed is proportional to the amount of unmatched data in the collection. As the collection grows, the collection index will increase with the amount of dissimilar data in the newly inserted files. We can see this in the Genome collection when the collection size increases beyond 15 Gbytes. This data set contains a large quantity of new data, while the preceding data sets
4.3. Results: CBC on a Compressed Collection

(i) Web snap-shots

![Graph showing memory use versus collection size for web snap-shots with different configurations of cobald encoding parameters.](image)

(ii) Genome

![Graph showing memory use versus collection size for genome with different configurations of cobald encoding parameters.](image)

**Figure 4.11:** CBC memory use by cobald versus collection size for the different configurations of the cobald encoding parameters.
were very similar to their predecessors.

In summary, the relationship between memory and collection size depends on the collection data. If, as the collection grows, new files are increasingly similar to the files already in the collection, such as the Genome collection appears to be, then memory use grows sublinearly with the collection size. On the other hand, an influx of dissimilar data will result in a corresponding leap in memory use. Recalling that the purpose of this investigation is to achieve improved collection compression by exploiting inter-file duplication that is not exploited by existing methods, we consider that our algorithm successfully satisfies the requirement for sublinear memory use, within the scope we set in Chapter 1. Another way of viewing this outcome is that our CBC system provides substantial benefit for highly similar collections, and is no worse than other linear systems for collections with low similarity.

4.4 Discussion

In this chapter, we applied the CBC algorithms developed in Chapter 3 to the problem of compressing collections, then tested collection-based compression with respect to a collection compressed in this manner. By first encoding the collection files, then inserting the deltas into the index, the collection is compressed while also allowing a file to be decoded without needing to reconstruct the files referred to in its match references. A small modification was made to the insertion algorithm so that only data in delta file ADD blocks was indexed, achieving a modest reduction in the size of the index. A second compression stage, using a conventional single-file compressor, compressed the delta files.

Compressing the Web snap-shots (55 Gbytes) and Genome (24 Gbytes) collections in this manner achieved a substantial improvement in compression, com-
pared to compression of the files individually with a conventional compressor. The compressed Web snap-shots collection was 4 times smaller, and the Genome collection was almost 17 times smaller, with the F8 cobald configuration than compression by xz and 7-zip of the individual files. Moreover, our two-stage compression required less computing resources, taking less time and less memory with $n = 1024$, although a little more memory with $n = 256$ on the Web snap-shots collection. Results for decompression of files from the collection are not presented as decoding is unchanged from Chapter 3.

While these are noteworthy improvements in compression, there are several aspects of these results that suggest caution when evaluating the cobald algorithms, or when applying the algorithms in other situations. The experiments revealed a complex dependence of the compression achieved and resources needed on the input data. A high degree of similarity between input files is needed to achieve the observed compression improvements. Also, the compression effectiveness varies to a small degree with the sequence in which the input files are inserted into the index.

Caution is also needed when comparing the effectiveness of the algorithm with other CBC and collection compression algorithms. It could be argued that compression of the collection as individual files by standard compression utilities is not informative, as those algorithms were not designed for the purpose of compressing collections of similar files. In particular, we acknowledge that comparison with gzip is not informative regarding the effectiveness of our algorithm as gzip uses comparatively minuscule computing resources. The comparison with gzip is included in this study as compression of individual files by gzip is the common use case employed by many existing collections. To assess the cobald algorithms, we focus on the comparison with 7-zip, which does use similar computing resources, and, along with xz, is the best available single-file compressor.
In particular, comparison with compression by 7-zip in archive mode, which did not achieve a large improvement compared to compressing individual files, is an informative test showing that the cobald algorithm does obtain a significant advantage. Unfortunately, we were unable to compare cobald with other systems. The comparison of delta encoders in Chapter 3 suggests that cobald would be more effective. However, these utilities are not well suited for compression of collections containing tens of gigabytes.

Six combinations of cobald parameters were tested, but we could not identify a single combination as being superior. F8 achieved the greatest compression, but F10, CD10.2 and CD10.5 used less than half as much memory. Encoding with a large maximum n-gram separation (CD8.5 and CD10.5) was a little faster. Selecting the most appropriate combination is a trade-off between compression, time and memory. An important improvement from the results in Chapter 3 is that inserting delta files into the index greatly improves the memory used by fixed-offset n-gram selection, so that there is little advantage from using content-defined selection.

Having compressed our collections, we reconsidered the collection-based compression problem. When compressing with the collection delta files as references, we found that the compression time is linear with the input file size, and the memory used is sublinear with the collection size when new data added to the collection has increasing similarity. These are essential requirements for a practical CBC scheme, as outlined in Chapter 1.

For the moment, the second compression stage is best performed by 7-zip or xz. However, future work on cobald will compress the deltas such that pseudo-random access into the compressed delta is supported, which allows a block of the compressed delta file to be decompressed rather than the entire file. Random access into the collection files is essential so that the encoder can resolve collisions
and extending matching strings, and also for the decoder to retrieve COPY blocks. We expect that this second-stage will not be as effective as 7-zip or xz as it will have a much smaller sliding window, but the compression may nonetheless be complementary with cobald which will have removed the long-range duplicates in the first stage. This issue is discussed further in Chapter 7.

While the memory used by cobald can be reduced by some careful optimisation, another area of future work is algorithms to implement a memory limit. When the limit is reached, fingerprint references must be discarded from the index, which will reduce the compression effectiveness. The algorithms will aim to reduce this impact and achieve a smooth and gradual performance degradation.
Chapter 5

Hash Function Testing

We now turn our attention to the hash functions employed in CBC systems to generate \( n \)-gram fingerprints. As noted previously, re-hashing the entire collection is expensive, perhaps even impractical, and we must expect that the hash function will be employed for the life of the collection. How can we be sure that a hash function is suitable?

The key requirements for a non-cryptographic, non-universal hash function are speed, and pseudo-randomness, which is the subject of this chapter. A non-random distribution of hash values can lead to an increased frequency of hash collisions, causing hash table query time to be greater than \( O(1) \). The mechanism is two-fold: multiple collisions at one or more high-frequency hash values lead to long chains at these entries in the hash table, so that these chains are searched more often, and each search takes more time. Reliable testing of hash function pseudo-randomness is important for numerous applications employing hash-based data structures. In the specific case of our CBC scheme, the collision resolution step involves an external read of an \( n \)-gram from the collection, and unnecessary collisions will rapidly degrade system responsiveness.

Collision testing is the principal evaluation methodology for non-cryptographic,
non-universal hash functions, as described in Section 2.2.1. This involves observing the collisions obtained when hashing a suite of input keys having a similar distribution to the keys expected in the application. The selection of keys is important; as the hash function is not truly random, it cannot generate a random distribution of hash values for all possible sets of keys. The best we can aim for is that it behaves randomly for all of the keys it may be expected to hash.

Collision testing is a form of black-box testing, where the evaluation is informed solely by the response to various inputs. The test method requires no knowledge of the internal operation of the hash function, and allows hash functions to be compared on equal terms, before assessing more subjective considerations such as their design and implementation.

Collision test outcomes are usually interpreted comparatively, ranking the hash functions from least collisions (best) to most collisions (worst) (Mulvey, 2007; Appleby, 2008; Estébanez et al., 2013). Interpretation of these results would be improved by including the significance of the test against the hypothesis that the distribution of hash values is truly random. The results could be supplemented with a $p$-value indicating the probability of the outcome being obtained by a random allocation of balls into bins. With this additional information, it may be that several of the functions at the top of the comparative ranking obtained significant $p$-values, indicating that they are statistically equivalent to a random function for that test. Then, rather than evaluate one function as superior to the others, we know they are all satisfactory. In short, we desire an absolute test against a concrete hypothesis, rather than a relative comparison with other functions.

An additional element of statistical testing is the possibility of accepting a bad hash function as good: a type II error. In comparative collision tests, the probability of type II errors is not usually published. However, an accurate mea-
sure of the statistical power of a test is necessary to determine how much testing is required in order to support the desired conclusions.

In this chapter, we present a collision testing method which evaluates, within well-defined statistical confidence limits and sensitivity, the behaviour of a hash function compared to a random allocation of balls into bins, as well as allowing comparative evaluation of multiple functions. Additionally, the test procedure employs multiple trials on subsets of the input key space, which may detect poor outcomes for some subsets that would not be found if the keys were tested in a single large set. A common example is sets of keys having a common prefix or suffix, such as “http://” or “@gmail.com”.

An alternate use for the test is to investigate properties of the hash function, such as balance, avalanche and resilience. This can be done by testing with keys having a specific distribution. In this chapter, we use randomly generated keys applied to unbalanced hash functions to demonstrate that the test is effective. In Chapter 6 we use key sets having different permutations of an alphabet to investigate the resilience of the hash functions.

5.1 Black-Box Collision Test

The test procedure involves multiple trials in which 256 unique keys are hashed and inserted into a hash table of size 128. Typically, a hash function generates a much larger hash value, so the hash table entry is determined by selecting seven bits from the hash to create a selected hash. The bit positions are selected randomly for each trial, then the same bit positions are used for the 256 selected hashes within each trial. The length of the longest probe sequence (LLPS) is recorded for each trial, and the mean of these measurements is the statistic used to test the hypothesis that the hash function generates a random distribution of
hash values. LLPS was chosen as the test statistic as it directly measures the aspect of non-randomness that impacts on system performance: the creation of long chains in a hash table. Also, the theoretical distribution of LLPS for the ideal case of randomly allocated balls into bins is known for small hash tables (Reviriego et al., 2011), allowing the statistical significance and power of the test to be determined. This improves on the previous method of performing a single collision test on a larger hash table where the theoretical distribution of the collision measure is not known exactly, which prevents accurate calculation of the test significance. While it is not usual practice to insert more keys than the size of the hash table, we use twice as many keys to obtain a wider distribution of LLPS values, and to reduce the likelihood of a biased function achieving an apparently random outcome.

Table 5.1 shows the theoretical probability distributions of LLPS for several combinations of balls and bins calculated using Matlab software provided by Reviriego et al. (2011). The distribution for 256 balls into 128 bins is illustrated in Figure 5.1. Distributions for larger numbers of balls and bins were not obtained as the solution involves calculation of large combinatoric quantities and the calculation time increases rapidly. The calculation of 256 balls into 128 bins took 3 hours, but 256 balls into 256 bins did not complete after 2 days of computation. Reviriego et al. (2011) give \( E[\text{LLPS}] \) values for several larger combinations, but do not provide the distributions. Consequently, 7-bit selected hashes are used in the test.

The distribution of the mean of the LLPS measurements, \( \bar{X} \), from \( T > 30 \) trials is approximated by the normal distribution \( N(\mu_0, \sigma^2/T) \) (central limit theorem), where \( \mu_0 \) is the mean, and \( \sigma^2 \) is the variance, of the theoretical LLPS distribution. The test significance is determined using a one-sided, single-sample \( t \)-test. The test hypothesis is that the hash distribution is produced by a random
### Table 5.1: Probability distribution of LLPS when 256 balls are randomly allocated to 124, 126, 127 and 128 bins, calculated using the method of Reviriego et al. (2011).

<table>
<thead>
<tr>
<th>LLPS</th>
<th>124 bins</th>
<th>126 bins</th>
<th>127 bins</th>
<th>128 bins</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0.000000</td>
<td>0.000000</td>
<td>0.000000</td>
<td>0.000000</td>
</tr>
<tr>
<td>2</td>
<td>0.000000</td>
<td>0.000000</td>
<td>0.000000</td>
<td>0.000000</td>
</tr>
<tr>
<td>3</td>
<td>0.000095</td>
<td>0.000148</td>
<td>0.000182</td>
<td>0.000223</td>
</tr>
<tr>
<td>4</td>
<td>0.077965</td>
<td>0.090133</td>
<td>0.096540</td>
<td>0.103151</td>
</tr>
<tr>
<td>5</td>
<td>0.437370</td>
<td>0.449160</td>
<td>0.454382</td>
<td>0.459160</td>
</tr>
<tr>
<td>6</td>
<td>0.338497</td>
<td>0.325573</td>
<td>0.319114</td>
<td>0.312672</td>
</tr>
<tr>
<td>7</td>
<td>0.112521</td>
<td>0.104585</td>
<td>0.100833</td>
<td>0.097219</td>
</tr>
<tr>
<td>8</td>
<td>0.026887</td>
<td>0.024463</td>
<td>0.023341</td>
<td>0.022275</td>
</tr>
<tr>
<td>9</td>
<td>0.005472</td>
<td>0.004893</td>
<td>0.004629</td>
<td>0.004380</td>
</tr>
<tr>
<td>10</td>
<td>0.000997</td>
<td>0.000877</td>
<td>0.000823</td>
<td>0.000773</td>
</tr>
<tr>
<td>11</td>
<td>0.00166</td>
<td>0.00143</td>
<td>0.00133</td>
<td>0.00124</td>
</tr>
<tr>
<td>12</td>
<td>0.000025</td>
<td>0.000022</td>
<td>0.000020</td>
<td>0.000018</td>
</tr>
<tr>
<td>13</td>
<td>0.000004</td>
<td>0.000003</td>
<td>0.000003</td>
<td>0.000003</td>
</tr>
<tr>
<td>14</td>
<td>0.000000</td>
<td>0.000000</td>
<td>0.000000</td>
<td>0.000000</td>
</tr>
<tr>
<td></td>
<td>Var[LLPS]</td>
<td>0.8583</td>
<td>0.8583</td>
<td>0.8583</td>
</tr>
</tbody>
</table>

**Figure 5.1:** Theoretical probability distribution of LLPS for 256 balls into 128 bins, from Table 5.1.
allocation of balls into bins, so the null hypothesis can be stated explicitly,

\[ H_0 : \mu_0 = E[\text{LLPS}] = 6.493 \quad (5.1) \]

and the alternative hypothesis,

\[ H_1 : \mu_0 = E[\text{LLPS}] > 6.493. \quad (5.2) \]

Note that the alternative hypothesis does not include the possibility that \( \mu_0 < E[\text{LLPS}] \) as the random allocation achieves the minimum \( E[\text{LLPS}] \). An observation \( \bar{X} < E[\text{LLPS}] \) is unremarkable in the sense that it would not imply that the hash distribution was non-random. The critical region of the test, being the set of outcomes where the null hypothesis is rejected, is

\[ \frac{\bar{X} - \mu_0}{\sigma/\sqrt{T}} \geq z_\alpha. \quad (5.3) \]

The standard deviation of the theoretical LLPS distribution \( \sigma = \sqrt{\text{Var}[\text{LLPS}]} = \sqrt{0.8583} = 0.92645 \). For a significance level of \( \alpha = 0.05 \), to obtain 95% confidence, \( z_{0.05} = 1.64485 \). The critical region calculated for several sample sizes is shown in Table 5.2. The \( p \)-value for the test is

\[ p(\bar{X}) = \Phi \left( \frac{\bar{X} - \mu_0}{\sigma/\sqrt{T}} \right), \quad (5.4) \]

where \( \Phi \) is the standard normal cumulative distribution.

The sensitivity of the test is not affected by the bit-selection operation used to obtain the selected hash from a much larger hash value. Recall that the significance level of the test is the probability of making a type I error: deciding that \( H_0 \) is false when it is really true. When \( H_0 \) is true, the hash values of the
Critical region

<table>
<thead>
<tr>
<th>$T$</th>
<th>$(\bar{X} \geq)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>6.645</td>
</tr>
<tr>
<td>300</td>
<td>6.581</td>
</tr>
<tr>
<td>1000</td>
<td>6.541</td>
</tr>
<tr>
<td>3000</td>
<td>6.520</td>
</tr>
<tr>
<td>10000</td>
<td>6.508</td>
</tr>
</tbody>
</table>

**Table 5.2:** Critical regions for $\bar{X}$, the mean observed LLPS, for 95% confidence with various numbers of trials, $T$, in the black-box collision test with 256 hashes into 128 bins.

hash function have uniform probability, therefore the selected hashes from any selection of 7 bits will also occur with uniform probability.

To illustrate the sensitivity of the test, consider whether a function which is unbalanced to varying degrees would be detected when hashing random keys. If a hash function with a range of 128 hash values was unbalanced such that two of its hash values never occurred, but the keys were uniformly distributed over the remaining hash values, it would have the same theoretical LLPS as the 126-bin function. A function with 126 bins has $E[LLPS] = 6.5427$ (Table 5.1), which is within the critical region when $T = 1000$ trials are used, and the test is able to detect the unbalanced hash function with greater than 95% confidence. To detect a function which is unbalanced such that 127 of the 128 hash values are used, which has $E[LLPS] = 6.5175$, $T = 10000$ trials are needed.

What is an appropriate sensitivity for the test? The previous example, of an unbalanced function having the keys from a few hash values evenly redistributed among the others, can be thought of as a best case, or least-harmful, form of unbalance. The hash function still behaves as a balanced function, only with a few less bins, so that the increase in expected LLPS is small, and the resulting impact on system performance is limited. Another form of unbalance, where a few keys from each hash value are redistributed to a small number of high-frequency hash
values, has more damaging effects. Long chains accumulate at the high-frequency hash values, and the distribution of hash values becomes increasingly unbalanced as more hashes are added to the table. We elected to set the test sensitivity to detect a moderate degree of the best-case type of unbalance, specifically the test should detect a function that does not use two hash values, but is balanced over the remaining 126. Functions that are less unbalanced have minimal impact on system performance, while we ensure the test will detect the more damaging cases. Hence, we use 1000 trials in the test, but note that the sensitivity can be increased by performing more trials.

It is useful to relate this intuitive indicator of unbalance to a formal measure, such as that of Bellare and Rogaway (1993), which we use in the later experiments. The balance $B_h$ of hash function $h : U \rightarrow V$ is

$$B_h = \log_M \frac{D^2}{d_1^2 + d_2^2 + \ldots + d_M^2},$$

where $M$ is the size of the range of $h$, $d_i$ is the number of keys in the pre-image $h^{-1}(i)$ of hash value $i$, and $D$ is the size of the domain of $h$. Note that $\sum_{i \in V} d_i = D$. As described in Section 2.2.1, balance is the normalised entropy of the distribution of hash pre-image sizes. A balanced function has a balance measure equal to 1.0, and a function hashing all keys to a single value has balance equal to 0.0. The relationship between balance and the example functions with unused hash values (and balanced over the remaining hash values) is shown in Figure 5.2. The numerical value of balance declines slowly as the number of unused hash values increases. Functions that are nearly balanced will have balance values very close to 1.0.
Figure 5.2: Relationship between the balance measure and the number of unused hash values of an unbalanced hash function having a range of 128 and which is balanced over the remaining hash values.

**Statistical power** The statistical power of the test is the probability of correctly accepting the alternative hypothesis, $H_1$. When the hash function is unbalanced with expected LLPS $\mu$, and $\{\bar{X} \geq C\}$ is the critical region where the null hypothesis is rejected, the power function $K(\mu)$ gives this probability,

$$K(\mu) = \Pr[\bar{X} \geq C; \mu]$$

$$\approx 1 - \Phi \left( \frac{C - \mu}{\sigma/\sqrt{T}} \right), \quad \mu \geq \mu_0. \quad (5.6)$$

The approximation error is insignificant when $T > 30$ due to the central limit theorem. The power function is plotted in Figure 5.3 for several numbers of trials. When $T = 1000$ trials, the statistical power approaches 1.0 when $\mu > 6.6$. Specifically, at $\mu = 6.594$, the expected LLPS of an unbalanced hash function allocating random keys to only 124 of the 128 hash values, $K(\mu = 6.594) = 0.97$. By definition, when the expected LLPS $\mu = C = 6.54$, which is approximately
the expected LLPS from a hash function using only 126 of its 128 hash values, the statistical power is 0.5.

The bit-selection operation, obtaining a 7-bit hash from the original hash value, affects the statistical power of the test, but we are unable state conclusively whether statistical power is enhanced or diminished. For our test, statistical power is the probability of correctly identifying a non-pseudo-random hash function. Numerous such hash functions exist, each leading to a different distribution of the selected hashes, and we have not attempted to find a general solution solving all of the cases.

Nonetheless, we have analysed a single contrived example, and found that the statistical power is increased, by a minute amount, by the bit-selection step. The function is constructed so that the statistical power depends on several $E[LLPS]$ results for balanced functions which were calculated using the method of Reviriego et al. (2011). The function is extremely unbalanced, but nonetheless illustrates some aspects of the problem supporting our tentative hypothesis that the bit-
selection process does not greatly diminish the statistical power of the test.

**Example of statistical power with an unbalanced hash function**  Consider a hash function that does not use one half of its 128 hash values, and uniformly distributes hashes over the remaining 64. It does this in a manner such that the first bit of the hash value is always equal to 0. When we fill a hash table with the hashes of 256 random keys, without using the bit-selection operation, the theoretical LLPS distribution into 128 bins has \( \mu = E[LLPS] = 6.4926 \) and that \( \sigma^2 = Var[LLPS] = 0.8583 \) (from Table 5.1). However, our unbalanced hash function is effectively a balanced function with 64 hash values, and we calculate that its distribution has \( \mu = E[LLPS] = 9.3703 \). The function is very unbalanced, so we test using \( T = 100 \) trials, and calculate the critical region with confidence level of \( \alpha = 0.05 \) as \( (C - 9.3703)/\sqrt{0.8583/100} \geq 1.6449 \), so that \( C \geq 9.6450 \). Then the power of this test for the unbalanced function is \( K(\mu = 9.3703) = 1 - \Phi((C - 9.3703)/\sqrt{0.8583/100}) = 1 - \Phi(-29.4169) \approx 1 - 10^{-185} \).

Now we introduce the bit-selection operation. Six of the seven bits of the hash are selected, and we again perform 100 trials of populating a hash table with hashes of 256 random keys. For 256 balls into 64 bins, the theoretical \( \mu'_0 = E[LLPS] = 9.3703 \), and the variance is \( \sigma^2 = 1.4393 \). The critical region with confidence level of \( \alpha = 0.05 \) is \( (C' - 9.3703)/\sqrt{1.4393/100} \geq 1.6449 \), so that \( C' \geq 9.5676 \). There are seven possible bit combinations. Six of them include the first bit which is never equal to 1, so these selections will only fill 32 of the possible 64 hash values in the 6-bit hash function, which has \( E[LLPS] = 14.3840 \). The last combination does not contain the first bit, and fills all 64 of its hash values uniformly, with \( E[LLPS] = 9.3703 \). Assuming the bit combinations are selected randomly, the overall expected mean LLPS of the unbalanced function from the 6-bit tests is the weighted average of these two cases, \( \mu' = (6 \times 14.3840 + 1 \times 9.3703) \).
\[ 1 \times 9.3703/7 = 13.6678 \]. The statistical power is 
\[ K(\mu' = 13.6678) = 1 - \Phi((C' - 13.6678)/\sqrt{1.4393/100}) = 1 - \Phi(-34.1766) \approx 1 - 10^{-253}. \]

This function is extremely unbalanced, so the statistical power is very close to 1. Nonetheless, the statistical power is higher when using the bit-selection operation. An important effect demonstrated by the example is that different bit selections from an unbalanced hash function lead to different values for the expected LLPS of the selected hashes. Consequently, the variance of the observed LLPS values is greater than the variance of the LLPS of the original hash values. As the LLPS distribution is skewed toward larger values, an increased variance implies a larger observed mean, and an increased statistical power. However, the theoretical mean LLPS of balls into the reduced number of bins is also higher, leading to a larger value for the critical region and a reduction in statistical power. In general, we are unable to determine which of these opposing effects is greater. However, we expect that neither effect will dominate, and that the statistical power is not substantially diminished when bit selection is used. Later in this chapter, we test this empirically.

**Statement of the test procedure**  The specific steps of the test procedure we choose are,

1. Obtain 1000 sets of 256 distinct keys. The keys should reflect the distribution expected in the application.

2. For each key set, make a random selection of 7 bit positions in the hash.

3. For each key set, hash the keys and convert the hashes into 7-bit hashes, known as a *selected hash*, using the selected positions from step 2. Insert the selected hashes into a hash table of size 128, and measure the LLPS.

4. Find \( \bar{X} \), the mean of the 1000 LLPS measurements.
5. If $\bar{X} < 6.541$ accept the null hypothesis and conclude, with 95% confidence, that there is no evidence that the hash function does not produce a random allocation of hashes, or if $\bar{X} \geq 6.541$, do not accept the null hypothesis, and conclude that the hash function does not produce a random allocation of hashes.

The random selection of bits in step 2 must be different for each key set to ensure a good coverage of the whole hash function. In hash applications, it is a common practice to truncate a large hash to obtain a shorter hash from the least significant bits. This works well when the hash function is unbiased, but is not sound when it is not known that the hash function behaves well. A random selection of bit positions can be efficiently obtained using Floyd’s algorithm (Bentley and Floyd, 1987) for random sampling from a set without replacement.

5.2 Validation of the Collision Test

We performed an empirical validation of the hash function test to demonstrate that it is able to identify both pseudo-random and non-pseudo-random functions. These experiments also show that the bit-selection step does not significantly diminish the power of the test. To do this, we constructed two families of hash functions having a parameterized degree of unbalance and applied the test procedure to them using randomly generated keys. The use of unbiased keys is necessary in these tests to ensure that the distribution of hashes is entirely due to the degree of unbalance of the hash function, and is independent of other function properties, such as avalanche and resilience. Although this situation may seem contrived, and hashing of random keys is an uninteresting application, the
5.2. Validation of the Collision Test

Purpose is not to demonstrate a useful hash function, but to validate the test procedure.

Generating unbalanced hashes The unbalanced functions, which we refer to as the derived functions, were created from a balanced hash function with a larger range by partitioning the balanced hashes in an uneven manner. Let the balanced function have a range of \( W \) hash values, which we refer to as bins, and the derived functions have \( V < W \) hash values, which are called buckets. When the derived function is balanced, each bucket contains \( W/V \) bins.

The derived functions were implemented as an array indexed by the bucket number with each element containing the least bin in the range corresponding to that bucket. The bucket containing a given bin was found by linear search through the array. In the experiments, the unbalanced functions were all less than 1 million buckets in size, so the array occupied a fraction of system memory. The CRC32 checksum function, which generates a 32-bit hash, served as the balanced hash function. CRC32 is known to be balanced as it involves a simple division in finite field \( F_2 \) by a primitive polynomial. The implementation was from the zlib module in the Python 3.3.0 distribution.

Two families of derived functions were implemented. The first, named HiLo, was created by starting with a balanced function: each bucket contained \( W/V \) bins. Then the buckets were grouped into pairs. A number of bins were moved from one bucket in the pair to the other bucket. The resulting function has two groups of buckets: the ‘low buckets’ having fewer bins than a balanced function, and the ‘high buckets’ having more bins by the same amount. The distribution of bucket sizes is shown in Figure 5.4 part (i). The degree of unbalance was determined by the parameter \( \beta \), \( 0 \leq \beta \leq 1 \), the fraction of bins moved between buckets. Specifically, the number of bins in a low bucket was

\[
    w_{\text{low}} = \lceil (1 - \beta)W/V \rceil,
\]

and...
the number of bins in a high bucket was \( w_{\text{high}} = \lceil (1 + \beta)W/V \rceil \). A value of \( \beta = 0.0 \) resulted in a balanced function as no bins were moved, and the maximum value of \( \beta = 1.0 \) resulted in all of the bins being moved, so that one half of the hash values would never occur.

The second derived hash function family, named Redistribute, was created from a balanced function by removing all of the bins from a few buckets and redistributing them uniformly to the remaining buckets. The controlling parameter, \( \gamma, 0 \leq \gamma \leq V - 1 \), specified the number of buckets to redistribute, so that \( \lceil \gamma W/V \rceil \) bins are removed from \( \lceil \gamma \rceil \) buckets and distributed over the remaining \( V - \lceil \gamma \rceil \) buckets. For example, if a balanced bucket contains 256 bins, \( \gamma = 4.5 \) specifies a function with four empty buckets, one half-empty bucket with 128 bins, and the remainder having 266 bins in 45 buckets and 265 bins in 78 buckets. A value of \( \gamma = 0.0 \) specifies a balanced function, and \( \gamma = V - 1 \) obtains a function where all keys hash to the same value. The distribution of bucket sizes is shown in Figure 5.4 part (ii).

The random keys were strings of length 32 over an alphabet of size 256: that is, they were arrays of 32 random bytes. The bytes were generated using the Python random library. In each individual trial, duplicate keys were discarded.

## 5.3 Results Without Bit-Selection

We first verify that the test reliably detects a biased distribution of hash values, while omitting the potentially complicating bit-selection step. This was done using derived hash functions that generated 7-bit hashes. Consequently, all 7 bits of the hash value were needed to create the selected hash, and step 2 of the test procedure, for making a random selection of hash bits, is not needed.

The results of the experiment are shown in Table 5.3 and Figure 5.5. Both
families of derived functions, HiLo and Redistributed, were used with a range of parameter values to obtain various degrees of unbalance. Each data point is the mean observed LLPS from 1 000 trials of filling a hash table of size 128 with the hashes of 256 random keys as described in Section 5.2, using a fixed balance parameter.

In Figure 5.5, the mean observed LLPS is plotted against decreasing function balance. The results from each derived function family show a broadly linear relation between LLPS and unbalance. There is some variability around the overall trend, reflecting the randomness involved in the experiment.

Interestingly, the results for each hash function family follow trend lines with different gradients. This suggests that the relationship between expected LLPS for unbalanced functions and the balance measure is complex. The difference between the two unbalanced function families is in the distribution of the unbalanced bins: in HiLo, the unbalanced bins are spread evenly over half of the buck-
Table 5.3: Results for the test without bit-selection. The mean observed LLPS and p-values from 1 000 trials of 256 random 32-byte strings hashed by the derived functions, HiLo and Redistributed, with a range of parameter values. A balance value of 1.0 indicates a balanced function, and the functions become more unbalanced as the balance value decreases. LLPS observations in the 95% critical region of $X > 6.541$ are shown in bold type.
Figure 5.5: Results for the test without bit-selection, listed in Table 5.3. The mean LLPS observed from 1000 trials of 256 random 32-byte strings hashed by unbalanced functions HiLo and Redistributed having 7-bit hashes. Balance equals 1.0 for a balanced function and decreasing balance to the right of the x-axis indicates a more unbalanced function. The critical region for the test procedure, mean observed LLPS $\bar{X} > 6.541$, and the expected LLPS, $E[LLPS] = 6.4926$, for a random allocation of 256 balls into 128 bins are shown as dashed lines.
ets, so the range of bucket sizes is more compact compared to the \textit{Redistributed} family where some of the buckets are empty. The relationship of LLP S to the distribution of pre-image sizes of an unbalanced function appears to depend on skewness or higher moments. Consequently, we urge caution when using the balance measure. It is a convenient indicator of the distance from balance, but care is needed when using it for comparison between unbalanced functions having different distributions.

The experiment obtained a sensitivity matching the intended sensitivity of the test procedure. The test is designed to detect a function assigning hashes to only 126 of its 128 hash values, which is the case when the \textit{Redistributed} function has $\gamma = 2$; and we see in Table 5.3 that is the first data point that the test procedure detected as not being a balanced function. The HiLo function was also detected as being unbalanced at a similar balance value, although, as noted earlier, we should not make a direct comparison between the functions based on the balance measure. We conclude that the test is able to accurately discriminate between pseudo-random and non-pseudo-random functions with the intended sensitivity.

\section*{5.4 Results With Bit-Selection}

Having seen in the previous experiment that the test is able to detect a biased distribution of hash values, we now investigate the effect of the bit-selection procedure when testing hash functions having a larger range. An anticipated criticism of the test procedure is that a hash table of size 128 is too small, and seemingly provides little insight into the behaviour of hash tables of many millions of hashes. However, this experiment shows that the test procedure is valid, and is in fact preferable to a test which fills a single large hash table as more data points are
obtained.

The results are shown in Table 5.4 and Figure 5.6. The \textbf{Redistributed} family of unbalanced hash functions was used with a hash length of 17 bits, obtaining a range of 131,072 hashes (the \textit{HiLo} family was not used in this experiment). From these hashes, \textit{selected hashes} were obtained, similarly to the test procedure, but with a range of lengths from 7 to 17 bits. Consequently, the selected hashes were inserted into hash tables having a range of sizes, from 128 values for the 7-bit selected hashes, to 131,072 values for the 17-bit selected hashes. For each test, the same number, $1024 \times 256$, of random 32-byte keys were hashed so that the tests involved comparable computational effort. The number of hashes inserted into each table was twice the size of the table, so that the LLPS distributions were broadly comparable between the tests. As noted earlier, when the number of keys is a fraction of the hash table size, there is a greater likelihood that the distribution of hash values will appear to be random. Consequently, the number of trials performed in each test was progressively smaller for larger selected hashes. Specifically, for the 7-bit selected hashes, there were 1024 trials in which 256 hashes were loaded into a hash table of size 128; for 9 selected bits, there were 256 trials with 1024 hashes loaded into a table of size 512; and so on. Finally, in the test using 17 selected bits, all 262,144 hashes were inserted into a single hash table of size 131,072. The numbers of trials for the selected bit lengths are shown in the bottom line of Table 5.4. Consequently, the experiment investigates the trade-off between testing longer hashes, which require a larger hash table and fewer trials, compared to shorter hashes which allow smaller hash tables and more trials. For each trial, a new random selection of the hash bits was made.

The tests with 7 selection bits which were detected as being unbalanced are shown in bold in Table 5.4. Note that the test procedure confidence was calculated for 1000 trials, whereas these results performed 1024 (to provide convenient
Chapter 5. Hash Function Testing

<table>
<thead>
<tr>
<th>( \gamma )</th>
<th>balance</th>
<th>7</th>
<th>9</th>
<th>11</th>
<th>13</th>
<th>15</th>
<th>17</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.0</td>
<td>1.000</td>
<td>6.488</td>
<td>7.485</td>
<td>8.430</td>
<td>9.338</td>
<td>10.075</td>
<td>11.1</td>
</tr>
<tr>
<td>1024.0</td>
<td>0.9993</td>
<td>6.528</td>
<td>7.438</td>
<td>8.438</td>
<td>9.313</td>
<td>10.00</td>
<td>11</td>
</tr>
<tr>
<td>2048.0</td>
<td>0.9987</td>
<td>6.525</td>
<td>7.523</td>
<td>8.547</td>
<td>9.688</td>
<td>10.25</td>
<td>14</td>
</tr>
<tr>
<td>4096.0</td>
<td>0.9973</td>
<td>6.485</td>
<td>7.582</td>
<td>8.265</td>
<td>9.500</td>
<td>10.50</td>
<td>10</td>
</tr>
<tr>
<td>8192.0</td>
<td>0.9945</td>
<td>6.551</td>
<td>7.730</td>
<td>8.516</td>
<td>9.375</td>
<td>10.25</td>
<td>12</td>
</tr>
<tr>
<td>16384.0</td>
<td>0.9887</td>
<td>6.705</td>
<td>7.781</td>
<td>8.750</td>
<td>9.813</td>
<td>10.75</td>
<td>11</td>
</tr>
<tr>
<td>24576.0</td>
<td>0.9824</td>
<td>6.833</td>
<td>8.000</td>
<td>9.063</td>
<td>10.063</td>
<td>11.00</td>
<td>12</td>
</tr>
<tr>
<td>32768.0</td>
<td>0.9756</td>
<td>7.023</td>
<td>8.191</td>
<td>9.453</td>
<td>10.625</td>
<td>11.00</td>
<td>14</td>
</tr>
<tr>
<td>40960.0</td>
<td>0.9682</td>
<td>7.093</td>
<td>8.410</td>
<td>9.609</td>
<td>11.063</td>
<td>12.00</td>
<td>12</td>
</tr>
<tr>
<td>49152.0</td>
<td>0.9601</td>
<td>7.308</td>
<td>8.828</td>
<td>10.328</td>
<td>11.563</td>
<td>12.50</td>
<td>14</td>
</tr>
<tr>
<td>57344.0</td>
<td>0.9512</td>
<td>7.475</td>
<td>9.207</td>
<td>10.328</td>
<td>11.688</td>
<td>13.25</td>
<td>14</td>
</tr>
<tr>
<td>65536.0</td>
<td>0.9412</td>
<td>7.653</td>
<td>9.285</td>
<td>10.578</td>
<td>10.313</td>
<td>14.50</td>
<td>15</td>
</tr>
</tbody>
</table>

Table 5.4: Results for the test using bit-selection. A redistributed unbalanced function generated 17 bit hashes from which selected hashes with fewer bits were created. In each test, 2,621,440 keys were hashed and inserted into hash tables. In each trial, the number of hashes inserted into the hash table was twice the hash table size, so fewer trials were run in tests having more selected bits, as listed on the bottom line. The number of keys used for the tests with a balanced function (\( \gamma = 0.0 \)) was 2,621,440 to obtain a more accurate estimate of the theoretical \( E[LLPS] \). LLPS observations using 7 selected bits that were greater than the rejection threshold of 6.541 are shown in bold type.
Figure 5.6: Results for the test using bit-selection experiment shown in Table 5.4. 17-bit hashes were generated by derived functions from the Redistributed family with a range of unbalance. These hashes were tested for a random distribution using several numbers of selected bits. Each curve shows the mean LLPS observations using a given number of selected bits over a range of function unbalance. Balance equals 1.0 for a balanced function and decreasing balance indicates a more unbalanced function.
division by 2 for number of trials with more selected bits), so the confidence is slightly greater than 95%. However, the theoretical $E[\text{LLPS}]$ is not known for the larger selected hashes, and a critical region cannot be determined. Instead, an estimate of $E[\text{LLPS}]$ was made experimentally by performing 10 times as many trials using a balanced function (when $\gamma = 0.0$). Consequently, evaluation of the longer selected hashes can only be by comparison to the mean LLPS when $\gamma = 0.0$, and we do not know the variance or an estimate of the 95% confidence limit.

Figure 5.6 shows that the mean LLPS for each selected hash size increases fairly linearly with the degree of unbalance, in a manner similar to the results from the previous experiment without the bit-selection step. With 7 selected bits, the LLPS exceeds the critical value at a similar balance to the first experiment, indicating that there was no reduction in LLPS caused by the bit-selection step. The curves are separated vertically as the $E[\text{LLPS}]$ value is higher for the larger hash tables. The deviations about the linear trend are greater for the larger selected hash sizes as the number of trials is smaller, resulting in a larger variance for the mean observed LLPS. Note that the runs with selected hash length equal to 17 bits only have a single measurement of LLPS, so the mean LLPS must be a whole number.

Figure 5.7 compares the results from the previous experiment with 7-bit hashes and no bit-selection with the results from this experiment with 17-bit hashes and the selection of 7-bits. The broad relationship between the mean observed LLPS and the degree of unbalance is similar for both data sets. Considering the noise present in the data, as expected from the random nature of the test, the chart gives no support for the hypothesis that the bit-selection process leads to a reduced LLPS from unbalanced functions, and in fact suggests that the LLPS is higher. Consequently, we tentatively conclude that the statistical power of the
test is unaffected by the bit-selection step.

Overall, the test using 7 selected bits is able to detect an unbalanced function at least as well as a test that uses more selected bits, while performing the same number of hash calculations. The sensitivity with 7 selected bits is similar to the results obtained in the previous experiment, and appears to be unaffected by the inclusion of the bit-selection step. Even if the $E[LLPS]$ were known for the larger hash tables with more selected bits, it seems likely that these tests would not be more accurate as fewer trials are performed resulting in a wider variance in the mean LLPS measurement.

## 5.5 Discussion

The black-box collision test procedure is able to correctly identify a sufficiently unbalanced hash function. It provides a significance measure of hash function performance compared to an ideal random allocation of balls into bins; an absolute measure of randomness rather than a relative comparison to other hash functions. If more precision is required, then it can easily be improved by increasing the number of trials.

While the analysis and empirical evaluations presented in this chapter have all involved hashing random keys with unbalanced hash functions, the test is intended for evaluating functions in other situations. The results from hashing random keys are well known, depending only on the function balance, and are independent of other properties such as avalanche and resilience. As such, random keys provide a convenient setting for demonstrating the effectiveness of the test. However, the test is also applicable in the practical circumstance of biased keys. The test can be used to evaluate a hash function for an application by using keys that are representative of the future data. By performing multiple small trials,
Figure 5.7: Comparison of testing with and without bit selection. The mean observed LLPS using the Redistributed function family with 7-bit hashes without bit-selection from Table 5.4, and with 17-bit hashes tested with 7 selected bits from Table 5.4. Expected LLPS is the theoretical LLPS for a uniform allocation of 256 balls into 128 bins, and the critical region is the 95% confidence limit.
small sets of keys representing specific situations can be included in the test, such as changes to single-characters, or common prefixes. Additionally, other function properties can be tested by employing a specific set of keys, such as in the next chapter where we test function resilience by hashing key sets created from permuted alphabets.

One aspect of this investigation deserving further work is to perform more detailed empirical studies. There are several parameters that could be varied, such as the hash table size and the number of keys which are hashed. The values we chose work well, but there may be more effective settings. The impact of the bit-selection step is another aspect requiring detailed experimental study using larger hashes, a greater range of functions, and more observations close to the critical region. While our experiments suggest that the power of the test is not decreased by the bit-selection process, a wider range of results are needed to confirm this tentative conclusion.
Chapter 6

Hashing Biased Data

The hash function for generating fingerprints is an essential element of a CBC system. The higher rate of hash collisions generated by a poor hash function makes the CBC system worse in various ways, depending on the algorithm used for handling collisions in the fingerprint index. If collisions are discarded, then the index will contain fewer \( n \)-gram references and the likelihood of identifying a matching string will be reduced, leading to a decrease in compression achieved. Alternatively, if hash collisions are recorded in the index, then the index query time will increase as more \( n \)-gram references must be retrieved from external storage for collision resolution.

How can we know that a hash function is suitable for CBC? In Chapters 3 and 4, our CBC system, \texttt{cobald}, used the rolling hash function by irreducible polynomial division of Cohen (1997). We noted that there is some uncertainty concerning the suitability of this function after Lemire and Kaser (2010) proved that rolling hash functions can have no more than pairwise independence. We addressed this uncertainty by proving that the independence is not necessarily limited when the function keys do not overlap, such as for the fixed-offset \( n \)-gram selection method of \texttt{cobald}. In this chapter, we seek further reassurance on this
question, and investigate the properties which are necessary for a hash function to be well behaved.

In Chapter 2, we reviewed the desirable properties of non-universal, non-cryptographic hash functions. As described in Section 2.2, current practice for evaluating hash functions is to test for one, or sometimes two, bits of avalanche, and to perform collision tests with a range of artificial and representative sets of keys. We also noted there is an inverse relation between function resilience and avalanche, described by Zheng and Zhang (2003), which bounds the avalanche of a function having a given resilience. They showed that a function on keys of length \( n \) with resilience degree \( t \) and avalanche degree \( l \) satisfies the bound \( t + l \leq n - 2 \). This relationship allows us to use function resilience as an indirect means to understand the role of avalanche.

In this chapter, we investigate the conditions under which function resilience is beneficial for a hash function. Specifically, how can a set of input keys be biased while still obtaining a uniform distribution of hash values with a resilient function? We prove that the hash values from a resilient function are balanced for pairs of keys having a limited degree of similarity. Then, in the particular case of zero-order biased keys, we show when a resilient function is effectively pairwise independent. These results lead to an empirical test for function resilience, and we present an experimental study in which we hash zero-order biased data with resilient functions to validate the test and confirm the theoretical results. Finally, we consider the importance of function resilience in practical situations by hashing some English text, and by testing the resilience of several popular hash functions.
6.1 Resilient Hash Functions on Biased Data

In order to study the effect of biased keys on the distribution of hash values, we introduce a probability distribution for the keys. Let \( h : U \rightarrow V \) be a hash function where \( U \) and \( V \) are finite sets, and let \( e : U \rightarrow [0, 1] \) be the probability of occurrence of each key, \( u \in U \). The probability of a hash value \( v \in V \) occurring is then the sum of the probabilities of the keys in its pre-image, \( p_v = \sum_{u \in h^{-1}(v)} e(u) \).

The expected uniformity of the hash values from function \( h \) and bias distribution \( e \) can then be measured by the hash distribution entropy, \( H_{h,e} = -\sum_{v \in V} p_v \log_2 p_v \) bits. The value of \( H_{h,e} \) ranges from \( \log_2 |V| \) bits, for a uniform distribution of hash values, to 0 bits for a hash function that always gives the same hash value.

For our purpose of inserting a finite set of keys into a hash table, we consider functions having \( H_{h,e} \) close to \( \log_2 |V| \) bits to be effectively uniform when the number of hash collisions is statistically indistinguishable from the ideal hash function. This is similar to the approach to hash testing adopted in Chapter 5, where a function was accepted as being ideal if it behaved as an ideal hash function within the statistical significance of the test. For a given biased input distribution \( e \), there will be many hash functions that are effectively uniform. While an exactly uniform function may not exist, there will be an optimal hash function for which \( H_{h,e} \) is maximised and the expected distribution of hash values is closest to uniform. However, the optimal hash function for particular biased input distribution, \( e \), is of little practical interest as the exact distribution, \( e \), is not known precisely in practical situations, and this function will not be optimal for input data that is biased in a slightly different way. Instead, we desire a hash function that is effectively uniform, while not necessarily maximal, over an entire class of biased input data. In this section, we present an advance towards this goal by proving that a \( t \)-resilient function is effectively pairwise independent.
when hashing keys which are biased in a certain way: specifically, when few pairs
of keys, $x, y \in \Sigma^n$, have the same symbol, $x_i = y_i, 1 \leq i \leq n$, at more than $t$
positions.

Resilient Boolean functions were defined by Chor et al. (1985) and generalised
to functions on a finite field by Camion and Canteaut (1999).

**Definition 8** ($t$-resilient function). Let $h$ be a hash function $h : \Sigma^n \to V$, where
the input keys are strings of $n$ symbols from alphabet $\Sigma$ of cardinality $q$, with
range $V$. The key $x$ is a string of $n$ symbols $x_1 x_2 \ldots x_n$. $V$ can be defined as the
set of integers $\{0..|V| - 1\}$. A function is balanced when equal numbers of keys
map to each hash value, $v \in V$. Function $h$ is $t$-resilient if it is balanced and
has the property that, when every set of $t$ positions of the input are arbitrarily
fixed, $h$ is balanced over the restricted domain of the remaining $n - t$ un-fixed, or
free, positions. Specifically, $h$ is balanced over each set of keys $L$, referred to as
the restricted key set, having $t$ positions $\{i_1, \ldots, i_t\} \subseteq \{1, 2, \ldots, n\}$ that are fixed
to $x_i = s, \forall s \in \Sigma$. Furthermore, $t$ is the maximum resilience of $h$ that is not
balanced if $t + 1$ positions are fixed.

To illustrate, consider a 2-resilient function on strings of length four over
the English upper case letters, $\Sigma = \{A, B, \ldots, Z\}$. The ‘*’ symbol indicates an
unfixed, or free, position that can take an arbitrary value from $\Sigma$. Fixing one or
more of the positions defines a restricted key set $L \subset \Sigma^n$. For example, when
the first two characters of the input string are fixed to $AB$, the restricted key set
is written $\{AB**\}$, and contain $\{ABAA, ABAB, \ldots, ABAZ, ABBA, ABBB, \ldots, ABZZ\}$.
The resilience condition implies that the hashes of the keys in the restricted key
set, $h(x, x \in \{AB**\})$, are balanced.

The 2-resilient function is also balanced over the other key sets restricted to
two positions, such as $\{PQ**\}, \{AB*\}$ and $\{Z*A\}$. Also, a restriction of a single
position, such as \{A***\}, is also balanced in a 2-resilient function. However a restriction at three positions \{ABC*\} will not be balanced as that would imply that the function is 3-resilient.

The following property of \(t\)-resilient functions is useful in the proof and later examples.

**Property 9 (Balanced \(t\)-grams).** A \(t\)-gram is a string in \(\Sigma^t\). A \(t\)-resilient function, \(h : \Sigma^n \to V\), has an equal number of each \(t\)-gram in each of its pre-images.

**Proof.** By definition, a \(t\)-resilient function has the property that when values are assigned to a set of \(t\) input positions, the function is balanced over the remaining free inputs. Let \(q = |\Sigma|\) be the alphabet size. Fixing the input symbols at \(t\) positions partitions each pre-image into \(q^t\) subsets containing \(q^{n-t}/|V|\) keys. These subsets must have equal size for the restricted functions to be balanced. Therefore, there must be an equal number of each combination of \(t\) symbols at the positions of the \(t\) fixed inputs in each pre-image. As a \(t\)-resilient function is balanced for all combinations of \(t\) fixed input positions, including the adjacent positions which form \(t\)-grams, the number of \(t\)-grams in each pre-image is balanced.

We now define a measure of the degree of unbalance of a function, which is needed in the later proof.

**Definition 10 (Unbalance over a subdomain).** Let \(h^{-1}_L(v)\) be the pre-image of function \(h\) for hash value \(v \in V\) over a key set \(L\). When \(h\) is balanced over \(L\), the cardinality of each pre-image over \(L\), \(|h^{-1}_L(v)| = |L|/|V|, v \in V\). When function \(h\) is unbalanced over \(L\), there exists a pre-image \(|h^{-1}_L(v)| \neq |L|/|V|, v \in V\). The
degree of unbalance, $\delta_L$, is the sum

$$
\delta_L = \sum_{v \in V} \| h^{-1}_L(v) \| - \frac{|L|}{|V|}.
$$

(6.1)

The degree of unbalance, $\delta_L$, is zero when $h$ is balanced over $L$. When $\delta_L > 0$, it represents twice the least number of keys whose hashes could be changed in order to transform $h$ into a balanced function over $L$.

The first step in our proof regarding the effective pairwise independence of $t$-resilient functions is to define the concept of pairwise balanced keys.

**Definition 11** (Pairwise balanced keys). Let $x, y \in \Sigma^n$, $x \neq y$ be distinct input keys. Furthermore, the keys $x$ and $y$ are said to constrain each other as they have matching symbols, $x_i = y_i$, at $k < n$ of their positions. Let $L$ be the set of keys satisfying the constraint, by fixing the same matching symbols at the $k$ positions, and having an arbitrary symbol at the free, or non-fixed, positions. The keys $x$ and $y$ are said to be pairwise balanced with respect to the hash function $h$ if $h(z)$, $z \in L$ is balanced.

For example, $ABCZ$ and $DEFZ$ match in their last position. Consider pairs of keys which match at $k$ positions. For example, $x = ABXY$ and $y = ABPQ$ match at two positions, and $L = \{AB**\}$. The matching positions are the constraints imposed by $x$ on $y$ and by $y$ on $x$. These keys are pairwise balanced with respect to $h$ if the function $h(z)$, $z \in \{AB**\}$ is balanced.

Pairwise balance is a useful property when the bias of the keys relates to the constraint. Suppose that the bias is entirely characterised by the constraint, and the remaining unconstrained aspect of the pair of keys is unbiased, we can then consider that the keys $x$ and $y$ were randomly selected from the balanced set $L$, resulting in a uniform distribution of hash values. While other constraints could
be defined, here we are specifically concerned with keys that are biased such that key pairs are likely to have \( k \) matching positions, but are otherwise un-biased.

Pairwise balance has some similarities to the familiar concept of pairwise independence of universal hash functions. In a universal hash function family, the hash values associated with a pair of input keys are constrained by the additional input of a seed that selects the hash function from the universal family. Pairwise independence is obtained when the hashes of each key pair are balanced over all of the seeds, and the seed is chosen uniformly at random. Consequently, for all key pairs, the hash of the second key is uniformly distributed independent of the first key. With the non-universal function \( h \), consider partitioning the key into two parts: one that is randomly selected, and is analogous to the seed of a universal hash function, and the other remaining part, which corresponds to the biased key of a universal hash function. Pairwise balance of the randomly selected part ensures that the distribution of hash values is uniform, regardless of the degree of bias of the other part of the key.

We now prove a bound on pairwise balance on some subsets of the domain of a \( t \)-resilient function. When the fraction of key pairs in the subset that are not pairwise balanced approaches zero, the hash function is effectively pairwise balanced over that subset.

**Theorem 12.** Consider a pair of distinct keys \( x \) and \( y \), \( x \neq y \), which match at \( k \) positions, let function \( h \) be \( t \)-resilient, and let \( j, 0 \leq j < t \), be a lower bound on the number of matching positions found in the key set so that \( j \leq k \leq t \).

1. When \( k > t \), the pair \( x, y \) are not pairwise balanced; and

2. The fraction \( f_{j,t} \) of pairs \( x, y \) matching in \( k \) positions that are unbalanced is

\[
f_{j,t} \leq \frac{q^{n-j}}{\sum_{c=0}^{t-j} \binom{n-j}{n-j-c} (q-1)^{n-j-c}} - 1. \tag{6.2}
\]
208 6.1. Resilient Hash Functions on Biased Data

Proof. Let $x \in \Sigma^n$ be a key, and $y \in \Sigma^n$ be a key having $j$ matching positions with $x$. The restricted key set $L_j$ contains the keys having these $j$ positions fixed to the matching symbols in $x$ and $y$. Observe that $|L_j| = q^{n-j}$ and also note that $z \in L_j$ may match $x$ at more that $j$ positions. Indeed the key $z = x$ is an element of $L_j$. To illustrate, suppose $h$ has $n = 8$, $x = ABCDEFGH$ and $y = ABCDETVU$ so that the first $j = 5$ positions are fixed to $ABCDE$. The set of possible keys is $L_5 = \{ABCDE***\}$, and contains $z = ABCDEFYZ$ which has $k = 6$, as well as $z = x = ABCDEFGH$ which has $k = 8 = n$.

The first part of the theorem is a direct consequence of the resilience of $h$: when $k > t$, $h(z), z \in L_k$ is not balanced, so the key pair $x, y$ is not pairwise balanced. For the second part of the theorem, when $j \leq k \leq t$, we know that $h(z), z \in L_j$ is balanced, from the definition of resilience. While $L_j$ is balanced, it contains keys that have more than $t$ matching positions with $x$. As we want to find the balance of pairs with $j \leq k \leq t$ matching positions, the keys with more than $t$ matching positions must be subtracted from $L_j$. Furthermore, as the function $h$ is unbalanced when more than $t$ positions are fixed, the residual set formed by removing those keys from $L_j$ must also be unbalanced by an equal amount. We now calculate the fraction of keys in this residual set that are unbalanced by counting the sizes of the sets which are removed.

Let $N_j \subset L_j$ be the set of keys matching $x$ only at the $j$ given matching positions. As none of the $n - j$ free positions of $z \in N_j$ can match the free positions in $x$, $|N_j| = (q - 1)^{n-j}$. Furthermore, let $E_j$ contain the keys in $L_j$ having more than $j$ matching positions with $x$, so that $E_j = L_j \setminus N_j$, and $|E_j| = q^{n-j} - (q - 1)^{n-j}$. To illustrate the relationship between sets $L_j, N_j$ and $E_j$, let $x = ABCDEFGH$ and $y = ABCDETVU$ so that the first $j = 5$ positions are fixed to $ABCDE$. The restricted key set is $L_5 = \{ABCDE***\}$, and contains $ABCDEXYZ \in N_5$ and $ABCDEFYZ \in N_6$. The set $E_5 = \{ABCDEF**\} \cup \{ABCDE*G*\} \cup \{ABCDE**H\}$, and
Chapter 6. Hashing Biased Data

\[ E_6 = \{ ABCDEFG* \} \cup \{ ABCDEF*H \} \cup \{ ABCDE*GH \}. \]

\[ N_5 = L_5 \setminus E_5 \setminus \{ ABCDE** \} \cup \{ ABCDE*G* \} \cup \{ ABCDE**H \}. \]

Generally, the number of keys matching \( x \) at the given \( j \) positions, and at a further \( i \) of the free positions, is

\[ |N_{j+i}| = \binom{n-j}{n-j-i} (q-1)^{n-j-i}. \] (6.3)

Also, \( E_{j+i} \) contains the keys in \( L_j \) having more than \( j+i \) matching positions with \( x \), and \( N_{j+i} = E_j \setminus E_{j+1} \). The cardinality of \( E_{j+i} \) can now be found,

\[ E_{j+i} = L_j \setminus \bigcup_{c=0}^{i} N_{j+c}. \] (6.4)

As \( N_j, N_{j+i}, \ldots \) do not intersect,

\[ |E_{j+i}| = |L_j| - \sum_{c=0}^{i} |N_{j+c}| \]

\[ = q^{n-j} - \sum_{c=0}^{i} \binom{n-j}{n-j-c} (q-1)^{n-j-c}. \] (6.5)

We can see that \( E_{j+i} \) is the union of several restricted keys sets. When \( j+i < t \), each of the restricted key sets in \( E_{j+i} \) is balanced, but \( E_{j+i} \) itself is not balanced as some keys occur in more than one of the restricted key sets. Conversely, when \( j+i = t \), the restricted keys sets in \( E_t \) are not balanced. In the worst case, assume that all keys in \( E_t \) hash to the same value. As \( L_j \) is balanced, the degree of unbalance of \( L_j \setminus E_t \) is at most \( |E_t| \), and the fraction \( f_{j,t} \) of keys matching in
Resilience, $t$

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{resilient_hash_functions.png}
\caption{Selected solutions to Equation 6.2, the upper bound of the fraction of pairs which are unbalanced, $f_{j,t}$, for various values of $j$, the lower bound on the number of matching positions between pairs, and functions having a range of resilience values. The functions have alphabet size $q = 128$ and key length $n = 32$, which are the values used in the experiment in Section 6.4. As the range $t - j$ increases, the fraction of pairs which are not balanced approaches 0, and the function is effectively pairwise balanced over the subdomain where key pairs have $k, j \leq k \leq t$ matching positions.}
\end{figure}

For a representative set of resilient functions are plotted in Fig-

\begin{align*}
\frac{|E_t|}{|L_j - E_t|} & \leq \frac{q^{n-j} - \sum_{c=0}^{t-j} \binom{n-j}{n-j-c} (q - 1)^{n-j-c}}{\sum_{c=0}^{t-j} \binom{n-j}{n-j-c} (q - 1)^{n-j-c}} \\
& \leq \frac{q^{n-j}}{\sum_{c=0}^{t-j} \binom{n-j}{n-j-c} (q - 1)^{n-j-c}} - 1
\end{align*}

The fraction $f_{j,t}$ approaches zero as $t - j$ increases.
ure 6.1, and show that $f_{j,t}$ is less than 0.001 when the range $t - j > 2$. When $f_{j,t}$ approaches zero, the hash function $h$ is effectively pairwise balanced for all of the key pairs having $k$ matching positions within the range $j \leq k \leq t$. If the keys are biased such that key pairs outside of this range are unlikely to occur, then the hash function $h$ will be effectively pairwise balanced for the biased data. Furthermore, if the keys are otherwise unbiased, then the hash function will be effectively pairwise independent for the biased keys. The parameter $j$ is included as the lower bound of the number of matching positions, $k$, to more accurately model highly biased key distributions where the probability of $k < j$ is very low.

**Hashing zero-order biased keys** The above proof shows that a $t$-resilient function is effectively pairwise independent for keys biased in a certain way: when the number of matching positions between keys is highly likely to be less than $t$; and furthermore that the symbols in the non-matching positions are not biased. We now argue that keys generated by a zero-order source meet these conditions in certain circumstances. A zero-order source assigns a symbol to each position of a key according to a probability distribution over the alphabet, where $p_i$ is the probability that the $i$-th symbol in $\Sigma$ is assigned, independent of the symbols at other positions. The probability that a pair of keys have matching symbols at a position is, $\Pr[x_i = y_i] = p_m = \sum_{j=1}^{m} p_j^2$. For keys of length $n$, the probability of obtaining a pair containing $k$ matches follows a binomial distribution,

$$\Pr[k\text{ matching positions}] = \binom{n}{k} p_m^k (1 - p_m)^{n-k}, \quad (6.10)$$

and the probability of having at most $t$ matching positions is

$$\Pr[k \leq t] = \sum_{i=0}^{t} \binom{n}{i} p_m^i (1 - p_m)^{n-i}. \quad (6.11)$$
We know that when the expected value of $k$ is less than the resilience $t$, then the key pairs are effectively balanced, meaning that the hashes of the keys in the restricted key set formed by the $k$ matching positions is close to balance. However, this does not imply that the distribution of hash values is uniform, as the keys are selected by a zero-order biased process. Nonetheless, the following informal argument shows that, when $k$ is significantly less than $t$, we expect the hash distribution to be effectively uniform for zero-order biased data. When $k$ key positions in a $t$-resilient function are fixed, the function over the remaining $n - k$ positions is $t - k$ resilient. Consider the case when $t = k$. A highly biased hash distribution could be achieved when one of the pre-images contained keys composed of the most likely symbols, leading to a high probability of the occurrence of that hash value. Similarly, another pre-image could contain keys composed of the least likely symbols, and that hash value would be very unlikely to occur. Now consider the case when $t - k = 1$. Due to Property 9 (Page 205), the 1-resilient function has balanced 1-grams, meaning that every symbol occurs in each pre-image an equal number of times. So the above situation for the $t - k = 0$ function cannot occur. Instead, a pre-image can only contain a few keys with a large number of the most likely symbol, and the other likely keys must be distributed among other pre-images. The situation is further improved when $t - k = 2$, and the distribution of hash values approached uniformity as $t - k$ increases. This phenomenon is demonstrated with a case study in Section 6.2. Consequently, when the expected value of $k$ is significantly less than $t$, a $t$-resilient function is effectively pairwise independent when hashing zero-order biased data.

**Discussion** The motivation for this proof is to identify when resilience is a useful property for a non-universal non-cryptographic hash function. Specifically, we showed when the resilience exceeds the number of matching positions between
pairs of keys, the function is balanced over the remaining non-matching positions, which results in an uniform distribution of hash values when the keys were generated by a zero-order biased source. While a zero-order source is an idealised simplification that is not encountered in practical situations, there is a zero-order component in the bias of keys from many common applications; the letter frequencies in written languages are an important instance. We expect that, when zero-order bias is a major component of the overall bias, pairwise balanced keys will tend to generate a uniform hash distribution, as the informal argument above will still apply. But we cannot expect the distribution of hash values to remain uniform as the degree of higher-order bias increases. We perform an empirical investigation of resilience with higher-order biased data using English text in Section 6.5.

6.2 Case Study of a Function Family

To illustrate the influence of input bias and function resilience on hash uniformity, and to provide a concrete context for the preceding proof, we analyse in detail the following family of functions having a range of resilience. Consider the family of functions $F_n^q \rightarrow F_q$ described by Camion and Canteaut (1999) (theorem 6). For a given $q$ and $n$, the family provides functions with varying resilience $1 \leq t \leq n - 1$, with the additional desirable property that each function has maximal nonlinearity, meaning that the resilience bound of Theorem 2 is an equality so that $d + t = (q - 1)n - 1$ where $d$ is the algebraic degree of the hash function. We made a minor extension to the family to include a 0-resilient function. We refer to the modified function family as the CC99 family.

Definition 13 (CC99 $t$-Resilient function family). The function $f : F_n^q \rightarrow F_q$, over a string $x_0x_1 \ldots x_{n-1}, x_i \in F_q$, where $q$ is even, $q > 4$, and $q \not\equiv 1 \pmod{3}$,
is a $t$-resilient function constructed in the following manner. First, define a 0-
resilient function on the first input symbol,

$$f(x_0) = x_0^{q-2}. \quad (6.12)$$

An $i$-resilient function with $i+1$ inputs may be constructed from an $(i-1)$-resilient
function with $i$ inputs,

$$f(x_0, \ldots, x_i) = (f(x_0, \ldots, x_{i-1}) + x_i^{(q/2)-1})^3. \quad (6.13)$$

This allows us to make a $t$-resilient function with $t+1$ inputs by beginning with
Equation 6.12, and successively applying Equation 6.13. Then the remaining
$n - t - 1$ inputs are added to the function without increasing the resilience by
successively applying

$$f(x_0, \ldots, x_i) = x_i^{q-1} f(x_0, \ldots, x_{i-1}) + (1 - x_i^{q-1}) \alpha f(x_0, \ldots, x_{i-1}), \quad (6.14)$$

where $\alpha \in \mathbb{F}_q \setminus \{0, 1\}$.

This case study uses $q = 8$ and $n = 3$, being the smallest values necessary to
illustrate our arguments. Let the finite field $\mathbb{F}_q$ be expressed as polynomials in
the indeterminate variable $a$, then $\alpha \in \{a, a+1, a^2, a^2+1, a^2+a, a^2+a+1\}$. In
these examples, we selected $\alpha = a$.

First, we construct the 2-resilient function $f_2$ from the family. The construction
begins with Equation 6.12 on the first input variable $x_0$, then Equation 6.13
is applied for variables $x_1$ and $x_2$.

\[
\begin{align*}
f_2(x_0) &= x_0^6 \\
f_2(x_0, x_1) &= (f_2(x_0) + x_1^3)^3 \\
&= x_0^6x_1^6 + x_0^5x_1^3 + x_0^4 + x_1^2 \\
f_2(x_0, x_1, x_2) &= (f_2(x_0, x_1) + x_2^3)^3 \\
&= x_0^6x_1^6x_2^6 + x_0^5x_1^3x_2^3 + x_0^4x_1^3x_2 + x_0x_1^6 + x_1^2x_2^6 \\
&\quad + x_1^4x_2^3 + x_1^6 + x_0x_2^3 + x_2^2
\end{align*}
\]

The degree of $f_2$ is 18, from the first term $x_0^6x_1^6x_2^6$. Note that the resilience bound $d + t = (q - 1)n - 1$ is satisfied, as expected.

The 1-resilient function $f_1$ is constructed by composing Equation 6.12 with Equation 6.13 once, then composing the result with Equation 6.14.

\[
\begin{align*}
f_1(x_0) &= x_0^6 \\
f_1(x_0, x_1) &= (f_1(x_0) + x_1^3)^3 \\
&= x_0^6x_1^6 + x_0^5x_1^3 + x_0^4 + x_1^2 \\
f_1(x_0, x_1, x_2) &= x_2^7f_1(x_0, x_1) + (1 - x_2^7)\alpha f_1(x_0, x_1) \\
&\quad = (a + 1)x_0^6x_1^6x_2^7 + (a + 1)x_0^5x_1^3x_2^7 + ax_0^6x_1^6 \\
&\quad \quad + (a + 1)x_0^4x_2^7 + (a + 1)x_1^2x_2^7 + ax_0^5x_1^3 + ax_0^4 + ax_1^2
\end{align*}
\]

The degree of $f_1$ is 19, from the term $(a + 1)x_0^6x_1^6x_2^7$, and the resilience bound $d + t = (q - 1)n - 1$ is satisfied.

The 0-resilient function $f_0$ is constructed by composing Equation 6.12 with
Table 6.1: Mapping of elements of $\mathbb{F}_8$ to input symbols and to hash values for the input and output of functions $f_0$, $f_1$, and $f_2$. The last column contains the arbitrary probability assigned to each input symbol for the example calculation of hash distributions for each function with biased keys.

\[
\begin{array}{cccc}
\text{F}_8 \text{ element} & \text{input symbol} & \text{hash value} & \text{symbol probability} \\
0 & A & 0 & 0.03 \\
a & B & 1 & 0.06 \\
a^2 & C & 2 & 0.08 \\
a + 1 & D & 3 & 0.11 \\
a^2 + a & E & 4 & 0.14 \\
a^2 + a + 1 & F & 5 & 0.17 \\
a^2 + 1 & G & 6 & 0.19 \\
1 & H & 7 & 0.22 \\
\end{array}
\]

To illustrate the function behaviour, we define a mapping, shown in Table 6.1, of elements of the finite field $\mathbb{F}_8$ to input symbols and to hash values. A biased distribution of symbol probabilities is created to illustrate the effect of input bias on the distribution of hash values, which is shown in Table 6.2. Observe that the high resilience function $f_2$ has hash probabilities which are very close to uniform, but $f_1$ diverges a little, and $f_0$ is distinctly non-uniform.

Equation 6.14 twice.

\[
\begin{align*}
    f_0(x_0) &= x_0^6 \\
    f_0(x_0, x_1) &= x_1^7 f_0(x_0) + (1 - x_1^7) \alpha f_0(x_0) \\
    &= (a + 1) x_0^6 x_1^7 + ax_0^6 \\
    f_0(x_0, x_1, x_2) &= x_2^7 f_0(x_0, x_1) + (1 - x_2^7) \alpha f_0(x_0, x_1) \\
    &= (a^2 + 1) x_0^6 x_1^7 x_2^7 + (a^2 + a) x_0^6 x_1^7 + (a^2 + a) x_0^6 x_2^7 + a^2 x_0^6 
\end{align*}
\]

The degree of $f_0$ is 20, from the term $(a^2 + 1) x_0^6 x_1^7 x_2^7$, and the resilience bound is satisfied.

To explain this observation, consider the function pre-images $f^{-1}(2)$ shown
Table 6.2: The probability distribution of hash values from the example functions $f_0$, $f_1$ and $f_2$ when the input is biased with the symbol probabilities shown in Table 6.1.

<table>
<thead>
<tr>
<th>Hash value</th>
<th>$f_2$</th>
<th>$f_1$</th>
<th>$f_0$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0.122</td>
<td>0.146</td>
<td>0.030</td>
</tr>
<tr>
<td>1</td>
<td>0.124</td>
<td>0.123</td>
<td>0.192</td>
</tr>
<tr>
<td>2</td>
<td>0.123</td>
<td>0.138</td>
<td>0.171</td>
</tr>
<tr>
<td>3</td>
<td>0.127</td>
<td>0.115</td>
<td>0.142</td>
</tr>
<tr>
<td>4</td>
<td>0.125</td>
<td>0.138</td>
<td>0.112</td>
</tr>
<tr>
<td>5</td>
<td>0.126</td>
<td>0.123</td>
<td>0.082</td>
</tr>
<tr>
<td>6</td>
<td>0.125</td>
<td>0.119</td>
<td>0.061</td>
</tr>
<tr>
<td>7</td>
<td>0.129</td>
<td>0.100</td>
<td>0.211</td>
</tr>
<tr>
<td>Entropy (bits)</td>
<td>3.000</td>
<td>2.991</td>
<td>2.813</td>
</tr>
</tbody>
</table>

in Table 6.3. A visual inspection shows that the pre-image of $f_2$ appears to be a well-mixed set of keys, whereas the pre-image of $f_0$ is clearly biased, having a large number of keys beginning with ‘F’, and some with ‘G’ and ‘H’, but none with ‘A’, ‘B’, ‘C’, nor ‘D’. Also, the symbol ‘F’ is more frequent than other symbols. The pre-image of $f_1$ is not so obviously biased as $f_0$, but not as well mixed as $f_2$.

More precisely, the pre-image of each function has the same cardinality, 64 keys, as expected for balanced functions. The number of occurrences of each symbol in each pre-image of functions $f_2$ and $f_1$ is also balanced; each symbol occurs 24 times. Furthermore, the 2-grams in the pre-images of $f_2$ are balanced; each 2-gram appears twice. These observations are in agreement with Property 9 (Page 205): that the pre-images of a $t$-resilient function have balanced $t$-grams.

The uneven distribution of symbols over the pre-images of $f_0$ is shown in Table 6.4, and the uneven distribution of 2-grams over the pre-images of $f_1$ is listed in Table 6.5. We are now able to explain the non-uniform distribution of hash values from functions $f_0$ and $f_1$ shown in Table 6.2. In the biased input keys, ‘H’ was the most likely symbol. For function $f_0$, the keys in pre-image
6.2. Case Study of a Function Family

Pre-image \( f_2^{-1}(2) \):

AAB ABE ACC ADG AEH AFA AGD AHF BAC BBD BCB BDF
BEA BFH BGE BHG CAH CBG CCA CDE CEB CFC CGF CHD
DAD DBC DCE DDA DEF DFG DGB DHH EAE EBB ECD EDH
EEG EFF EGC EHA FAG FBH FCF FDB FEE FFD FGA FHG
GAA GBF GCH GDD GEC GFB GGG GHE HAF HBA HCG HDC
HED HFE HGH HHB

Pre-image \( f_1^{-1}(2) \):

ABB ABC ABD ABE ABF ABG ABH AEA BFA BGB BGC BGD
BGE BGF BGG BGH CAA CDB CDC CDD CDE CDF CDG CDH
DCB DCC DCD DCE DCF DCG DCH DHA EAB EAC EAD EAE
EAF EAG EAH EDA FBB FBC FBD FBE FBF FBH FCB FCC FCD FCE FCF
FGC FCH FDB FDC FDD FDE FDF FDG FDH FEB FEC FED
FEE FEF FEG FEH FFB FFC FFD FFE FFF FFG FFH FGB
FGG FHB FHC FHD FHE FHF FHH
GCA GHB GHC GHD GHE GHF GHG GHH HFB HFC HFD HFE
HFF HFG HHH HGA

Pre-image \( f_0^{-1}(2) \):

FBB FBC FBD FBE FBF FBG FBC FCC FCD FCE FCF
FGC FCH FDB FDC FDD FDE FDF FDG FDH FEB FEC FED
FEE FEF FEG FEH FFB FFC FFD FFE FFF FFG FFH FGB
FGG FHB FHC FHD FHE FHF FHH
GFA GGA GHA HAA

Table 6.3: A pre-image from each of the example functions \( f_2, f_1 \) and \( f_0 \), showing the all of the keys that generate a hash value of 2.

\( f_0^{-1}(7) \) contain more ‘H’ symbols than the other pre-images. Consequently, these keys occur more frequently than the keys in other pre-images, and 7 is the most likely hash value. Generally, we expect that for every biased input distribution generated by a zero-order model, the unbalanced allocation of symbols to each pre-image in a 0-resilient function will cause the hash distribution to be non-uniform.

The function \( f_1 \) also has a non-uniform probability distribution of hash values with the biased keys, although markedly less so than \( f_0 \). However, as each symbol occurs an equal number of times in the pre-images of \( f_1 \), the preceding explanation for the non-uniformity of \( f_0 \) is not applicable. Instead, the non-uniformity of \( f_1 \)
is due to the uneven distribution of 2-grams in its pre-images. As the keys are generated with the biased distribution of symbol probabilities, the 2-grams also have a biased distribution. For example, as ‘H’ is the most frequent symbol, ‘HH’ will be the most frequent 2-gram. The pre-image $f^{-1}_1(0)$ contains 9 of the 16 occurrences of ‘HH’ in the keys (Table 6.5), and we observe that hash value 0 is the most likely (Table 6.2).

Generally this example illustrates that, at least for keys generated by a zero-order bias model, functions with high resilience have pre-images containing a better mixing of likely and unlikely keys, so are expected to generate a more uniform hash distribution.

**Pairwise balance in the CC99 family** To further illustrate the relationship between function resilience and effective pairwise balance, we investigated the pairwise balance of functions from the CC99 family with respect to a small selection of keys. Recall the definition of pairwise balance from Definition 11 (Page 206). We select an arbitrary key and allow it to impose a constraint on the other keys that may be selected to form a pair. In Theorem 12, the constraint is that the keys have matching symbols at at least $j$ positions, and at most $t$
<table>
<thead>
<tr>
<th>2-gram</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>AA</td>
<td>9</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>AB</td>
<td>1</td>
<td>1</td>
<td>8</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>AC</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>8</td>
<td>2</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>AD</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>8</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>AE</td>
<td>1</td>
<td>8</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>AF</td>
<td>1</td>
<td>1</td>
<td>8</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>AG</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>8</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>AH</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>BA</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>8</td>
<td>2</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>BB</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>8</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>BC</td>
<td>9</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>BD</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>8</td>
</tr>
<tr>
<td>BE</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>8</td>
<td>2</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>BF</td>
<td>1</td>
<td>8</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>BG</td>
<td>1</td>
<td>1</td>
<td>8</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>BH</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>8</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>CA</td>
<td>1</td>
<td>8</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>CB</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>8</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>CC</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>8</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>CD</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>8</td>
</tr>
<tr>
<td>CE</td>
<td>9</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>CF</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>CG</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>CH</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>8</td>
<td>2</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>DA</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>DB</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>8</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>DC</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>DD</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>DE</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>DF</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>DG</td>
<td>9</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>DH</td>
<td>1</td>
<td>8</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

Table 6.5: Counts of each 2-gram in the pre-images of function $f_1$. 
positions. The function is said to be pairwise balanced with respect to the first key if the hashes of the set of second keys are balanced. If the function is pairwise balanced with respect to all keys, then the function is said to be pairwise balanced subject to the constraint. Table 6.6(i) shows the number of keys in each pre-image of the 3-resilient function with $n = 6$ and $q = 8$ from the CC99 family with respect to an arbitrarily selected key. If the function was pairwise balanced, the numbers in each column would be identical, however there are small variations in the numbers for all values of $k$.

Our theoretical results for resilient functions relate to key pairs having a range of matching positions. Table 6.6(ii) shows the counts of keys hashing to each value where they key pair matches at $k$ or fewer positions. The magnitude of the variation between sizes of these subsets of the pre-images is measured by the standard deviation. The standard deviations are very small compared to the means, less than 0.2% for all values of $k$, and we can consider the function to be effectively pairwise balanced, at least for the particular key with which these results were obtained.

The standard deviations for the full range of resilient functions using the selected key are shown in Table 6.7. The principal theoretical claim of this chapter, that functions with higher resilience are closer to pairwise balance, is clearly illustrated in the table. One clear trend is that the standard deviation declines with increasing $k$. This is partly because, as $k$ increases, the set of keys having greater than $k$ matching positions shrinks, and as the functions are all balanced, the potential deviation from the mean decreases. However, there is an important exception to this trend: there is an increase in standard deviation from $k = t$ to $k = t + 1$ for the 1-, 2-, and 3-resilient functions. This agrees with the theoretical result that $t$-resilient functions approximate a pairwise balanced function when $k \leq t$, and are not pairwise balanced when $k > t$. 
(i) Number of keys matching at \( k \) positions.

<table>
<thead>
<tr>
<th>hash value</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>14700</td>
<td>12628</td>
<td>4472</td>
<td>872</td>
<td>92</td>
<td>4</td>
</tr>
<tr>
<td>1</td>
<td>14736</td>
<td>12496</td>
<td>4641</td>
<td>796</td>
<td>86</td>
<td>12</td>
</tr>
<tr>
<td>2</td>
<td>14712</td>
<td>12582</td>
<td>4536</td>
<td>836</td>
<td>96</td>
<td>6</td>
</tr>
<tr>
<td>3</td>
<td>14701</td>
<td>12624</td>
<td>4478</td>
<td>868</td>
<td>93</td>
<td>4</td>
</tr>
<tr>
<td>4</td>
<td>14700</td>
<td>12628</td>
<td>4472</td>
<td>872</td>
<td>92</td>
<td>4</td>
</tr>
<tr>
<td>5</td>
<td>14700</td>
<td>12628</td>
<td>4472</td>
<td>872</td>
<td>92</td>
<td>4</td>
</tr>
<tr>
<td>6</td>
<td>14700</td>
<td>12628</td>
<td>4472</td>
<td>872</td>
<td>92</td>
<td>4</td>
</tr>
<tr>
<td>7</td>
<td>14700</td>
<td>12628</td>
<td>4472</td>
<td>872</td>
<td>92</td>
<td>4</td>
</tr>
</tbody>
</table>

Totals: 117649 100842 36015 6860 735 42

(ii) Number of keys matching at \( \leq k \) positions.

<table>
<thead>
<tr>
<th>hash value</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>14700</td>
<td>27328</td>
<td>31800</td>
<td>32672</td>
<td>32764</td>
<td>32767</td>
</tr>
<tr>
<td>1</td>
<td>14736</td>
<td>27232</td>
<td>31873</td>
<td>32669</td>
<td>32755</td>
<td>32768</td>
</tr>
<tr>
<td>2</td>
<td>14712</td>
<td>27294</td>
<td>31830</td>
<td>32666</td>
<td>32762</td>
<td>32768</td>
</tr>
<tr>
<td>3</td>
<td>14701</td>
<td>27325</td>
<td>31803</td>
<td>32671</td>
<td>32764</td>
<td>32768</td>
</tr>
<tr>
<td>4</td>
<td>14700</td>
<td>27328</td>
<td>31800</td>
<td>32672</td>
<td>32764</td>
<td>32768</td>
</tr>
<tr>
<td>5</td>
<td>14700</td>
<td>27328</td>
<td>31800</td>
<td>32672</td>
<td>32764</td>
<td>32768</td>
</tr>
<tr>
<td>6</td>
<td>14700</td>
<td>27328</td>
<td>31800</td>
<td>32672</td>
<td>32764</td>
<td>32768</td>
</tr>
<tr>
<td>7</td>
<td>14700</td>
<td>27328</td>
<td>31800</td>
<td>32672</td>
<td>32764</td>
<td>32768</td>
</tr>
</tbody>
</table>

Totals: 117649 218491 254506 261366 262101 262143

mean: 14706 27311 31813 32671 32763 32768

SD: 11.94 31.96 24.58 2.05 2.96 0.33

Table 6.6: Unbalance of keys matching at \( k \) positions for the 3-resilient function from the CC99 family with \( n = 6 \) and \( q = 8 \). Table (i) shows the numbers of keys hashing to each hash value, and having \( k \) matching positions with the key \( x = [5, 2, 3, 7, 4, 1] \). Table (ii) shows the cumulative numbers: for instance the counts for \( k \leq 2 \) are the sum of the counts for \( k = 0, k = 1 \) and \( k = 2 \) from (i). The mean and standard deviation of the cumulative counts is shown at the bottom.
Another trend that is clear in the columns of Table 6.7 is the decline of standard deviation with increasing function resilience. Again, the only departures from this trend are when \( t = k + 1 \). These results appear to be a reversion to the trend following an especially sharp decline of standard deviation when \( t = k \). This is an important observation concerning the practical circumstances in which resilience is a useful property. If the data is biased such that key pairs with many matching positions are highly unlikely, a function with high resilience approaches pairwise balance with nearly equal numbers of the likely keys in each pre-image. While we cannot be sure that the resulting hash values will be uniformly distributed, as the biased keys are not selected uniformly at random, we can state that the function will achieve a more uniform hash distribution than a function with less resilience.

Note that these results were obtained with a single key, and the function will behave differently with other keys. We did not perform this analysis with all keys as it would have approached the limits of our computational resources, requiring calculations for \( 2^{34} \) key pairs. However, the standard deviations using three keys are shown in Table 6.8. We performed the calculations for several other keys, and these achieved the same results as one of the keys listed in the table. We can see that there are large differences in the standard deviations among the various keys, especially for the functions with low resilience. But the overall trends noted earlier are preserved.

### 6.3 Test For Effective Resilience

Our interest in function resilience is motivated by a desire to minimize collisions when hashing a set of biased keys. When hashing randomly generated keys in Chapter 5, functions having a small degree of unbalance achieved a low num-
### 6.3. Test For Effective Resilience

#### Table 6.7:
The standard deviation of the number of keys hashing to each hash value having less than or equal to $k$ matching positions with the key $x = [5, 2, 3, 7, 4, 1]$ for functions having a range of resilience properties from the CC99 family, with $q = 8$ and $n = 6$.

<table>
<thead>
<tr>
<th>Resilience</th>
<th>$t$</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>2998.82</td>
<td>2340.37</td>
<td>741.10</td>
<td>119.42</td>
<td>9.82</td>
<td>0.33</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>468.07</td>
<td>180.98</td>
<td>226.00</td>
<td>63.91</td>
<td>7.53</td>
<td>0.33</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>74.11</td>
<td>113.00</td>
<td>13.12</td>
<td>24.58</td>
<td>5.24</td>
<td>0.33</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>11.94</td>
<td>31.96</td>
<td>24.58</td>
<td>2.05</td>
<td>2.96</td>
<td>0.33</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>1.96</td>
<td>7.53</td>
<td>10.49</td>
<td>5.91</td>
<td>0.70</td>
<td>0.33</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>0.33</td>
<td>1.65</td>
<td>3.31</td>
<td>3.31</td>
<td>1.65</td>
<td>0.33</td>
<td></td>
</tr>
</tbody>
</table>

#### Table 6.8:
The standard deviation of the number of keys hashing to each hash value having less than or equal to $k$ matching positions with several keys. The functions are from the CC99 family with $q = 8$ and $n = 6$. Key 1 = [5, 2, 3, 7, 4, 1] (from Table 6.7), key 2 = [0, 0, 0, 0, 0, 0] and key 3 = [4, 1, 3, 0, 7, 2]. Several other keys were analysed, and obtained identical results to one these three keys.

<table>
<thead>
<tr>
<th>Resilience</th>
<th>Key</th>
<th>$t$</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2998.82</td>
<td>2340.37</td>
<td>741.10</td>
<td>119.42</td>
<td>9.82</td>
<td>0.33</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>5558.39</td>
<td>3970.28</td>
<td>1134.37</td>
<td>162.05</td>
<td>11.58</td>
<td>0.33</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>3</td>
<td>3276.52</td>
<td>2212.39</td>
<td>603.82</td>
<td>88.72</td>
<td>7.81</td>
<td>0.33</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>468.07</td>
<td>180.98</td>
<td>226.00</td>
<td>63.91</td>
<td>7.53</td>
<td>0.33</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>794.06</td>
<td>340.31</td>
<td>356.51</td>
<td>87.97</td>
<td>8.93</td>
<td>0.33</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>794.06</td>
<td>340.31</td>
<td>356.51</td>
<td>87.97</td>
<td>8.93</td>
<td>0.33</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>2</td>
<td>74.11</td>
<td>113.00</td>
<td>13.12</td>
<td>24.58</td>
<td>5.24</td>
<td>0.33</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>113.44</td>
<td>178.26</td>
<td>23.15</td>
<td>35.06</td>
<td>6.28</td>
<td>0.33</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>83.59</td>
<td>138.75</td>
<td>31.45</td>
<td>23.16</td>
<td>3.67</td>
<td>0.33</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>3</td>
<td>11.94</td>
<td>31.96</td>
<td>24.58</td>
<td>2.05</td>
<td>2.96</td>
<td>0.33</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>16.21</td>
<td>43.99</td>
<td>35.06</td>
<td>3.31</td>
<td>3.64</td>
<td>0.33</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>11.94</td>
<td>31.96</td>
<td>24.58</td>
<td>2.05</td>
<td>2.96</td>
<td>0.33</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>4</td>
<td>1.96</td>
<td>7.53</td>
<td>10.49</td>
<td>5.91</td>
<td>0.70</td>
<td>0.33</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>2.31</td>
<td>8.93</td>
<td>12.57</td>
<td>7.28</td>
<td>0.99</td>
<td>0.33</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>3</td>
<td>1.96</td>
<td>7.53</td>
<td>10.49</td>
<td>5.91</td>
<td>0.71</td>
<td>0.33</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>5</td>
<td>0.33</td>
<td>1.65</td>
<td>3.31</td>
<td>3.31</td>
<td>1.65</td>
<td>0.33</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>0.33</td>
<td>1.65</td>
<td>3.31</td>
<td>3.31</td>
<td>1.65</td>
<td>0.33</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>0.33</td>
<td>1.65</td>
<td>3.31</td>
<td>3.31</td>
<td>1.65</td>
<td>0.33</td>
<td></td>
</tr>
</tbody>
</table>
ber of collisions, similar to the number expected from a balanced function. The hash test was unable to distinguish between these functions, and the slightly unbalanced functions were accepted as being ‘effectively balanced’, at least for the purpose of hashing random keys. A similar approach is appropriate when considering the effect of resilience on hash functions, and we define an effectively $t$-resilient function to be a function which is indistinguishable from $t$-resilient function when assessed by the number of collisions generated. To illustrate, consider constructing a function by starting with a $t$-resilient function, then swapping two keys between two pre-images. The function will no longer be $t$-resilient in the strict mathematical sense, but when hashing a set of keys, the number of collisions will be indistinguishable from those generated by the original $t$-resilient function. In Chapter 5, we used the balance measure to quantify the degree of unbalance of a function. However, there is no similar measure with which we can quantify effective resilience. Instead, we can only compare functions by observing the collisions. A similarly stochastic approach is employed elsewhere to measure function avalanche using avalanche matrices, as described in Section 2.2.1.

We can now apply the theoretical results to implement a black-box test for effective resilience, based on the hash collision test described in Chapter 5. The test uses sets of keys that are generated by a zero-order biased model. By increasing the degree of bias when generating the key sets, we are able to increase the expected number of matching positions between pairs of keys. If the hash function has high effective resilience such that $t > k$, we know from Theorem 12 that the function is effectively pairwise independent. Then the probability distribution of the hash values will be uniform, and we expect to observe a mean LLPS value within the critical region of the test. On the other hand, when $t < k$ we know from Theorem 12 that the function is not pairwise balanced. Then the probability distribution of the hash values will be non-uniform, and we expect
the mean observed LLPS value to be outside of the critical region of the test.

A single test is only able to determine whether \( t \) is greater than or less than \( k \). However, by performing several tests using key sets having a range of bias, and therefore having a range of expected values for \( k \), we can obtain a coarse estimate of \( t \). The relationship between \( k \) and the bias of the key sets is not precise, so we are not able to directly relate the measurements of \( \bar{X} \) from the tests to a precise value \( t \). Nonetheless, we can test multiple hash functions and rank them in order of effective resilience, or perhaps compare the results with those obtained using a function with known resilience.

In each test, 1000 trials are performed, each using a set of 256 keys generated using a biased zero-order distribution. The distributions used to generate the key sets have the same entropy, but the symbols are randomly permuted. Specifically, a key \( s_1s_2\ldots s_n \) is a string of symbols \( s \in \Sigma \). We define a probability distribution \( p_j, j \in \{1,2,\ldots,q\}, q = |\Sigma| \) having the same cardinality as the alphabet. In the zero-order generation model, the probability of each symbol occurring is \( \Pr[s] = p_{\pi(s)} \), where \( \pi : \Sigma \rightarrow \{1,2,\ldots,q\} \) is a mapping of the symbols onto the probability distribution. In each trial, the probability distribution \( p_i \) is the same, but a different randomly generated mapping \( \pi \) of symbols to probabilities is used. Generating the keys in this manner ensures that the symbols have a uniform expected frequency over all of the trials, but the expected number of matching positions \( k \) between keys can be varied.

The specific steps of the test procedure are,

1. Generate several probability distributions over the alphabet size \( q \) having a range of degrees of bias. Then perform the remaining steps using each distribution.
2. Generate 1,000 random permutations of \( \{1, 2, \ldots, q\} \).

3. Using each permutation from Step 2, map symbols onto the probability distribution and generate a set of 256 distinct keys.

4. For each key set, make a random selection of 7 bit positions in the hash.

5. For each key set, hash the keys and convert the hashes into 7-bit hashes, known as a selected hash, using the selected positions from step 4. Insert the selected hashes into a hash table of size 128, and measure the LLPS.

6. Find \( \bar{X} \), the mean of the 1,000 LLPS measurements.

The test procedure produces a measurement of \( \bar{X} \) for each of the tests having a different probability distribution. The null hypothesis, that the function resilience \( t \) is greater than the number of matching positions between keys in the key sets, is accepted when \( \bar{X} \leq 6.541 \).

### 6.4 Validation of the Resilience Test

To validate the test for resilience, we applied the test procedure to functions having a known resilience. We used the CC99 hash function family, Equations 6.12, 6.13, and 6.14, with all functions \( h : \mathbf{F}_{128}^{32} \rightarrow \mathbf{F}_{128} \) having symbols and hashes of 7 bits and key length of 32 symbols. We used all 32 functions from the family, with resilience \( t \) varying from 0 to 31. We implemented the CC99 family using Sage (Stein et al., 2012; Decker et al., 2011; Gautier et al., 2005), an open source mathematical system implemented in Python. Our program uses the Sage module for symbolic algebra on a finite field to generate lookup tables implementing the hash function. These tables were then included by a C program that efficiently performed the hash calculations. The experiment followed the procedure
described in Section 6.3. For each pairing of a resilient function and a biased data set, 1000 trials were run hashing 256 keys, and the mean of the observed LLPS values, $\bar{X}$, was recorded.

The biased input keys were artificially generated using a zero-order model. Nine probability distributions were used, with a range of degrees of bias, denoted by the bias parameter $\alpha$, $0 \leq \alpha \leq 1.0$. The value $\alpha = 0$ gives a uniform distribution, and $\alpha = 1.0$ produces a single outcome having probability 1.0. The relation for generating the probability distribution was,

$$p_i = \frac{(\alpha(q - 1) + 1)(1 - c_{i-1})}{q - i + 1},$$

(6.24)

where $p_i$ is the probability of the $i$-th term, and $c_{i-1}$ is the cumulative probability of the previous $i-1$ terms. The generated distribution is a discrete approximation of the exponential distribution, which can be seen from the relation: $1 - c_{i-1}$ is the remaining probability available to be allocated, and $q - i + 1$ is the number of probabilities yet to be generated. A value of $\alpha = 0.5$ will allocate half of the remaining probability to the next value.

At the start of each trial, the input alphabet was randomly permuted before assignment to the probabilities, ensuring that in each trial the probability distribution of the symbols was different, while having the same entropy. The distribution entropies, and charts of the probabilities, are shown in Table 6.9 and Figure 6.2. To generate the strings in each trial, each symbol was randomly selected according to the symbol distribution for that trial. Each symbol was chosen independently of previously symbols, and duplicate keys were discarded.

**Results** The observed mean LLPS values from the tests are listed in Table 6.10, and graphed in Figure 6.3. The functions with resilience of 10 or greater passed
Figure 6.2: Probability distributions of input symbols for the experiment in Section 6.4, using several values of the bias parameter, $\alpha$. The top chart shows the full distribution, and the lower chart is zoomed in to lower part of the probability range.
6.4. Validation of the Resilience Test

<table>
<thead>
<tr>
<th>$\alpha$ (bits)</th>
<th>Entropy (bits)</th>
<th>$p_m$</th>
<th>$E(k)$</th>
<th>95-th percentile</th>
</tr>
</thead>
<tbody>
<tr>
<td>(uniform)</td>
<td>7.000</td>
<td>0.008</td>
<td>0.250</td>
<td>1</td>
</tr>
<tr>
<td>0.02</td>
<td>6.197</td>
<td>0.016</td>
<td>0.520</td>
<td>2</td>
</tr>
<tr>
<td>0.05</td>
<td>5.332</td>
<td>0.032</td>
<td>1.011</td>
<td>3</td>
</tr>
<tr>
<td>0.07</td>
<td>4.939</td>
<td>0.042</td>
<td>1.349</td>
<td>3</td>
</tr>
<tr>
<td>0.10</td>
<td>4.487</td>
<td>0.058</td>
<td>1.870</td>
<td>4</td>
</tr>
<tr>
<td>0.20</td>
<td>3.511</td>
<td>0.117</td>
<td>3.735</td>
<td>7</td>
</tr>
<tr>
<td>0.30</td>
<td>2.876</td>
<td>0.182</td>
<td>5.819</td>
<td>10</td>
</tr>
<tr>
<td>0.40</td>
<td>2.385</td>
<td>0.255</td>
<td>8.162</td>
<td>12</td>
</tr>
<tr>
<td>0.50</td>
<td>1.969</td>
<td>0.338</td>
<td>10.815</td>
<td>15</td>
</tr>
</tbody>
</table>

Table 6.9: Distributions of the zero-order biased strings generated for the experiment. Several sets of strings were generated with various degrees of bias ranging from unbiased, $\alpha = 0.0$, to highly biased, $\alpha = 0.5$. The degree of bias is indicated by the distribution entropy. The last three columns give parameters relating to the probability distribution of $k$, the number of matching positions between a pair of keys, from Equations 6.10 and 6.11. The upper tail of the distribution is indicated by the 95-th percentile: 95% of key pairs have $k$ less than or equal to this value.

the test for all key sets, generating a random distribution of hashes, even with the most biased input. There were 11 observations from these functions that were outside the 95% critical region, which is close to the expected rate of 10 (5%) failures from the 198 trials. These functions were statistically indistinguishable from a truly random hash function, at least when hashing biased keys from a zero-order source.

On the other hand, the 0-resilient function only achieved a uniform hash distribution for unbiased data ($\alpha = 0$). Between these upper and lower extremes of resilience, there is a clear monotonic relation between the degree of bias in the input data, and function resilience: (i) with data having certain degree of bias, a function having greater resilience will achieve a more uniform hash distribution, up to a resilience threshold where the hash distribution is effectively uniform; and (ii) a function with a certain degree of resilience will uniformly dis-
tribute the hashes of biased keys up to threshold of bias, beyond which the hash
distribution becomes increasingly non-uniform. This threshold of bias is higher
for functions with higher resilience. Consequently, the test is able to distinguish
between functions having different degrees of effective resilience, and rank them
in order.

In addition to validating the test for effective resilience, this experiment sup-
ports the theoretical results presented earlier in this chapter. As predicted, the
functions having resilience \( t > k \) passed the test, and are effectively pairwise inde-
pendent. Comparing the results with the expected values of \( k \) listed in Table 6.9,
for all key sets, functions with \( t \) less than the 95-th percentile value for \( k \) passed
the test. This suggests that a small fraction of pairs may have \( k > t \) while still
achieving effective pairwise independence. From our results, when \( k > t \) for 5% of
pairs, there is a high likelihood of obtaining a uniform distribution of hashes.
Furthermore, in 7 of the 9 keys sets, functions having resilience \( t > E(k) \) passed
the test. The other two cases were \( \alpha = 0.02 \) having \( E(k) = 0.52 \) where the
1-resilient function failed the test, and \( \alpha = 0.1 \) having \( E(k) = 1.87 \) where the
2-resilient and 3-resilient functions did not pass. It appears that the threshold of
function resilience needed to achieve pairwise independence is close to \( t = E(k) \).

6.5 Hashing English Text

We now briefly investigate the performance of resilient functions on data that
is not generated by a zero-order source. While we know from the theoretical
results that a function with \( t > k \) is effectively pairwise balanced, the data may
be biased so that the function does not produce pairwise-independent hashes.
Consequently, we do not expect resilient functions to perform as well with key
sets that are not generated by a zero-order source. To demonstrate this, we
### Table 6.10

<table>
<thead>
<tr>
<th>Resilience (t)</th>
<th>Symbol distribution bias parameter, α</th>
<th>0.0</th>
<th>0.02</th>
<th>0.05</th>
<th>0.07</th>
<th>0.1</th>
<th>0.2</th>
<th>0.3</th>
<th>0.4</th>
<th>0.5</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td>6.457</td>
<td>10.241</td>
<td>16.518</td>
<td>21.076</td>
<td>27.865</td>
<td>50.501</td>
<td>75.212</td>
<td>100.079</td>
<td>125.722</td>
</tr>
</tbody>
</table>

**Table 6.10**: Means of the observed LLPS values from the experimental validation of the resilience test. Nine sets of keys ranging from unbiased (α = 0.0) to most biased (α = 0.5) were hashed by functions having various degrees of resilience, t, from the CC99 family. Observations which are outside of the critical region (a mean LLPS greater than 6.540) are in bold type.
Chapter 6. Hashing Biased Data

Figure 6.3: Means of the observed LLPS values from the experimental validation of the resilience test, which are listed in Table 6.10. Nine sets of keys ranging from unbiased ($\alpha = 0.0$) to most biased ($\alpha = 0.5$) were hashed by functions having various degrees of resilience, $t$, from the CC99 family. The top chart shows the full range of observations, and the bottom chart shows the region close to the theoretical mean LLPS. Observations above the line at 6.540 are outside the 95% confidence interval, and have less than 0.05 probability of being generated by a uniform distribution of hash values.
hashed English text using the hash functions from the CC99 family used in the previous section.

The input keys were obtained from the EnronSent corpus (Styler, 2011), a collection of 96107 email messages collected during the prosecution of Enron corporation. The email headers and attachments have been removed, retaining text mainly entered by people. Two sets of keys were extracted: Separated Strings, where a random length gap between 10 and 30 characters was skipped before copying the next key (a string of length 32) to ensure that dependence between the keys was not significant; and Overlapping Strings, where the next key begins at the second character of the previous key, as would be used in a rolling-hash algorithm. The zero-order empirical entropy of the keys was 4.845 for Separated Strings, and 4.776 for Overlapping Strings. As with Experiment 6.4, each function from the CC99 family was evaluated by recording the mean LLPS from 1000 trials, each hashing 256 keys. No trial contained duplicate keys.

The observed mean LLPS values are shown in Figure 6.4. There are clear differences from the results obtained with zero-order biased keys shown in Figure 6.3. With English text, only the functions with very high resilience (greater than or equal to 27) passed the test. The functions having lower resilience performed noticeably worse with the English text than with the zero-order biased keys. As the number of matching positions $k$ between keys in the English text data is much less than 27, we conclude that the keys are biased in some way so that pairwise balanced keys do not result in a uniform distribution of hash values. We suggest that this is because English text is not a zero-order source. Each letter has a strong dependency on the preceding letters: knowledge of the preceding five letters allows much improved prediction of the next letter (Teahan and Cleary, 1997). This more complex bias model leads to more complex constraints between pairs of keys, which, in turn, are not well suited to the resilient
Figure 6.4: Mean observed LLPS from hashing English text with resilient functions. The upper chart shows the full range of observations, and the lower chart shows the region close to the theoretical mean LLPS. The upper limit of the 95% critical region at 6.54 is shown as a horizontal line.
functions. The overlapping keys have an additional constraint, compared to the separated keys, and the resilient functions produce a less uniform distribution of hashes. Overall, these results confirm our earlier result that resilient functions are best suited to data from a zero-order source. That the resilient functions do not perform as well when the data contains additional biases is expected. For practical situations with biased keys from complex sources, these results suggest that very high resilience is needed.

6.6 Resilience of Popular Hash Functions

We investigated the resilience of popular hash functions. High function resilience was not an explicit design goal during the development of these functions. However, as there is extensive experience of use of them in numerous systems, we accept that they behave well with a wide variety of biased data. Consequently, we expect that they will have a high degree of resilience.

The functions were tested using the zero-order biased key sets from Section 6.4: nine sets of keys ranging from unbiased \((\alpha = 0)\) to highly biased \((\alpha = 0.5)\). The test procedure was similar, of 1 000 trials inserting the hashes of 256 keys into a table of size 128. The additional step of random bit-selection was performed for each trial to convert the hash values to 7 bits, as described in Chapter 5. The results are shown in Figure 6.5.

The function Adler32 is a checksum used in the zlib compression library. It is not intended to be used as a hash function, and is known to produce a skewed distribution of values from short keys (Stone et al., 2002). It was included in these tests as an example of a function likely to have poor resilience. The results confirm this, showing that it is not even balanced for short keys as it did not pass the test with the unbiased key set, and the hash distribution becomes progressively
Figure 6.5: Mean observed LLPS of several popular hash functions tested using the zero-order biased keys from Section 6.4. Adler32 is a checksum included to show a function with poor resilience. Polynomial division and Karp-Rabin are rolling hash functions. The other functions are non-universal, non-cryptographic hash functions widely used in hash-based data structures.
less uniform with more biased keys.

Two rolling hash functions were tested: the Karp-Rabin hash function with 
radix = 256 and prime divisor = 36 028 797 018 963 913 = 2^{55} − 55 producing a 54-
bit hash value; and the 62-bit Cohen hash function using division by irreducible 
polynomial described in Chapter 3. The implementation of the CRC32 function 
was from the zlib module in the Python 3.3.0 distribution. The other functions, 
SpookyHash, Murmurhash2, MurmurHash3, CityHash, and the Python built-in 
hash function were described in Section 2.2.

All of these functions passed the test for all sets of keys. When the functions 
with known resilience were tested in Section 6.4, only functions having resilience 
greater than or equal to 10 passed the test with all of the key sets, so we conclude 
that the resilience of these popular hash functions is at least 10.

6.7 Discussion

We have presented several results indicating that high function resilience is nec-
ессary when hashing biased data. The proof in Section 6.1 showed that resilient 
functions are effectively pairwise balanced when key pairs have a low number of 
matching positions, and that increased resilience is needed for pairs having more 
matching positions. The test with English text demonstrated that the connec-
tion between pairwise balance and effective pairwise independence is complex, and 
that a higher resilience was required to obtain a uniform distribution of hash val-
ues compared to the test using zero-order biased keys. Finally, all of the popular 
hash functions that we tested were found to have high resilience. While function 
resilience has not been explicitly stated as a design goal of these functions, the 
functions nonetheless require high resilience in order to be successful.

Our particular motivation for this investigation was to determine whether the
rolling hash functions are suitable for use in a CBC system. The results show that, when the keys do not overlap, the rolling hash functions have high resilience, and also that they have similar resilience to the popular hash functions which are widely accepted to be well behaved. Consequently, we conclude that the rolling hash functions are suitable for use in CBC systems such as cobald.

The popular hash functions, SpookyHash, CityHash, MurmurHash2 and MurmurHash3, are not well suited for use in cobald as they are not rolling-hash functions: they do not calculate the hash of the next position using the previous hash, instead processing the entire key to calculate the next hash. Hence we conclude that the rolling hash functions are the preferred hash functions to use in cobald.

We noted at the start of this chapter that resilience imposes a bound on function avalanche, and the necessity for high resilience suggests that avalanche is not an important property of non-cryptographic hash functions. This proposition is an important question for further investigation, as it contradicts the currently accepted view. Until that is resolved, we suggest that a hash function should possess at least a small degree of avalanche. Our results with English text obtained a uniform distribution of hash values when $t = 27$, while the maximum available resilience was $t = 31$, leaving scope for 1 or 2 degrees of avalanche. Generally, our experience suggests that a good hash function should have a balance of properties if it is to perform well with a wide variety of input data, and it is clear that a maximally resilient function will not perform well with highly similar keys differing in only one or two positions; a realistic scenario in some practical situations.

Resilient functions have the further advantage of being symmetric in the sense that, because they have balanced $t$-grams, a permutation of the symbols in the alphabet will not affect the pairwise balance. Therefore, the hash value distri-
bution is not affected by which symbols are most or least frequent, but only by the distribution of frequencies. A consequence of this property is that a resilient function is a safe choice when little is known about the biased distribution of the input data. Furthermore, while it may be possible to exploit knowledge of the biased distribution to achieve a more uniform hash distribution using a non-resilient function, that function may achieve a poorer hash distribution when used with data biased in a different way.

There are many avenues of further work to extend this investigation. The theoretical result could be extended to find the balance fraction between triples or larger sets of keys, and the conditions needed to obtain effective three-wise and higher independence. Improved independence obtains a more uniform distribution of hash values compared to pairwise independence. Furthermore, the theoretical LLPS result used in the collision tests is for the ideal case of complete independence, while we have only proved results for effective pairwise independence. The experimental results were consistent with the ideal case, although some divergence should have been observed if the functions were only pairwise independent. A potential explanation is that the functions with high resilience, such as when \( t \) is significantly greater than \( k \), are effectively three-wise independent or greater.

Another aspect needing further work is to improve the resilience test so that it is better able to discriminate between adjacent values of \( t \). We suggest that this can be achieved by testing with key sets having exactly \( k \) matching positions between each pair, rather than the zero-order keys used here, where \( k \) follows a distribution.

Last, the suitability of highly resilient functions for hashing typical data should be rigorously tested using a range of key sets from common applications, and also some pathological data sets to anticipate worst case situations. If highly resilient
functions prove to be suitable, there remains substantial work to improve their speed, which has not been addressed in this investigation.
6.7. Discussion
Chapter 7

Conclusions

The storage of large collections of data, which is being created at an ever-increasing rate, is an acute and growing problem. To address this issue, this thesis advances the art of collection-based compression (CBC). While existing storage techniques, such as single-file compression, provide some alleviation, CBC can provide further benefits on suitable data. It achieves this by exploiting long-range matching data from all parts of a data collection, and we have demonstrated that substantial improvement can be obtained on collections of genomic data and regular archives of web sites where the data has a high degree of similarity.

The specific focus of this investigation is the use of string hashing to efficiently identify long-range matching data. In this concluding chapter, we summarise the outcomes of the investigation, then discuss their implications for further work. We describe avenues to extend this research program, as well as the potential impact of this research on the design and evaluation of non-universal hash functions and compression of data collections generally.
7.1 Thesis Outcomes

We proposed a new algorithm, in Chapter 3, for delta encoding with respect to multiple reference files that uses time linear with the size of the input file. We implemented this algorithm in the cobald CBC system. It obtained significant improvement of compression effectiveness compared to single-file compression: between 5 and 20 times the compression obtained from compressing the files individually using 7-zip and xz, the most effective single-file compressors. The cobald system was compared to other delta encoders and found to achieve better compression and to use less memory for inputs larger than 1.5 Gbytes (sum of the input and reference file sizes). The most effective settings for the key algorithm parameters were found to be a trade-off between compression and resource use. Significant compression was obtained using $n$-gram lengths of 256 and 1024 bytes, with $n = 256$ achieving better compression, but at the cost of a larger index and using more memory. We also investigated two methods to construct the index, finding that fixed-offset selection of $n$-grams achieved better compression, while content-defined selection used significantly less memory.

We proved that the hashes from a rolling hash function are not limited to pair-wise independence when the keys do not overlap by more than half of their length. This result is applicable to the hash index used in our CBC algorithm when every $n$-th $n$-gram is selected with fixed-offset selection, and also for most $n$-grams selected using content-defined selection.

We demonstrated, in Chapter 4, the effectiveness of compressing entire collections by storing the CBC deltas, obtaining between 3 times compression improvement on a 55 Gbytes collection of web files, and a 17 times improvement on a 26 Gbytes collection of genome files, compared to compression of the files individually by 7-zip. Moreover, the total compression time was faster using
cobald while it used slightly more memory than 7-zip on some of the data.

We satisfied the requirements of a successful CBC system relating to resource use, (as stated in Chapter 1), when the collection is stored in a compressed form as cobald delta files. Memory used by cobald is sublinear with the size of the collection, at least when there is a high degree of similarity between the data, and the compression time was linear with the input file size.

We developed, in Chapter 5, a new black-box test procedure for evaluating the collision properties of a hash function, which obtains a $p$-value for the hypothesis that the hash function is indistinguishable from an ideal hash function.

We proved, in Chapter 6, that a $t$-resilient function is effectively pairwise independent when the set of keys is biased such that key pairs have fewer than $t$ matching positions. Using the test procedure from Chapter 5 we confirmed this result by hashing sets of zero-order biased keys having a range of degrees of bias.

Finally, we demonstrated that high resilience is a necessary property of hash functions when hashing data biased in ways which are encountered in practice by hashing English text with functions having a range of resilience, and also by showing that several commonly used hash functions which are thought to be well behaved, have high resilience.

Three key outcomes of this thesis are of immediate interest to practitioners. We have established that compressing a collection using CBC may provide a significant improvement in compression compared to the current practice of compressing files individually, while using a similar amount of computer memory and in less time. Owners of data collections having a high degree of similarity between the collection files can expect to reduce their storage needs several-fold by adopting CBC.

Users of non-universal, non-cryptographic hash functions, as well as developers of these functions, can immediately adopt the black-box collision test. By
allowing the tester to select the significance and power of the test, the tester can avoid drawing unwarranted conclusions from testing too little, or conversely from wastefulness by performing more tests than necessary. Furthermore, the test provides an absolute measure that may lead to acceptance of good hash functions that were previously overlooked because they were marginally outperformed by other functions in relative tests.

The results concerning hash function resilience are also of immediate interest to designers and evaluators of non-universal, non-cryptographic hash functions. Current practice for testing hash functions does not include any consideration of function resilience, but we have shown that it is an essential property. The test for function resilience can be readily added to hash function test suites. A related implication arises from the trade-off between function resilience and avalanche. Current practice is to test hash functions for avalanche, requiring that the hash function has at least one bit of avalanche, sometimes two, on the assumption that a high degree of avalanche is necessary for a good hash function. Our results show that, while a small degree of avalanche is useful, any more may be detrimental as it limits the degree of resilience.

7.2 Further Work

There are many avenues that would extend the ideas developed in this thesis. The CBC algorithm is incomplete in two key respects. A memory limit is not described, and we did not investigate this in detail, but it is essential if the algorithm is to be deployed in practice. An obvious resolution is to discard a record from the index before adding a new one when the hash index in memory reaches its maximum size. If the lowest or highest hash in the index is discarded, we expect that the discarded records will be from random locations in the collection.
While it is not clear how this will affect system performance, we expect it will lead to a gradual degradation of compression, as a similar scheme is used by other delta encoders, such as \texttt{xdelta}, leading to a steady decline in compression as the input file size increases.

File deletion is the other respect in which our algorithm is incomplete, and was not investigated during the course of this thesis. The problem arises when the collection files are stored as the compressed deltas. Before a file is deleted from the collection, the COPY blocks in other delta files that refer to the deleted file must be replaced. There are multiple ways to implement this; an elementary scheme would be to replace the COPY blocks with an ADD block containing the original string, then re-compress the altered files. Importantly, in a collection of delta files such as we discussed in Chapter 4, COPY blocks can only refer to ADD blocks in other files, so there is no cascading of COPY block re-writes.

A related issue is that the compression obtained when compressing a collection depends on the order in which files are added. In \texttt{cobald}, COPY blocks in later files refer to ADD blocks in files already in the collection, but an alternative scheme could rewrite the collection files to refer to the new file. We did not explore this phenomenon in detail, but it is possible that other methods may achieve better results than \texttt{cobald}.

There is scope for additional development of the algorithm implementation in \texttt{cobald}. We expect one fruitful line of development to be the use of pseudo-random access into the compressed delta files. An example of pseudo-random compression is the use of the \texttt{Z\_FULL\_FLUSH} option in \texttt{zlib} to insert sync points. The file can then be partially decompressed starting from a sync point. Note that this blocking technique results in less compression than would be achieved otherwise as data preceding the sync point cannot be used to compress the data following it. Essentially, the compressor is using a smaller sliding window. Cur-
rency in cobald, the stored deltas are compressed (by 7-zip or other single file compressor) files, and must be decompressed before cobald can read the delta file during encoding. With pseudo-random access, only the desired block of the compressed delta needs to be decompressed, providing a significant saving of time and eliminating the need to temporarily store the decompressed files. An interesting possibility is that cobald may mitigate the reduction in compression obtained by the second stage as it will have removed all of the large matching strings beforehand. Furthermore, compressing the delta in blocks permits the use of a parallel compressor for the second stage. Some commonly used single-threaded compressors have been re-implemented using this approach, for example pigz\textsuperscript{1}, a multi-threaded implementation of gzip; and PBZIP\textsuperscript{2}, a re-implementation of bzip2. This would provide some improvement in the overall compression time for cobald.

Another aspect of the implementation of cobald which has not been considered in this thesis is its interaction with other systems, and what additional facilities it should provide to support efficient operation in a larger application. For instance, a common scenario may be the regular addition to the collection of copies of a file system or web site, similar to Web snap-shots collection used here. These images will include files which have not changed since the previous addition, and which will already be in the collection. Re-compressing the file and adding it to the collection will not consume significant additional storage space, but it is an unnecessary use of time. However, the only way to compare files in a cobald collection is to decompress them. By storing a hash of the unencrypted file with the compressed delta in the cobald collection, duplicate files could be detected more efficiently.

\textsuperscript{1}http://www.zlib.net/pigz/
\textsuperscript{2}http://compression.ca/pbzip2/
There are several aspects of black-box collision test that would benefit from further investigation. We did not justify the selection of some parameters of the black-box test, in particular the hash table size of 128, which consequently appear to be arbitrary choices. and the key set size of 256. In our investigation, these values were chosen for pragmatic reasons. Specifically, we were able to find the theoretical solution to the balls-in-bins problem for these values, and we were unable to implement a family of resilient functions having a larger range. While this does not diminish the effectiveness of the test, it is possible that other parameter choices may provide more statistical power.

Further work is needed to definitively determine whether the power of the test is reduced by the bit-selection process when inserting the hash into a table of size 128. Our assessment, based on an informal argument and an empirical evaluation, suggests that it is not, but the possibility remains that bit-selection may, in some circumstances, be detrimental.

Another avenue of investigation is to resolve the mismatch between the black-box test, which compares the hash function to an ideal hash function having independent hashes, and the proof regarding effective pairwise independence of resilient functions. One approach is to extend the proof to effective 3-way and higher degrees of independence. Alternatively, it would be interesting to find an analytic solution to the balls-in-bins problem with pairwise independent balls, which to our knowledge, and perhaps surprisingly, has not been found.

The test for resilience can be improved by using a more carefully selected set of keys. In particular, if the keys had exactly $k$ matching positions between each pair in the set, rather than merely a distribution having mean $k$, the resilience of the function could be resolved with greater fidelity. However, generating such a key set is non-trivial if we are to also satisfy the condition that the keys are zero-order biased.
This thesis is a significant contribution to the fields of string hashing and data compression with immediate implications for practitioners and providing several research questions for future investigation.


Bibliography


Minerva Access is the Institutional Repository of The University of Melbourne

Author/s:
PEEL, ANDREW

Title:
String hashing for collection-based compression

Date:
2015

Persistent Link:
http://hdl.handle.net/11343/57184

File Description:
PhD thesis