Algorithms and Architecture for
Fast, Distributed Text Matching

by

Dmitri Nikulin

Thesis
Submitted by Dmitri Nikulin
in partial fulfillment of the Requirements for the Degree of
Bachelor of Software Engineering with Honours (2770)

Supervisor: Dr. David Squire

Clayton School of Information Technology
Monash University
November, 2009
## Contents

Abstract ................................................................. vi

1 Introduction .......................................................... 1
  1.1 Motivation ......................................................... 1
  1.2 Contribution ...................................................... 1
  1.3 Structure .......................................................... 1

2 Literature review ....................................................... 3
  2.1 Automated plagiarism detection ................................. 3
     2.1.1 Substring matching ........................................... 4
     2.1.2 Ranking and Fingerprinting ................................ 4
     2.1.3 Source code matching ....................................... 4
  2.2 Exact substring matching ........................................ 5
     2.2.1 Suffix trees .................................................. 6
     2.2.2 Suffix arrays ................................................ 7
     2.2.3 Karp-Rabin string search ................................. 8
  2.3 Distributed processing and storage .............................. 9
  2.4 Conclusion ........................................................ 10

3 Detection primitives ................................................... 11
  3.1 Text representation .............................................. 11
     3.1.1 Efficiency ................................................... 11
     3.1.2 Language neutrality ........................................ 11
     3.1.3 Importability ............................................... 12
     3.1.4 Encoding technique ......................................... 12
  3.2 n-gram hashing .................................................... 12
  3.3 Generalised n-gram hash table .................................. 13
     3.3.1 Structure .................................................... 13
     3.3.2 Table elements .............................................. 13
     3.3.3 Abstract algorithms ........................................ 14
     3.3.4 Filtered indexing .......................................... 14
     3.3.5 Partial indexing ............................................ 14
  3.4 Index — CPU–efficient n-gram hash table ....................... 15
     3.4.1 Hash filtering .............................................. 15
     3.4.2 Element dimensions ........................................ 15
  3.5 Mindex — CPU– and memory–efficient n-gram hash table .... 16
     3.5.1 Offset encoding ............................................. 16
     3.5.2 Hash filter .................................................. 16
     3.5.3 Further encoding for memory .............................. 17
     3.5.4 Implemented specification ................................. 17
  3.6 Dexin — Inverted index for scalable batch search ............ 18
6 Conclusion ................................................................. 45
  6.1 Primitives .............................................................. 45
  6.2 Techniques ............................................................. 45
  6.3 Results ................................................................. 46
  6.4 Further work .......................................................... 46

Appendix A Raw results .................................................. 47
  A.1 Primitive performance .............................................. 47
      A.1.1 Varying corpus size ........................................ 47
      A.1.2 Varying table size ........................................ 48
  A.2 Mindex performance ............................................... 48
      A.2.1 Varying corpus and candidate sizes ..................... 48
      A.2.2 Varying corpus and table sizes ......................... 48
  A.3 Threaded search scalability ...................................... 49
  A.4 Batch search performance ....................................... 50
      A.4.1 One Mindex per thread ................................ 50
      A.4.2 One Dexin per thread ................................ 50
  A.5 Distributed search performance ............................... 51
      A.5.1 Round trip time ........................................... 51
      A.5.2 Response communication time ........................... 51
Algorithms and Architecture for Fast, Distributed Text Matching

Dmitri Nikulin
dnik3@student.monash.edu.au
Monash University, 2009

Supervisor: Dr. David Squire
David.Squire@infotech.monash.edu.au

Abstract
Chapter 1

Introduction

1.1 Motivation

The prevention and detection of plagiarism is of great interest to both educational and commercial institutions. For their curriculum to be effective, teaching staff must ensure that student work is original, at minimum a sophisticated permutation of existing work, generally accepted as a proxy for understanding. Intellectual Property specialists are often required to investigate potentially derivative or infringing work, both to avoid legal action against their parties, and to accumulate evidence against infringing parties.

With an ever-growing number of texts being written in natural and computer languages, and an already massive global archive of text freely available to any writer, accurately detecting duplication of work remains a significant problem technologically, logistically and procedurally.

The mature fields of Text Search and Information Retrieval have accumulated many techniques for efficiently answering short human-generated queries even on massive text corpora. Precise duplication detection necessarily requires processing far larger queries with at worst an exhaustive search of the corpus, which altogether greatly complicates the techniques required for efficient solutions.

1.2 Contribution

The research will include the definition and implementation of a number of detection primitives, some known and some novel, that can be integrated into detection techniques to meet these high-level requirements. The software produced will form a robust and economical duplication detection system able to meet these requirements for a large body of users and texts. The intent is to support realistic requirements with a high degree of flexibility in usage and system specifications. Multiple primitives and techniques will be designed to be applicable in the design and implementation of real systems for varying requirements.

1.3 Structure

Chapter 1 introduces the high-level context and contribution of the research. Chapter 2 explores relevant research literature. Chapter 3 details duplication detection primitives and their uses. Chapter 4 details system implementation techniques for scalable and efficient duplication detection. Chapter 5 defines several experiments intended to evaluate performance characteristics of the primitives and techniques, and presents and discusses their results. Chapter 6 concludes the findings of the research.
Chapter 2

Literature review

Plagiarism detection is of great interest to commercial and educational institutions. In spite of this, little research has been conducted into the design and development of large-scale plagiarism detection systems.

A practical solution to any real-world plagiarism detection problem follows from the selection of requirements, constraints and acceptable compromises. It is generally impossible to suit one solution to a wide spectrum of requirements and constraints, although it is certainly possible to share significant design and implementation artifacts between solutions.

As the aim of this research is to evaluate a number of techniques in terms of different realistic requirements and constraints, it is appropriate to explore the literature surrounding applicable problems and existing solutions.

The literature review will begin with the selection of a “type” of plagiarism detection, the most general constraint on solutions. From there, the selection of appropriate algorithms, data structures and system designs will follow, with relevant literature material to support selection and establish the contribution of this research. The review will conclude with the selection and definition of research subjects.

2.1 Automated plagiarism detection

Automated plagiarism detection is a new and immature field, with the first “plagiarism analysis” conference having been held in 2007, and the first open competition in 2009 (PAN09 (2009)). The technical problems faced by plagiarism detection system designers are extensive and varied, and at this early stage many rapid improvements are being made. However, available literature on automated plagiarism detection techniques remains scarce.

Plagiarism detection is a special case of a general field of document similarity comparison. Similarity comparison is often used to select related and relevant documents given known documents as input, and plagiarism detection constrains the similarity query specifically to detect the unattributed duplication of creative work (PAN09 (2009)).

The actual definition of plagiarism is contentious at best. In the context of higher education, plagiarism is the representation of third-party work as if it was one’s own (Squire (2006)). A simple guideline given by Monash University is that “more than five words taken directly” (Hurst and Eaves (1998)) without citation is an infringement. More generally, statements duplicated without attribution, even if slightly paraphrased, can be considered plagiarism.

The standards vary between educational institutions, and as represented in the copyright law of individual states and nations, though specific policy details are not especially important to the technical problems of detection. Instead policies set requirements that detection systems can help to enforce.
Plagiarism is committed in many ways, only some of which are practically detectable by algorithmic means (Wise (1996); Prechelt et al. (2000); Bernstein and Zobel (2004)). The types of plagiarism to be detected require varied algorithmic approaches, with different constraints. A few common types will be discussed in this section.

2.1.1 Substring matching

The most basic and general form of plagiarism detection confirms that sections of text are duplicated across documents, with some leniency towards formatting and representation of terms (Wise (1996)). The purpose of the search is to quickly and precisely identify where possible plagiarism has occurred, so that a human can perform a simple comparison in context and determine if the duplication is grounds for disciplinary or legal action.

Exact substring matching is, by definition, unable to detect simple cases of paraphrasing (Bernstein and Zobel (2004)). This is difficult to calibrate, as for short match lengths it gives many false positives, and for long match lengths it can be entirely subverted by a change to a single word. In practice it still detects very common cases of plagiarism (Wise (1996); Prechelt et al. (2000)), especially for student assignments where plagiarism is often committed specifically to minimise writing effort.

As the simplest detection technique, it is the easiest to optimise and verify, and thus forms the focus of experimentation in this project. Common techniques will be discussed in section 2.2.

Approximate substring matching techniques aim to detect permutated copies, but necessarily lead to far more false positives. These techniques are more difficult to optimise and verify, and thus are not the primary experimental focus of this research.

A straightforward technique for approximate string matching calculates the edit distance between pairs of substrings (Levenshtein (1966)). This can be done in a number of ways, most often with well-known Dynamic Programming implementations that have $O(nm)$ memory and runtime, entirely impractical for large-scale detection.

 Sophistication is necessary to narrow the scope of an approximate match so that the matched text is not lost among many edits, and this sophistication is in itself a difficult problem.

Bernstein and Zobel (2004) designed a robust system supporting a small scale matching scenario with generally acceptable limitations. The use of string chunk hashing allows substring pairs to be lined up efficiently, leading to a similarity measure based on matching chunks, though the memory and runtime consumption of the technique are still too high for a large scale matching system (Bernstein and Zobel (2004)).

2.1.2 Ranking and Fingerprinting

Classic Information Retrieval techniques such as document ranking and fingerprinting have been applied to derivative document detection (Hoad and Zobel (2003)). These techniques can be used to greatly narrow the scope of a substring search, also reducing the coverage of duplicates found (Bernstein and Zobel (2004)), or to produce an “similarity” report to suggest to human users which documents to compare manually. As string matching techniques are able to give far more reliable results with acceptable performance (Bernstein and Zobel (2004)), ranking and fingerprinting are excluded from this research.

2.1.3 Source code matching

The detection of software source code plagiarism is far more difficult than detection for natural languages, as duplicated program source can be made to appear independent just by renaming variables, changing comments, and altering structure, all without any change
to the semantics of the code (Ottenstein (1976); Parker and Hamblen (1989); Joy and Luck (1998); Ducasse et al. (1999); Prechelt et al. (2000)), and thus with arguably no original work having occurred.

Automating the detection process is more difficult still. Detection algorithms must make especially subjective assumptions on the relations between similar program code, without understanding the code itself (Ottenstein (1976); Parker and Hamblen (1989); Joy and Luck (1998)).

Ottenstein (1976) described one of the first flexible algorithms for identifying potential plagiarism in software source code, tolerant of several common plagiarist alterations such as symbol renaming, comment changes and structural changes. By calculating straightforward feature statistics regarding program source, the high-level similarity of programs can be compared with trivial arithmetic, suggesting to a human user where programs are unusually similar (Ottenstein (1976)). However, this technique is rather easily defeated by other plagiarist alterations such as adding redundant functions to skew the statistics without significant effort or understanding (Joy and Luck (1998)).

Parker and Hamblen (1989) extends the gathered feature statistics to improve robustness in the face of more advanced alterations, though the technique itself shares the same fundamental limitations.

Wise (1996) describes the YAP3 system, using an algorithm called “Running-Karp-Rabin Greedy-String-Tiling” (see section 2.2.3 for more on Karp-Rabin string search), which enables the detection of re-ordered program code fragments, common in plagiarised program source. Although it is effective for that limited class of detection, it can be defeated by some of the kinds of changes supported by Ottenstein (1976) and Parker and Hamblen (1989).

Prechelt et al. (2000), noting the inherent weakness of feature statistics techniques, describes an advanced technique incorporating abstract structural information extracted from program code. This algorithm tiles together substrings of pairs of programs with an advanced matching algorithm, producing an overall similarity metric which is very robust against many forms of alteration (Prechelt et al. (2000)). Although this technique has exceptional detection performance, its quadratic runtime complexity limits it to handling small sets of submissions, which is sufficient to detect whether one assignment in the set small duplicates another (Prechelt et al. (2000)), though insufficient for investigating whether one duplicates code from any of the many thousands of open source programs available freely on the internet.

The detection of source code plagiarism is important to both educational and commercial institutions, but because of its weaknesses and complexities, it is excluded from this research.

2.2 Exact substring matching

Simple substring matching, with one pattern string and a large string in which the pattern may exist, has many well optimised solutions for different requirements. A selection of common techniques will be described in this section.

The queries performed by plagiarism detection systems vary. It is generally sufficient to detect whether a substring is matched somewhere in the text corpus (Wise (1996)). However, to enable a fair and reasonable review of a plagiarism report, the source of the copied substring should be included so that the comparison can be made in context. Although an efficient set data structure is sufficient to detect that a given string exists (Karp and Rabin (1987)), more sophistication is necessary to enumerate the matching sources. So while most of these techniques support matches against a single large string,
it is necessary to match a set of strings (the text corpus), and still be able to recover the identity of each original text.

For the purpose of searches and matches, text can be represented in a number of normalised forms. A straightforward representation that is efficient to generate, store and process is an array of integers, with each integer uniquely representing a word. This can be taken a step further by uniquely identifying word stems (Porter (1980)), or even conflated concepts (Frakes (1984)), though such normalisations are outside the scope of this research. See section 3.1 for the chosen text representation.

The impact that text representation has on indexing structures is chiefly the size of the “alphabet” of the resulting data string, that is, the set of distinct symbols available for each discrete position in the string. For plain ASCII-encoded English the alphabet is small and constant, but a normalized form that represents unique words has a potentially unbounded alphabet, with many hundreds of thousands of words and terms appearing in any large corpus (Frakes (1984)). Advanced string algorithms and data structures are often divided between those that work for a fixed alphabet and those for an arbitrary alphabet (Na and Park (2007); Schürmann and Stoye (2007); Grossi and Vitter (2000)). Because solutions suitable for arbitrary alphabets are necessary to support natural languages such as Chinese, they form the focus of this research.

2.2.1 Suffix trees

A trivial data structure for linear-time string matching is a trie, a sparse data structure in which the root node links to each suffix, and every node’s children are indexed by the symbol next in the string. However, this representation is exceedingly expensive in memory.

Weiner (1973) introduced the suffix tree, a trie-like data structure that allows linear search for substrings of an arbitrarily large string, as well as fast solutions to a number of other string operations, yet much more memory-efficient than an equivalent trie. A suffix tree can be searched by traversing child nodes indexed by symbols in the substring, until the entire substring is matched, or a terminal symbol (nominally $, often actually a null terminator) is reached. See Figure 2.1.

![Suffix Tree Example](image)

Figure 2.1: Suffix Tree Example (not shown: sharing common nodes)

Suffix trees operate best on texts with fixed alphabets, as an array may be used to locate child nodes based on input symbols (Weiner (1973)). For an arbitrary alphabet, data structures such as balanced binary trees and hash tables can be used to follow suffixes (McCreight (1976); Tian et al. (2005)), introducing significant runtime overhead for construction and search.
2.2. EXACT SUBSTRING MATCHING

The suffix tree is of great importance to string matching theory, specifically because its match costs for a fixed-alphabet match are theoretically optimal — linear in runtime and constant in memory (Weiner (1973)). For the most common plagiarism detection case where a substring does not match, this is confirmed after reaching the first symbol that does not match (Weiner (1973)).

However, in practice the structure has significant weaknesses. Most critically, the memory consumption of a suffix tree is very high, generally several times as large as the string it represents (Tian et al. (2005)). Furthermore, as a tree structure, it exhibits very poor memory locality during construction and search, increasing the practical runtime overhead significantly (Tian et al. (2005)).

In a practical setting where the size of the database is significant and searches may take longer to complete, using a more memory-efficient data structure is often more appropriate, even though such a structure may exhibit worse search runtime (Manber and Myers (1990)). This is often the case with plagiarism detection, where the text corpus is very large and searches may be executed asynchronously in batches.

Weiner (1973) described the original construction algorithm, which read strings in reverse and thus moved existing nodes further from the root by adding prefixes. Although the resulting tree structure was efficiently searchable, its construction time was very poor, and reading the string in reverse made incremental growth impossible.

McCreight (1976) greatly improved the construction algorithm’s space complexity by constructing the tree as a series of in-place tree transformations until all desired properties held. This construction technique did not support online growth, but the same paper approached the problem of incremental update (in response to changes to the string) through efficient split/merge operations.

Ukkonen (1995) discovered the first linear-time online construction algorithm that allowed incremental growth, with applications to tries in general. The new algorithm constructed the tree by reading the string in order, efficiently appending to the tree and adding new suffix roots, both of which are constant-time operations through the use of suffix links (Ukkonen (1995)). This intuitive and understandable contribution encouraged the wider use and development of suffix tree techniques.

Tian et al. (2005) described a practical suffix tree construction algorithm with strong practical performance characteristics, and experimentally demonstrated that online construction with poor worst-case scalability but good memory locality outperforms scalable algorithms (such as Ukkonen (1995)) that have poor locality. This construction algorithm is on average more efficient than that of Ukkonen (1995) in practice (Tian et al. (2005)), so for a plagiarism detection system where runtime efficiency is critical, this technique is clearly preferable.

However, the memory costs of suffix trees are still significant to the extent that they are not commonly used for large volumes of data, where a less scalable but more memory-efficient structure would make better use of high-speed memory and cache locality. Thus suffix trees are not the focus of this research.

2.2.2 Suffix arrays

Manber and Myers (1990) introduced the suffix array, a data structure conceptually similar to a suffix tree, but using far less memory. A suffix array is simply a list of positions in the string, ordered lexicographically by the suffix beginning at each position. As it does not store addresses within itself, it occupies far less memory than an equivalent suffix tree, but is not as efficient for most queries (Manber and Myers (1990)).

This ordering property allows substrings to be matched using a binary search, reducing the time complexity of a single substring search to $O(\log n)$ suffix comparisons. However, as each comparison may require the entire string to be checked, the overall runtime is

CHAPTER 2. LITERATURE REVIEW

Figure 2.2: A simple string and its suffix array

$O(m \log n)$ (Manber and Myers (1990)). This is not common in practice, as the “common prefixes” of suffixes of real text are rather short, and for texts with long common prefixes, a generated table can be used to reduce the runtime complexity towards $O(m + \log n)$ at some cost to memory (Schürmann and Stoye (2007)).

Although it is not efficient to grow a suffix array incrementally, it is simple to merge two complete suffix arrays in $O(n + m)$ comparisons, each of which can be made near-constant using the same common prefix approach as in construction and search. In this way, online growth can be simulated in batches. In a practical implementation, this merge will likely take twice as much memory as the sum of the individual arrays, though compromises can be made to reduce this peak cost.

More recently, the suffix array has been evolved into a more advanced data structure, the compressed suffix array (Grossi and Vitter (2000)). CSAs factor out encoding redundancy to reduce the size of the structure, while still allowing reasonably efficient search. A number of advancements in the construction of CSAs have been developed, chiefly Sadakane (2000), Hon et al. (2003), Kärkkäinen et al. (2006), and Na and Park (2007). However, CSAs have not seen much experimental use with natural language texts, and such an extensive experiment is outside the scope of this research.

2.2.3 Karp-Rabin string search

Karp and Rabin (1987) describes the string search algorithm using hashes to greatly reduce the scope of a symbol-by-symbol string comparison. Although its worst case performance to match a single string is poor, it can be generalised via a fast set structure to support efficient many-to-many string matches (Karp and Rabin (1987)) as is necessary for plagiarism detection. The core concepts of this technique form part of the experimental focus of this research.

Matching many substrings of uniform length can be achieved with a set data structure and a simple algorithm. Given a needle length $m$ and a haystack length $n$, check whether any $m$-length substring of the haystack matches a needle in the set of needles. See Figure 2.3. Note that reversing the get and put gives equivalent results, so the direction only determines the memory used.

An efficient set structure for many arbitrary strings is a hash table, but for smaller sets it is likely more efficient to use a balanced binary tree and thus avoid calculating hashes.

With an efficient hash table structure that has $O(1)$ insertion and lookup, and a balanced hash, the algorithm is expected to run in $O(n + m)$ time (Karp and Rabin (1987)). The worst case is that all needles must be checked because their hashes collide, and this results in a poor $O(nm)$ runtime, but with an appropriate hash function this is extremely unlikely to occur in practice (Karp and Rabin (1987)).

This technique can be generalised to operate on any representation of a substring, such as an n-gram. The selection of hash functions, set structures, text representations and detection workflow integrations is a large problem, partly addressed by this research.

<table>
<thead>
<tr>
<th>$i$</th>
<th>$T_i$</th>
<th>$i$</th>
<th>$SA[i]$</th>
<th>$T_{SA[i]}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>banana</td>
<td>0</td>
<td>a</td>
<td>a$</td>
</tr>
<tr>
<td>1</td>
<td>anana</td>
<td>1</td>
<td>ana$</td>
<td>a$</td>
</tr>
<tr>
<td>2</td>
<td>nana</td>
<td>2</td>
<td>anana$</td>
<td>a$</td>
</tr>
<tr>
<td>3</td>
<td>ana</td>
<td>3</td>
<td>banana$</td>
<td>a$</td>
</tr>
<tr>
<td>4</td>
<td>na$</td>
<td>4</td>
<td>na$</td>
<td>a$</td>
</tr>
<tr>
<td>5</td>
<td>a$</td>
<td>5</td>
<td>nana$</td>
<td>a$</td>
</tr>
</tbody>
</table>
2.3 Distributed processing and storage

Distributed processing (aka. Cluster computing, grid computing) and, to a lesser degree, distributed storage, have applications to almost any high-volume computing problem. Distributed processing in particular is perfectly suited to “embarrassingly parallel” problems such as plagiarism detection.

Most commonly, distributed storage is implemented as a necessary part of a distributed processing system, where parts of a data set are split among nodes, with redundant copies enabling fault tolerance in the event that a node becomes unavailable. Ramabhadran and Pasquale (2008) explores probabilistic measures of data lifetime as related to storage and redundancy factors, aiding cost-benefit calculations for redundant and distributed storage.

For plagiarism detection, it appears best to divide the (semi-)permanent text data set among computing nodes each with their own reliable storage and redundant copies. See Figure 2.4. Note that although each member of a redundant pair stores the same data, each member can search different parts of that data. So while the storage is redundant, the processing is not.

The precise nature of the communication and storage required depends upon detection requirements and constraints, and this project aims to explore general guidelines and compromises for the construction and configuration of a distributed plagiarism detection system.
A number of distributed storage designs have been developed to meet these needs, with various qualities. Louis and Aravindan (2008) briefly describes a scalable and fault-tolerant system that reclaims unused disk space on heterogenous networked workstations, though not to utilise their computational resources. In contrast to conventional “reactive” redundancy measures, which replicate data fragments when their redundancy level falls, Duminnco et al. (2007) describes a “proactive” approach which can raise redundancy levels as affordable in anticipation of a fault.

Distributed processing and storage are not significant subjects of this research, however, elements of their concepts will feature in the implemented apparatus. Efficient distribution of data and work complements the efficient storage and processing of each work node.

2.4 Conclusion

Plagiarism detection is an important and challenging problem, with a variety of requirements and appropriate solutions. Practical solutions vary depending on requirements and constraints.

There is little work on the design and implementation of scalable, efficient plagiarism detection systems. It is important to contribute experiments and designs to the field, so that the next generation of detection systems can meet ever-increasing demands.

For this research, exact substring matching has been chosen as the type of plagiarism detection, so Suffix Arrays (section 2.2.2) and Karp-Rabin String Search (section 2.2.3) have been selected as experimental primitives for research. Suffix trees (section 2.2.1) are noted as a solution with excellent runtime but often unacceptable memory consumption.

Suffix arrays offer efficient construction and search, though they lack efficient online growth. Once constructed, they occupy reasonably little memory, all of which is contiguous.

Karp-Rabin string search uses hash tables to efficiently narrow string comparisons in a string match. Realistic hash function performance makes this technique very attractive in terms of runtime, and the generality and flexibility enable a number of applications.

The scalability of a detection system is critical to sustainable use, and distributed processing is a practical and mature solution to system scalability, with distributed storage offering fault tolerance and high throughput for a detection database. Advanced distributed processing and storage techniques are excluded from this research, but a general framework with primitive features will be integrated into the experimental apparatus, allowing it to be highly effective in practice.

The significant problem of educational and commercial plagiarism can be managed through a combination of administrative policies, human effort and machine assistance. This research will further the growth of machine assistance technology for the purpose of plagiarism detection.
Chapter 3

Detection primitives

The basic work unit of any duplication detection system is a detection primitive that enumerates matches for a combination of candidate and corpus texts. Corpus texts are those archived as potential sources for duplication. Candidate texts are those being searched live for potentially duplicated text.

The design and implementation of a detection primitive necessarily influences its performance characteristics and thus its suitability to certain uses. In this chapter, several aspects of detection primitives, and specific detection primitives themselves, will be defined and justified.

3.1 Text representation

It was explained in section 2.2 that the chosen text representation is an array of integers representing unique words. This has strengths and weaknesses that will now be discussed.

3.1.1 Efficiency

Once generated, texts represented in this way are often more efficient in memory and may require fewer CPU instructions to process. Even arbitrarily long words are encoded as a bit string sufficient to identify them uniquely, and for a restricted set of English words, this bit string may be even fewer than 20 bits wide. Entire word comparisons can thus be performed with a single integer equality comparison, one of the most efficient CPU instructions across all modern instruction set architectures.

3.1.2 Language neutrality

Any practical duplication detection system must be usable over a broad range of natural and computer languages. The word integer representation is certainly not always optimal or even suitable for a given language or query, but with an appropriate definition and resolution of a ‘word’, it is especially flexible and robust. Expand

For Chinese language texts, it is in fact possible to use a complete Unicode integer directly, as these are generally already at an appropriate resolution. It is still necessary to remove extraneous non-character symbols from the text when importing it.

As the environment in which this research has been performed is rich with English texts, those are the experimental focus, though it will be made clear that no element of the algorithms and data structures relies on any property of English text or its encoding.
3.1.3 Importability

There is no natural mapping between words and integers, so any transformation process is either artificially constrained or difficult to reproduce. An example of an artificially constrained method is a prepared mapping table, which necessarily cannot support words not included in the table. It is simple to use an algorithm which selects new integers for words when they are first encountered, using a simple counter, but the association then depends entirely on the order in which the words occur, and the order in which the texts are transformed.

3.1.4 Encoding technique

The very simple encoding algorithm implemented in this project’s experimental apparatus recognises only contiguous substrings of letters as “words”. As implemented, this only accepts the 26 core letters in the Latin alphabet, and any other byte is equivalent to a separator.

Only words of 4 letters or longer (i.e., \( \geq 4 \) consecutive letter bytes) are encoded. This is largely because repeated sequences of short words are not at all indicative of plagiarism, and thus long words must be the focus of match detection. Additionally, requiring 4-letter words largely avoids pollution from poorly extracted plaintext, where many single letters may be scattered throughout binary or markup data.

Word integers are assigned consecutively from 0, in the order of first appearance. This has a significant limitation in that the word-integer association database is entirely dependent upon the order and content of texts introduced to the system, making any encoded text incompatible with virtually any other system. There are many alternative solutions, but all with their own shortcomings — for example, using a hash function for word identification introduces the risk of collision. If the set of expected words can be completely enumerated and never changed, a perfect hash function or prepared associative table can avoid collisions while remaining portable though inflexible.

\[
\text{const char str1[]} = \text{"This is a short sentence.";}
\]
\[
\text{const int words1[]} = \begin{bmatrix} w0 & w1 & w2 \end{bmatrix}
\]
\[
\text{const char str2[]} = \text{"Another sentence.";}
\]
\[
\text{const int words2[]} = \begin{bmatrix} w3 & w2 \end{bmatrix}
\]

Figure 3.1: String encoding example

3.2 \( n \)-gram hashing

Several of the experimental primitives featured in this research rely on calculating hashes for fixed-size substrings of the texts. The hashes themselves serve to indicate where possible matches occur, and direct an algorithmic search to confirm those matches in isolation.

In the experimental software, hashing is very simply performed by passing a standard hash function (such as \text{crc32} provided by \text{zlib}) over the raw integer sub-array memory. Although this hash is not portable to platforms of different endianness, nor to the same words represented by different unique integers, the hashes themselves are not stored or transmitted anyway, and are only retained as part of an index structure resident only in random-access memory.
It is critical that any chosen algorithm is able to ensure a good random distribution of hashes even for a short string, such as the very common \( n \)-gram configuration consisting of 5 integers of \( \leq 32 \) bits each. CRC32 exhibits this quality, so it is recommended as a safe choice for general use.

It is assumed that the performance of any common hash function will not significantly impact overall search performance. In the worst case, where a hash-based matcher is empty, any candidate text hashing will be wasted, but still consume only a small amount of CPU time. Even such a pathological case can be treated specially by an implementation.

### 3.3 Generalised \( n \)-gram hash table

A general template for copy search using \( n \)-gram hashes is described here. This template is specialised in different ways later, offering many optimisation opportunities.

The basic point of any such data structure is to enable a search to efficiently identify where \( n \)-grams match between texts, so that a more comprehensive comparison can then be performed only on very few specific candidates with a high chance of success. It is critical that the structure retain sufficient data to ensure that no actual match is ignored, though it is acceptable to produce many false positives if they can be efficiently eliminated during the search. The details of the search algorithm, and how it interacts with specialised structures, will be discussed further below.

Abstractly, any search for a specific \( n \)-gram must return a set of \((text, offset)\) pairs (hereafter \((T, O)\)) that potentially contain matching \( n \)-grams. To be correct, this must be a superset of the set of actual matches, but any other contents are a matter of optimisation choices.

#### 3.3.1 Structure

The structure of interest is, at its most abstract level, a hash table using dynamic arrays for each hash bucket. The structure of the entries is the key point of variation in the specialised structures, and the overall data structure itself may be required to store other sub-structures necessary to efficiently store, grow or search the hash table.

In the system implemented, hash buckets are selected by the lowest \( k \) bits of each \( n \)-gram’s hash. This \( k \) depends on the “table size” parameter. Any part of the hash may be used by the table elements, and this affords further opportunities for optimisation.

#### 3.3.2 Table elements

As the purpose of the table is to identify all potential \( n \)-gram matches, each element in the table must reference an \( n \)-gram in the text archive with an appropriate resolution. The search algorithm may be generalised to test multiple corpus \( n \)-grams if an entry does not precisely specify a single \( n \)-gram.

The specialised versions of the table structure mostly differ in the contents of their table elements, and the supporting sub-structures that are required to translate the table elements into correct \((T, O)\) pairs.
3.3.3 Abstract algorithms

Each specialisation has its own slightly different growth and search algorithms, but each conforms to a simple abstract template. The algorithms are given here in a simple Python-like pseudocode.

```python
def build(buckets, text):
    for ngram in text:
        sum = crc32(ngram)
        bucket = sum mod buckets.size
        buckets[bucket].append(ngram)

def search(buckets, text):
    for ngram1 in text:
        sum = crc32(ngram1)
        bucket = sum mod buckets.size
        for ngram2 in buckets[bucket]:
            trymatch(ngram1, ngram2)
```

The procedure `trymatch` may be defined as required for the query being performed. This may record matches, gather statistics, etc. as required by the problem. A higher-level longest-copy search algorithm compatible with most match primitives is defined in section 5.1.2 — in implementations, the longest-copy algorithm may be integrated with the hash table search algorithm, and this is the primary design for this project’s experimental apparatus.

3.3.4 Filtered indexing

If assumptions may be made about the possible contents of the candidate set, the index structure — and indeed almost any substring search structure — may be optimised during construction to reduce the runtime and memory costs to be incurred during search.

A very specific and simple form of structure optimisation is entry filtering. If the set of $n$-grams in the candidate text set is known, the corpus index can be constructed listing only (potential) matches for those $n$-grams. This is more economical, though less thorough as a filter, if the set is composed only of $n$-gram hashes rather than full $n$-grams.

This usage is atypical, as normally candidate sets are introduced to a pre-constructed corpus index. However, for batch detection workflows where indices must be constructed only to match a specific candidate set, this simple technique can drastically reduce both memory usage and runtime (see section 4.4 for techniques and section 5.5.2 for performance measurements).

3.3.5 Partial indexing

A useful generalisation of $n$-gram-based match search is to skip periodic $n$-grams while retaining at least one $n$-gram per substring of minimal match length. The most straightforward $n$-gram indexing design sets the $n$-gram size to the minimal match length, and thus an entire $n$-gram must match to be included in the result set. More generally, to match a substring of size $m$, all substrings of size $\leq m$ starting at equivalent positions must also match.

As false positives are eliminated via deep search anyway, it is only necessary to store one substring of size $\leq m$ in order to narrow the scope for deep search, and any matches with a total length $< m$ are simply discarded as usual. This property enables essentially arbitrary reductions in the $n$-gram retention rate of an index structure, as every $n$-gram but one may be omitted from a string of length $> n$. 
For example, if the minimum match length is 5, the following configurations are correct:

- Store 5-gram (100% memory)
- Store 4-gram, ignore other 1 4-gram (50% memory)
- Store 3-gram, ignore other 2 3-grams (33% memory)
- Store 2-gram, ignore other 3 2-grams (25% memory)
- Store 1-gram, ignore other 4 1-grams (20% memory)

The value of this memory reduction decreases significantly with each reduction in \( n \)-gram size. In section 5.2.4 it is shown that runtime efficiency suffers notably from \( n = 5 \) to \( n = 4 \) and far more to \( n = 3 \).

This technique clearly has potential to reduce index memory consumption, with what may be an acceptable runtime cost. If the memory is available, it should of course be used effectively, but if memory is only available for a partial index of the corpus, this technique allows the entire corpus to be searched within memory and thus without resorting to more expensive search techniques such as disk-backed batch search (see section 4.4).

3.4 **Index — CPU–efficient \( n \)-gram hash table**

The most rudimentary realisation of the abstract table stores the text ID and \( n \)-gram offset directly in each element, minimising the CPU instructions necessary to enumerate the candidate \( n \)-grams. The text ID may be a pointer to a text structure (as appropriate in the implementation language), or indirected through an array of pointers — the latter is clearly more efficient as the set of possible IDs is necessarily smaller than the set of possible pointers, and thus fewer bits may be used per table entry to identify a text.

This structure is correct and sufficient (see section 5.2.2). Although in theory it is efficient in CPU instructions, in practice — where false positives must be filtered by retrieving the \( n \)-gram’s contents and comparing them word-for-word — it performs many memory indirections that may incur a high access latency and interfere with cache performance. It is very helpful to be able to encode more information in each table element so that false positives can be eliminated with fewer memory accesses in the average case.

3.4.1 **Hash filtering**

A very straightforward technique is to store, in each table entry, some high bits of the relevant \( n \)-gram’s hash (recall that the low bits already match due to the bucket selection). These can be compared with a trivial integer comparison instruction, with no extra memory retrieval count per element (but several bits more retrieved). The entry tuple is then \((T, O, H)\).

Although this filter requires an extra comparison and branch during entry iteration, it often avoids more expensive retrievals and comparisons, so if false positives are common, this technique offers a net saving in runtime. The precise implementation can be well calibrated; storing more hash bits reduces the probability that a deep search will be required, but increases memory consumption.

3.4.2 **Element dimensions**

A naïve specification for the \((T, O)\) pairs would use the smallest integer large enough to fit at minimum any text identifier or offset encountered in the corpus. In such a design, even though the average document has only thousands of words, it would be represented
by integers capable of counting millions of words, if such a document occurs in the corpus at least once. This is clearly inefficient.

Assuming a larger archive, and the hash optimisation, suppose that 64 bits are used per \((T, O, H)\) entry. This specification is termed \textit{Index}. This straightforward design is already practical for production use (see section 5.2.4). However, significant improvements to memory consumption are possible, with only modest sacrifices of CPU efficiency.

\section{Mindex — CPU– and memory–efficient \(n\)-gram hash table}

Several simple encoding techniques are applicable to reduce memory consumption while retaining CPU efficiency.

\subsection{Offset encoding}

To reduce the non-hash total from 40 bits to approximately 30 bits, it is necessary to reduce the size of the offset integer below what would be sufficient to represent all offsets \textit{within} a text. It is thus necessary to create an extra level of indirection between the real text identifier and the identifier stored in each reference. A table of “subtexts” stores tuples of the form \((text, offset)\), much like elements of the naïve table, and each \(n\)-gram reference tuple refers to a subtext by an integer code.

Each subtext represents a section of a text containing only thousands of words, with the offset identifying where in the original text this section begins. Thus the \(n\)-gram reference can use a very small integer (approximately 12 bits) for its word offset, and an integer for the subtext ID that is only slightly larger than the original text ID integer. There is likely to be an optimal balance between subtext ID size and offset size, and it appears to favor offset sizes that are just large enough to fit the average number of words per full text, though this merits experimental calibration.

The subtext integer is necessarily larger than the original text integer, but the premium is less than the saving in the offset integer. Thus the overall reference is far smaller than the naïve implementation, with no loss of precision or generality. The parameters of this optimisation depend upon the distribution of text sizes, so in production use it is valuable to confirm useful values experimentally.

\subsection{Hash filter}

This offset encoding is memory efficient, but it still stands to benefit from the hash comparison optimisation suggested for the CPU-efficient table. This may still be kept as high bits alongside the \((\text{subtext}, offset)\) tuple, but a simple technique has been developed to retain most of the benefits of the hash using no additional memory.

The order in which texts are added to the table is also the order references to them are added to the subtext table, thus the subtext codes within a table bucket are already in a natural integer order. Similarly, the offsets within a given text also follow a natural order, though this only applies if an \(n\)-gram appeared more than once in the same subtext.

During construction and search, the hash of an \(n\)-gram is known, and it is already used to select the bucket. It is possible to store the \(n\)-gram reference tuple as the bitwise exclusive-or (XOR) of the original tuple and the high bits of the hash. During search, the exclusive-or can be reversed only for a \textit{matching} hash, and for any other hash it will give an incorrect tuple.

By checking the ordering properties of the tuple components, it is trivial to detect when an incorrect hash has been used, with a small probability of false positives. False
positives only affect further runtime checks, not correctness, as the full n-gram contents of a false positive will still be confirmed. When a match is confirmed the now known-good subtext code can be recorded and used to enforce further order. This hash mixing technique allows a significant reduction of overall CPU effort with no extra memory.

If the hash is 32 bits (true for CRC32), the XOR of the tuple and the hash may also be stored in 32 bits. This has significant benefits. Except in a very full table, the high bits of the subtext code will be mostly 0s, so that when they are XORed with an incorrect hash, they will become non-zeroes and the subtext code will almost certainly exceed the range of subtext codes already created. This requires only a trivial integer comparison, and eliminates the vast majority of false positives. Unlike a straightforward bitwise tuple packing representation (as in the CPU-efficient design above), this gracefully degenerates as the table fills, offering slightly less elimination of false positives, but requiring no greater memory usage.

3.5.3 Further encoding for memory

Alternatively, the ordering property can be used to implement delta encoding of the subtext code and offset. In this scheme, each array element stores the difference since the previous subtext code, which takes far fewer bits per tuple than storing the absolute code. For the rare case that the difference exceeds the integer size, it can be accumulated through multiple references, with a sentinel value for the first reference (or the high bits of the first reference).

However, because the ordering in each individual element can no longer be maintained, the hash mixing technique above cannot be applied to the subtext code. It can still be applied to the order and range of offsets within a subtext, though in practice this is far less effective at reducing runtime costs.

In the original 5GB text database experiment, this delta encoding reduced the size per reference to a mere 20 bits, at a notable but potentially acceptable penalty to search runtime. Each additional bit stored reduces the runtime significantly, so in production use, it is important to determine experimentally the optimal size given memory and computational resources.

It is alternatively possible to store only a single unique integer representing a unique n-gram in the whole table’s text list — essentially a packed pointer. This requires only constant time during construction, but has penalties during search as it may take far longer to decode such an integer than one which is separated into (sub)text and offset. The combined integer may also be subjected to delta encoding (or to hash mixing, if CPU is more valuable than memory), and a skip table may reduce the worst case costs of the integer interpretation at a small cost to memory.

3.5.4 Implemented specification

All techniques described here have been tested experimentally, and others have been tested as well but found lacking in utility. The delta encoding and packed pointer techniques were found to have an unjustifiable runtime penalty, while saving mere bits per entry.

The implemented Mindex supports filtered indexing, partial indexing, offset encoding and XOR’d hash filtering. It is integrated with the aggressive search algorithm specified in section 5.1.2. Each n-gram entry is specified to be exactly 32 bits in size, affording reductions in CPU time compared to arbitrarily-sized entries. The low 14 bits are reserved for the offset element, leaving 18 bits for the subtext element. All 32 bits are XOR’d with the hash, as there is no need to extract only high bits since they must all match for a correct substring match.

See sections 5.2.4 and 5.3 for a performance evaluation of the Mindex.
3.6  *Dexin* — Inverted index for scalable batch search

For searches that merely enumerate all combinations of matching substrings, searches are *symmetrical* — that is, a candidate set can be searched against the indexed corpus and give the same results as if the corpus was searched against the candidate set. The same is not true for more advanced searches such as the aggressive longest-copy algorithm described in section 5.1.2.

Similar to how batch search can be made more efficient with filtered indexing (see section 3.3.4), it can also be made potentially far more efficient with inverted indexing. An inverted index is that one is designed specifically to index a candidate set, and receive corpus chunk sets to retain the longest copies from the perspective of the candidate set.

Instead of creating a complete index of every chunk, and repeatedly searching the candidate set against it, an index is created of the candidate set only once, and each chunk is iterated against that index. It is immediately clear that this is likely to be considerably more efficient and scalable.

### 3.6.1 Data structure

This design results in the experimental *Dexin* primitive. The primary difference between this structure and the *Mindex* is that each entry stores not only an *n*-gram reference but also a record of the longest match yet encountered starting from that *n*-gram.

Using the same *n*-gram reference encoding techniques as in section 3.5.1, the additional match reference only doubles the total size of the entry — however, so that the length can be recovered without retaining and re-searching the reference’s text data, it can also be encoded into the entry directly.

As each *n*-gram reference may be easily packed into less than 32 bits, 64 bits is likely sufficient for two such references and a short match length count. This must be calibrated against the expected dimensions of each text set, and the expected maximum match length.

### 3.6.2 Result collection

After a *Dexin* is populated with match records, the records must be collected in a manner similar to the search algorithm specified in section 5.1.2. As no more full texts must be scanned, the collect stage merely takes note of the longest copy already recorded for each candidate *n*-gram, skipping irrelevant copies as typical of the aggressive algorithm.

In section 5.2.2 it is experimentally verified that this is consistent with the non-inverted search design typical of *Index* and *Mindex*.

### 3.6.3 Implemented specification

In the experimental system implemented, 96 bits are used — 32 for each *n*-gram reference, and 32 for the match length count — as this is simple to implement and very efficient in CPU time.

As the *Dexin* is intended to be used for *candidate* text sets, it is not necessary for it to be as memory-efficient as a *Mindex* which is expected to store thousands of times more *n*-gram references.

See sections 5.2.4 and 5.5 for a performance evaluation of the *Dexin*. 
3.7 **Bindex** — Suffix array for archive-only binary search

As a point of performance comparison for the experimental primitives above, **Bindex** has been implemented as a simple suffix array supporting the same search specification as the other primitives. **Bindex** does not follow the state of the art in suffix array optimisation; it is only intended to compare the use of binary search against hash table search.

Note that while most suffix array implementations search for a single element, this is not the case with the search specification for **Bindex**. The binary search isolates one adequate match in the data structure, then iterates in both directions of the array until matches are no longer adequate. This ensures that the linear component of the search tests only adequate matches (with the exception of the first one found inadequate in each direction), and the binary component retains its classical logarithmic runtime.

**Bindex** contains no common prefix cache specifically because of the internal text representation (see section 3.1). A suffix array operating on a letter-by-letter encoding of natural language text benefits greatly from a prefix cache (Schürmann and Stoye (2007)), however, it is clear that word-level string resolution has a much lower average common prefix length. For use on such short-prefixed representations, a prefix cache will cost memory and runtime without significant returns.

As implemented, **Bindex** shares the same offset encoding technique as **Mindex**. However, because binary search must be performed, XOR hash filtering is incompatible, and high-bit hash inclusion (as in **Index**) has not been implemented.

Prototypes have been tested with a hybrid data structure that is a hash table with every bucket’s entries sorted. This reduces the binary component of the search (because hash tables narrow the candidate array) but requires hashing runtime and table memory overheads comparable to that of a pure hash table structure such as **Mindex**.

3.8 **Basic** — Reference brute–force search algorithm

As a reference for correctness verification only, a **Basic** primitive has been implemented. It has no optimisations of any kind, even the most trivial. Every substring in the candidate set is searched against every substring in the corpus set, using the aggressive search specification in section 5.1.2. The only persistent data structure is a simple list of every text considered to be included in the corpus.

3.9 Conclusion

Several primitive design and implementation techniques have been discussed, intended for use at the core of an efficient duplication detection system. **Mindex** is positioned as an efficient primitive for interactive (in-memory) search. **Dexin** is positioned as an efficient primitive for batch (disk-backed) search.

In chapter 4 these primitives will be integrated into detection techniques, and in chapter 5 the primitives and techniques will be subjected to performance evaluation.
Chapter 4

High-level techniques

Having defined detection primitives, several high-level techniques and designs must now be defined in order to appropriate the primitives to real applications and requirements.

This chapter will define and justify several high-level techniques that cooperate to enable efficient duplication detection in a variety of scenarios.

4.1 Text identification using content checksums

For the purpose of duplication detection in general, it is not necessary to uniquely identify corpus texts. In practical settings, however, many optimisations (both to the system and to user interactions) are possible if texts may be uniquely identified by some token.

The choice of an identification technique necessarily depends on the definition of identity. Even for something as simple as a plaintext, any number of identification methods are possible, suitable to different purposes. For duplication detection uses such as plagiarism detection, the name of a text (if any) is irrelevant, as long as it is clear from the content whether a corpus text is historically distinct from the candidate text it supposedly matches — i.e., whether it is supposed to be partly duplicated or not.

For this reason, a very conservative identification technique is used to identify texts in this system. Specifically, texts are identified by a checksum of their contents, as in several other document management systems, including plagiarism detection systems (Miller et al. (2000)). The checksum algorithm must be chosen based on acceptable trade-offs of computation time, collision probability, and checksum memory size — for the system tested in this project, MD5 is used, applied directly to the raw byte string stored in the text’s file. Although MD5 is no longer a cryptographically secure algorithm, it is sufficient for experimental use.

4.1.1 Redundancy considerations

If two versions of a text are present in the system, differing by even a single byte, they will receive two different checksums and thus be identified as separate. This does not affect search coverage, as all versions will be checked as potential matches for a candidate text, though it is considerably inefficient to store redundant text matter. There is a wealth of research literature on partial de-duplication, though it is not studied or applied in this project.

Conversely, if the exact same text is entered into the system from multiple sources, it will not be stored or searched in duplicate. Depending on the design and operation of the detection system, this may significantly reduce the redundant load on the system.
4.1.2 Verification, security

The use of checksums as an identification also serves as a form of integrity verification. If a text’s content is modified in any way, its checksum will no longer match, and it can be marked as corrupted and excluded from searches. For texts stored as flat files, any checksum tool can be used to verify the entire file database, without requiring any software specific to the project.

A security consideration is that even a novice attacker can construct a hash collision for a weak hash such as MD5, submitting a version of a text which will reduce to the same checksum as another text, in order to prevent the second text from being included in the database.

For example, a lazy student may copy an entire paragraph from Wikipedia, which would be caught instantly; if the same student pre-submits a text with a hash collision against that same Wikipedia page, that version of the Wikipedia page will never be included in the corpus, and later versions may be included too late for the student to be caught.

This is very easily addressed in either of two ways. Rather simply, a strong hash function can be used with an especially low chance of collision, such as SHA-512. This places collisions outside the reach of even well motivated and funded attackers, although the size of each stored hash becomes rather large.

Offering greater security with slightly greater complexity, HMAC checksumming can be used (Krawczyk et al. (1997)). To have any chance of generating a correct checksum, the system and attackers must have a secret pre-shared key that is stored only on the detection system’s server. Although this is known to be secure, and does not increase the size of stored hashes themselves, it relies entirely on the secrecy of the pre-shared key.

4.2 Text batching

The experimental apparatus is designed and implemented at all levels with batch processing in mind. In the worst case, where a batch contains only a single text, the overhead is still trivial — in general, enough to identify that it is only one text.

For the far more common cases where batches contain many texts, management and processing operations are simplified to operating on a whole batch at once, reducing many overheads. This further applies to storage and networking, cooperating with existing buffering systems to ensure that latencies are suffered only once per batch and not once per text.

Virtually all high-level techniques described in this chapter rely upon text batching to achieve efficiency while remaining simple in design and implementation.

4.3 Threaded match search

As search problems in general are highly parallel, a threaded search engine can reduce the CPU runtime component of system resource costs. In practice, however, several design and implementation details are critical to the efficiency of a threaded search engine.

4.3.1 Domain partitioning considerations

For any parallelism effort, a domain partitioning strategy must be chosen. It is not necessary to parallelise every possible operation, and so doing greatly complicates software verification and maintenance. It is best to only parallelise only the few execution bottlenecks encountered in production. Batch match search is an obvious first choice, as it is almost certain to dominate CPU consumption.
If a batch is partitioned at the text level, it is clear that a small batch may be unable to occupy all available processing units — for example, a batch with 3 texts cannot be run on 4 processors concurrently. Below the text level, higher concurrency is possible, but design and implementation are greatly complicated.

In practice, text-level partitioning is adequate. The search time for a single text of realistic length is never long, so it is essentially necessary to aggregate texts into batches in order to benefit from parallelism. Furthermore, in practice, if individual texts are being submitted for matching, it would be necessary for many users to submit individual texts at the same time in order to load the system to the point that parallelism is beneficial. In this case, single-text batches can be searched in parallel, even though they do not belong to the same batch.

For these reasons, text-level partitioning has been selected as a practical compromise.

### 4.3.2 Work imbalance

A significant real-world problem experienced by parallel systems is the effect of arbitrary work unit sizes on concurrency. Even if work units are assigned to processing units uniformly, if the runtime cost of the work units is not uniform, the runtime of each processing unit is most likely not uniform (Arora et al. (2001)). In such common cases, optimal concurrency is not achieved, because some processing units continue to work after others have completed their work.

Several parallel processing design and implementation techniques have arisen to address this problem. A modern and very effective solution is work stealing, where processing units automatically re-balance allocated work to maximise concurrency (Arora et al. (2001); Agrawal et al. (2008)).
In this balanced case, although each processing unit is unaware of how long a work unit will take, it will pull from work queues whenever it is free. So if one processing unit repeatedly receives small work units, it can remain busy while another is occupied with a large work unit.

Once again, domain partitioning resolution interacts with concurrency. If work unit resolution is very fine, it can be balanced effectively, but the overhead to schedule and rebalance it is high. If work unit resolution is coarse, the overheads are low, but the worst cases for balance may be significantly worse.

In the experimental apparatus as implemented to date, work stealing is not implemented; only a primitive but adequate form of work pull thread pooling is implemented, shown to give good scalability in practice (see section 5.4). Severe work imbalance has not been observed experimentally, and to solve minor imbalance, a trivial technique is applied.

### 4.3.3 Over-partitioning

The naïve domain partitioning approach is to give each processing unit a seemingly equal share of the work, and never re-partition or re-balance. A very simple sophistication of this approach gives far better practical performance (see section 5.4).

The problem with straight partitioning as described is that some sub-batches may require significantly more work than others, leading to the imbalance problem above. While work stealing can minimise imbalance overall, sufficient improvements can be achieved simply by increasing the resolution of partitioning without necessarily increasing the level, as long as work is taken rather than given.

In the case of text batches, without splitting up a single text between threads, the system can still divide a batch into more sub-batches than available processing units. Instead of pushing work to processing units, processing units request individual sub-batches only when they have completed their previous work. The combination of these two features tends to balance work automatically, by reducing the likely maximum work unit size and increasing the concurrency by aggressively distributing remaining work.

In the original best case where all work is uniform, it will still process uniformly; in the worst case where a single sub-batch is significantly longer than the others, it will occupy a single processing unit while others will aggregate multiple smaller sub-batches in order to remain occupied.
4.4 Low-memory batch search

To support the undesirable but realistic deployment situation in which a detection system’s random access memory is insufficient to contain an entire index of the corpus, an alternative mode of search has been designed and prototyped. A key theme in this research is the use of batch processing to minimise redundant costs. This batch search mode is an outcome of that effort.

4.4.1 Disk access costs

When constrained by memory, the first sacrifice is necessarily response time. Any access to on-disk data structures incurs large latencies, and for a search that necessarily performs random accesses within a large database, the many combined latencies severely reduce both throughput and responsiveness.

Clearly, if disk access is highly optimised to minimise latencies, the disk access cost of a search may remain small and affordable. The theoretical best case, assuming that the entire database must be read, is to read the entire database only once and contiguously without any disk seeks or pauses. Of course, to search for matches in a single text, far from all of the database must be read; but for almost any index structure, a large enough body of texts will require access to the entire database at least once.

4.4.2 Batching and chunking

This observation suggests a radical departure from traditional index search techniques. If a large enough body of texts requires (even most of) the entire database, it is likely cheaper to read the entire database only once rather than re-access parts of it repeatedly during the search. Of course, as the stated problem is that memory is too scarce to contain the entire database in memory, it is necessary to perform this one read in such a way that it can complete the search.

The solution is rather simple — read the database in chunks, and match chunks against the text batch. The whole database is still only read once, in contiguous chunks that are very efficient to read off disk, making good use of storage locality and minimising latencies. The problem remains that chunk matching itself incurs redundancies that must be minimised through good use of data structures and algorithms that themselves must be conservative in memory consumption.

The abstract data flow of the batch search is trivial. The search process requests chunk files from the operating system, decompresses them if they are compressed, and performs an appropriate match search against the active text batch, releasing chunks once they have been searched.

![Detection Process Diagram]

Any practical implementation will involve many optimisations to make the best possible use of system resources.
4.4.3 Parallelism

The CPU component of chunk search can be parallelised to use multiple threads and thus increase its throughput. This is redundant if the search is dominated entirely by disk read time, but if the CPU time is high, it can be reduced with parallel processing.

There are several approaches that can be used to parallelise the search, with different compromises in total concurrency and memory consumption. The one preferred by this project is a combination of parallel and pipelined processing.

In the data flow described above, the data dependencies are simple and static. To search a chunk, it must have been read from disk already, and this is independent of whether any other chunk has been read or searched.

It is clear that chunks can be searched entirely in parallel, at a level of concurrency appropriate for the processing resources available. In fact, they can also be read in parallel from multiple storage systems, increasing read throughput significantly.

In so doing, disk reading and match searching execute concurrently. However, in the most primitive implementation of this design, fast disk reads will fill up the queue unnecessarily, consuming great amounts of memory without gaining performance.

4.4.4 Metabuffer

A simple solution is to use a finite queue that causes the readers to block if there are too many chunks waiting to be searched. The same machinery can be reused to completely avoid dynamic memory allocation.

The “metabuffer” is a very simple data structure designed for this project, and most likely many others with similar requirements. Abstractly, the metabuffer is a combination of two blocking queues, with each queue node being a reference to a pre-allocated memory buffer.

For clarity, the two queues will be called empty and ready. The empty queue contains buffers that have undefined contents, and the ready queue contains buffers that have contents read from disk which have not yet been consumed by the searchers.

Each reader requests a buffer from the empty queue, blocking until one is available. If its read operation completes, the data in the buffer is now ready, so the buffer is inserted into the ready queue, thus unblocking a searcher.

Each searcher requests a buffer from the ready queue, blocking until one is available. Once its search is complete, it inserts the buffer into the empty queue, thus unblocking a reader.

When all requested chunks have been read, the metabuffer is placed into a read-only state, and so searchers will only read buffers that are already ready, and will not block for
any further buffers. Once all searches have completed, all buffers will remain in the *empty* queue, ready to be reused in another batch search.

Using well-known concurrent data structure implementation techniques, the cost of a buffer transfer is trivial, often only a pointer exchange. As only the buffer pointers are exchanged, and no bulk data is actually copied, so in practice this has very small overheads compared to a primitive synchronous read-and-search.

The number of buffers available in the metabuffer must be chosen to achieve at least the level of concurrency desired, i.e. at least as many as the number of readers plus the number of searches. In practice, having several more buffers in reserve can significantly help if either end of the process executes for longer than desired, as this reduces the time spent blocking on the opposite end. So if disk throughput drops unexpectedly, the searchers will still have work to complete, giving the readers time to catch up without stalling overall.

**4.4.5 Data structures**

As each chunk is likely to be small, it is appropriate to index each chunk into a data structure that does not need to scale especially well in size, but must be efficient in memory and construction time. In section 5.5.2 it is shown experimentally that the Mindex primitive is suitable for chunk search, as its CPU and memory costs are low, especially for small databases where large hash buckets cannot accumulate.

Alternatively, it may be preferable to index the text batch itself, and pass database chunks through as candidate texts. For a large number of batch texts, this is likely to be faster than re-searching the batch texts against the chunks. The Dexin structure described in section 3.6, optimised for longest-copy searches, is also shown in section 5.5.2 to be often superior in chunk search performance.

**4.4.6 Memory costs**

The peak memory consumption of a batch search, excluding automatic caches managed by the operating system and storage hardware, is essentially only the total of the batch texts, any database chunks being searched at the time, and any index structures involved in the search. If chunk sizes are small, and not many chunks are resident at any given time, the total memory cost is easily affordable.

Parallel searches increase the memory cost considerably because each processing unit (typically a thread) is responsible for an entire text chunk and possibly its associated index structure. This cost is multiplied by the number of active search threads.

**4.5 Distributed search**

When a single processing system is no longer sufficient to perform economical searches against the corpus required, it is necessary to resort to distributed search to expand the supported corpus size. Distributed search itself may integrate with threaded or batch search as appropriate to the system’s requirements and purpose.

The distributed processing model designed for this project follows the project’s pervasive “batch oriented” design philosophy. Virtually all remote operations are performed on a specific batch, including text imports, storage and searches.

Every node in the network is either a master or slave. Masters automatically issue storage and search requests to connected slaves. A master issuing requests to another master is effectively a *client* of that master.

The protocol as implemented is extremely simple, and all supported workflows are constructed from several primitive operations.
1. Node query
   Requests node information for distribution decisions

2. Batch add files
   Parses raw files to add to the batch

3. Batch add texts
   Adds pre-parsed texts to the batch

4. Batch search
   Find matches between the batch and any other texts

5. Batch store
   Permanently record the batch as part of the corpus

Assumptions within the protocol are kept to a minimum, allowing specific nodes to implement advanced functionality while remaining compatible with other nodes and clients. Because of the simplicity of the protocol and distribution requirements, very flexible deployment configurations are possible.

The distributed search architecture is a classical map-reduce model — each node is given a separate part of the work (partitioned either by corpus or candidates) and the per-node results are assembled into a complete result set to be stored or returned to clients.

### 4.5.1 Load balancing

In any distributed search design, it must be decided how load is to be distributed in order to ensure that requirements for scalability, redundancy and efficiency are met. It is clear that this must follow from specific deployment requirements.

At the highest level, load balancing can be performed either by dividing the search database across each node, or by dividing the candidate texts. There are advantages and disadvantages to each method.

In section 5.3.2 it is shown that the runtime of a Mindex text search is proportional to the number of candidate texts and the number of corpus texts. This suggests that partitioning either the corpus or the candidate set will reduce the total search runtime, though the consequences of either decision must be explored.

#### Corpus partitioning

By dividing the text archive evenly across processing nodes, and issuing the same candidate texts to each node during a search, complete search coverage can be achieved with low load on each individual node. Because the archive is partitioned, each node’s stored archive and live index remain small, and thus memory and storage costs are low.

It must be noted that the Mindex corpus size test results in section 5.3.1 confirm that the search runtime is, for a sufficiently high table size, only sublinearly proportional to corpus size — indeed, that is the entire purpose of the structure. Thus, finer corpus partitioning will most likely not reduce the search runtime — however, if the maximum memory size of each node is fixed, including more nodes allows more partitions to be searched in the same time.

For disk-backed batch search as described in section 4.4, corpus partitioning is very well suited. At a fine enough level of partitioning, in-memory indexing will be available, and disk batch search is unnecessary and inefficient. However, for any deployment in which disk reading is the bottleneck, splitting the load to multiple disks will improve performance. Although a single machine can accommodate a great number of disks, after
a certain point, CPU will once again become the bottleneck, and multiple machines must be recruited.

**Candidate set partitioning**

Partitioning the candidate set instead of the corpus will reduce the linear factor of a text search. However, as explored in section 4.3, the partitioning resolution must be appropriately balanced so that high concurrency is achieved without incurring significant overheads.

In general, this is difficult for all but very large candidate batches. Combined with the obvious inefficiency of replicating the entire corpus and its index per node, this mode of load balancing is unlikely to be a good choice in practice.

Candidate set partitioning is extremely ill-suited to disk-backed batch search as described in section 4.4. Unlike in-memory indexed search using an optimised primitive such as Mindex, disk batch search requires runtime that is at minimum directly proportional to the size of the corpus. As the corpus is not partitioned in this design, this runtime will be at its maximum for each node, and the savings made by partitioning the candidate set are negligible at best.

### 4.6 Conclusion

Several high-level design and implementation techniques have been discussed, intended for use as components in the engine of an efficient duplication detection system.

Batch processing is central to the design and implementation of the system and many of its techniques. It allows very simple work partitioning for parallel and distributed search, as well as high throughput disk-based search where redundant processing and memory overheads are minimised.

In chapter 5 these techniques will be subjected to performance evaluation.
Chapter 5

Experiments

There are several performance aspects of the system that must be evaluated in order to validate both the underlying science and the specific software design and implementation. Several experiments have been constructed and performed to quantitatively evaluate qualities and limitations of the experimental system.

5.1 Common experimental context

Design details of the duplication detection system to be used experimentally is described in chapters 3 and 4. Details specific to experimental use follow.

5.1.1 Random text generation specification

Every experiment is primarily conducted with randomly generated texts. As a key variable in almost every experiment is the total corpus or candidate set size, it is critical that this be controlled precisely, and that data artefacts of a “real” text corpus do not interfere with results.

Random text generation is performed with a C++ procedure prototyped as follows:

```cpp
void makeRandomTexts(
    Texts & texts, // std::vector to store texts
    size_t minwords, // minimum number of words (inclusive)
    size_t maxwords, // maximum number of words (inclusive)
    size_t maxvalue, // cardinality of random word set
    size_t count // number of such texts to generate
);
```

The procedure body is defined in `src/dox/test/textgen.cc`

For every experiment that uses random texts, these parameters will be explicitly stated in the test specification. The random words are selected uniformly from a set with the cardinality given in `maxvalue`. Although this is completely unlike the word distribution of any real text corpus, this is deliberately designed to minimise the statistical impact of irregularities within text material, while still allowing a necessary degree of control over the frequency of matches found.

5.1.2 Search algorithm specification

For the purpose of plagiarism detection in particular, a text match report should identify the substrings of a text that are found to match others in the database, as well as at least one confirmed source for each substring. So while any of the many “set” data structures are sufficient to confirm the existence of substrings, more data must be recorded to enumerate
all occurrences and thus allow a search result to identify the original text from which a string was copied.

To maximise the readability and relevance of reports to the system’s users, matching substrings should be as long as possible, and thus potentially many database candidates must be compared to return the strongest results for a given substring query.

Experience with this project has indicated that the most effective search reports record the largest possible scope of copied text (that is, beyond reasonable doubt, actually “copied”, with at least 5 non-trivial words matching in sequence) with a minimal set of source documents. Although this is a challenging optimisation problem for the theoretical best results, very simple search logic gives satisfactory results with computational efficiency.

The basic intention is that the search should consider all appropriate matches, but only retain those most useful to a human user.

Note that most of the data structures studied in this research operate on n-grams, and so an n-gram length must be defined, and the minimum match length must be at least as long as the n-gram length. For the experiments here, the minimum match length is defined as 5 normalised words, and the n-gram length is no greater than that.

The search algorithm progresses along a candidate text, storing at each point only the longest match found starting at that point, and then not searching any substrings until after the end of the longest match. Although this is not theoretically optimal in total match count or length, it is very computationally efficient and trivial to implement. A skeletal C++ version follows:

```cpp
std::vector<Copy> copies;

for (size_t i = 0; i < text.size(); i++) {
    Copy best;
    for (/* every candidate copy */) {
        Copy copy(/* parameters */);
        if (copy.length > best.length)
            best = copy;
    }
    if (best.length >= Copy::MIN_LENGTH) {
        copies.push_back(best);
        // copierEnd() is the position after the last
        // matching word in this text.
        // Subtract 1 since i will be incremented
        // before the next iteration.
        i = best.copierEnd() - 1;
        assert (i <= text.size());
    }
}
```

A deliberately minimal example is available in `src/dox/search/basicmatcher.cc` — this is the most rudimentary matcher possible, used for consistency testing (see section 5.2.2).

### 5.1.3 Timing specification

Unless otherwise stated, all timing measurements are given as raw numbers of milliseconds. Each “time” is actually the minimum of at least 8 consecutive executions. Averages have also been measured but not recorded, as although an average is, in these experiments,
5.2. EXACT SUBSTRING MATCHING

only very slightly higher than a minimum, it includes far more external pollution from the 
non-realtime system used for measurements.

Each search procedure is deterministic and idempotent. On a completely dedicated 
and deterministic experimental system, each execution would be essentially equivalent 
in execution time. However, as the experiments have been performed on a standard 
interactive desktop system with resources shared among applications and the operating 
system, timing pollution is observed in many samples. By recording only the minimum, the 
pollution is statistically minimised, leaving measurements that best reflect the expected 
performance on a dedicated system.

5.1.4 Experimental platform

All experiments, with the exception of distributed search, are performed on the same 
commodity desktop computer.

The system contains an Intel Core i7 920 at the stock 2793 Mhz frequency with Hyper-
Threading disabled, 6 gigabytes of DDR3 RAM at 1066 Mhz, and an eVGA X58 SLI 
motherboard. The processor has 4 physical cores, limiting its maximum level of real 
concurrency to 4 threads.

All source code is compiled to 64-bit binaries with gcc 4.3.3 on Ubuntu 9.04 with a 
64-bit userland and kernel (desktop variant). The precise compilation flags are recorded in 
the Sconstruct definition file in the project source code, except that experiments are run 
with -DNDEBUG so that runtime assertions do not interfere with runtime measurements.

5.2 Exact substring matching

The core of a detection system is a match detection primitive that enumerates copied 
substrings within some set of texts. Overall system performance is largely a reflection on 
the correct use of appropriate primitives.

The intent of these experiments is to evaluate the practicality of the implemented match 
primitives for substring search requirements in the context of an economical plagiarism 
detection system. These experiments and their results are not intended to influence the 
selection of substring search primitives for unrelated sets of requirements.

5.2.1 Primitives considered

Every match primitive retained for specification and experimentation is considered in 
this experiment. As Mindex has been the focus of optimisation efforts, it is used as the 
reference for partial indexing (see section 3.3.5).

- A simple suffix array as per section 3.7 (Bindex)

- A straightforward n-gram index as per section 3.4
  - With hash filtering (Index)
  - Without hash filtering (Index-NF)

- An optimised n-gram index as per section 3.5
  - With all of its n-grams, each at length 5 (Mindex)
  - With 1/2 of its n-grams, each at length 4 (Mindex-1)
  - With 1/3 of its n-grams, each at length 3 (Mindex-2)

- An inverted n-gram index as per section 3.6 (Dexin)
5.2.2 Correctness and consistency

The suggested primitives (see chapter 3) must be verified and evaluated experimentally, at least to ensure that the given implementations are correct and consistent with their requirements. To validate match coverage, primitives will be compared against a trivial brute-force substring matcher over randomly assembled sets of texts (see section 3.8). No runtime or memory efficiency results will be collected.

Each matching technique will be required to detect substring matches between a candidate set of 1024 texts and a corpus set of 1024 texts, with no whole text being repeated within a set or between sets. To enable n-gram techniques to function, the minimum match length is 5 symbols (in the representation, these are words, though this is not important to the algorithms or results). Each technique is expected to produce consistent match lists — while they are not required to refer to the same “copied” chunks in the corpus, they are required to highlight the same “copier” chunks in the candidate set, precise to exactly the number of symbols copied as defined in section 5.1.2.

makeRandomTexts(..., 100, 1000, 20, 1024) is used for both the corpus and candidate sets.

<table>
<thead>
<tr>
<th>Matcher</th>
<th>Number of copies</th>
<th>Total length of copies</th>
<th>Consistent</th>
</tr>
</thead>
<tbody>
<tr>
<td>Basic</td>
<td>3414</td>
<td>17150</td>
<td></td>
</tr>
<tr>
<td>Bindex</td>
<td>3414</td>
<td>17150</td>
<td>Y</td>
</tr>
<tr>
<td>Index-NF</td>
<td>3414</td>
<td>17150</td>
<td>Y</td>
</tr>
<tr>
<td>Index</td>
<td>3414</td>
<td>17150</td>
<td>Y</td>
</tr>
<tr>
<td>Mindex</td>
<td>3414</td>
<td>17150</td>
<td></td>
</tr>
<tr>
<td>Mindex-1</td>
<td>3413</td>
<td>17145</td>
<td>N</td>
</tr>
<tr>
<td>Mindex-2</td>
<td>3413</td>
<td>17144</td>
<td>N</td>
</tr>
</tbody>
</table>

Table 5.1: Exact substring match consistency experiment

The results for Bindex, Index and Mindex confirm correctness and consistency — within the requirements given, and for the test data available, each primitive’s design and implementation is correct.

Mindex-1 and Mindex-2 have a small inconsistency in this experiment. In theory, they should be able to give identical results to a complete Mindex (see section 3.3.5). The flaw arises because of a subtle interaction between the primitive’s own search logic and the best-first search (see section 5.1.2) — because the latter is an aggressive algorithm that skips subsequent candidates, it requires candidate information that is often excluded in the Mindex-N until later in the search. This is easily solved with changes to the aggressive search, but the solution has yet to be implemented in the experimental apparatus.

5.2.3 Memory consumption

To indicate the memory overheads of the structures, this experiment loads each data structure with precisely 512 MiB of normalised text data (see section 3.1) and measures the total memory consumption before and after data is loaded. Each table structure is given $2^{22}$ hash buckets, accounting for almost all of the starting overhead.

Note that the raw text data is not included in these counts. The raw text data requires 32 bits per retained word (see section 3.1), so the 512 MiB of normalised texts represents precisely $2^{27}$ words.

As each table structure is essentially equivalent when empty, the memory overhead appears uniform. This constant memory is a table of dynamic array heads, with none
## 5.2. Exact Substring Matching

<table>
<thead>
<tr>
<th>Primitive</th>
<th>Empty</th>
<th>Full</th>
<th>Delta</th>
</tr>
</thead>
<tbody>
<tr>
<td>Index</td>
<td>96</td>
<td>1119</td>
<td>1023</td>
</tr>
<tr>
<td>Mindex</td>
<td>96</td>
<td>608</td>
<td>512</td>
</tr>
<tr>
<td>Mindex-1</td>
<td>96</td>
<td>352</td>
<td>256</td>
</tr>
<tr>
<td>Mindex-2</td>
<td>96</td>
<td>267</td>
<td>171</td>
</tr>
<tr>
<td>Dexin</td>
<td>96</td>
<td>1631</td>
<td>1535</td>
</tr>
<tr>
<td>Bindex</td>
<td>&lt; 1</td>
<td>512</td>
<td>512</td>
</tr>
</tbody>
</table>

Table 5.2: Data structure memory consumption in MiB (64-bit system)

of the arrays allocated. When loaded, the difference is significant, but in line with the specifications in chapter 3.

*Bindex* appears to have essentially no notable overhead. This is to be expected, as it is only a sorted array, with a small list of subtexts for offset encoding (see section 3.5.1).

*Dexin* consumes 3 times as much memory as an equivalent unfiltered *Mindex*, as it must also reserve space for match records (see section 3.6). This is acceptable because it is designed to index far smaller text sets, and the extra memory commitment affords optimisations to batch search (see section 5.5).

### 5.2.4 Scalability

In order to evaluate the design and implementation of the match primitives, and to appropriate their selection and configuration to system requirements and constraints, simple search runtime measurement experiments are conducted.

The intent is to identify the dominant runtime complexity class by varying the corpus size and measuring the resulting search runtime. This helps to select search primitives (and configurations thereof) to appropriately balance runtime and memory use for different corpus sizes.

For the table-based primitives (i.e. everything except Bindex) the table size is also varied while maintaining corpus size, to confirm the extent to which table size influences runtime growth.

**Varying corpus size**

Several primitives are subjected to the same search task with a varying corpus size. The experiment is performed twice, once with a very high match rate and once with a very low match rate.

Each table structure is given $2^{22}$ hash buckets. For the high match rate, the candidate set is generated with `makeRandomTexts(..., 8192, 8192, 20, 1000)`, and each corpus is generated with `makeRandomTexts(..., 8192, 8192, 20, size)` where `size` is the corpus size. The low match rate is the same except using a random set of 200 words instead of 20, leading to a far lower match rate.
CHAPTER 5. EXPERIMENTS

Figure 5.1: Primitive runtime over various corpus sizes (high match rate)

Figure 5.2: Primitive runtime over various corpus sizes (low match rate)
5.2. EXACT SUBSTRING MATCHING

See appendix A.1.1 for raw timing data.

Every matcher exhibits sublinear runtime growth over corpus size. However, further experiments will show this to be rather misleading — see sections 5.2.4 and 5.3.1.

Although Bindex — the classic suffix array using binary search — scales very well throughout any corpus size, its resulting total runtime is never even nearly competitive with table-based primitives. That is, although the amount of CPU work rises only logarithmically, the constant factor of the work is high — partly because any binary comparison must be performed with a deep search that requires access to each text, and partly because the number of such comparisons tends to be far higher than in a table-based structure with a table of sufficient size.

The results unambiguously confirm that table-based structures can significantly outperform classic suffix arrays with competitive memory overhead. The remaining question is how such table structures should be specified and optimised to make the best use of system resources.

The significant differences between operation in low and high match rates are the time spent eliminating false positives and the time spent considering confirmed matches. Note that as Index-NF has its hash filtering disabled, false positives are far more expensive to eliminate, and its efficiency suffers considerably.

For a high match rate, Mindex clearly outperforms Mindex-1 because it has to consider fewer n-grams to find complete matches, and incomplete ones are mostly eliminated by hash filtering. However, for a low match rate, Mindex-1 actually outperforms Mindex because it has far fewer candidates to check, and almost all of them are eliminated by the same hash filtering.

Varying table size

Several primitives are subjected to the same search task with a varying table size. As Bindex has no table, it is excluded from this experiment. A candidate set is generated with makeRandomTexts(..., 8192, 8192, 20, 1000), and each corpus is generated with makeRandomTexts(..., 8192, 8192, 20, 4096). Note that as random words are drawn uniformly from a set of 20, the match rate is very high.

See appendix A.1.2 for raw timing data.

It is clear that table-based structures rely on a sufficient table size to perform efficiently. This is easily explained.

Recall that for any given candidate-vs-corpus combination, the set of matches is equivalent regardless of primitive design or specification. The table structure merely minimises the number of false positives that must be eliminated during the search — correct matches will still occupy the same bucket regardless of bucket count.

At a low rate of false entries per bucket, this elimination is very efficient — the entire bucket may be loaded in a single memory page, and quickly scanned using the efficient elimination techniques defined in section 3.5. At a high rate of false entries per bucket, even this efficient elimination may consume far more CPU time and memory latency than the is used necessarily for confirming actual matches.

Mindex-1 and especially Mindex-2 have smaller n-gram sizes, so false positives are less likely at the bucket level, but far more of these n-grams must be checked to find sufficiently long copies. For this reason, the effect of table size is far less important for these partial indices.

Dexin operates very differently from any other Index-style matcher (see section 3.6). In the Dexin search, the candidate set is indexed and the corpus set is enumerated. The very small candidate set does not load the table to the extent that it requires a large table size. However, as the runtime grows in direct proportion to the enumerated corpus size,
it does not appear to “scale” as the matchers which attempt to minimise the influence of the corpus size.

The results confirm that table size is a critical factor in the performance of table-based structures. The precise details necessarily vary by implementations and deployments, but it is clear that there is, for any implementation, a ratio between table size and corpus size that maximises runtime efficiency. See section 5.3.1 for a more elaborate evaluation using Mindex in particular.

5.3 Mindex performance

As the most optimised search primitive, Mindex (see section 3.5) is subjected to further evaluation so that its configuration in practice can be calibrated empirically.

Each text set is generated with `makeRandomTexts(..., 8192, 8192, 20, size)` where `size` is the text set’s intended size.

5.3.1 Varying corpus and table sizes

This experiment attempts to establish the efficiency relationship between corpus size and table size. For various table sizes, from $2^{12}$ to $2^{22}$, measurements are made of the total runtime for various corpus sizes. This will show empirically how table size affects runtime efficiency.

See appendix A.2.2 for raw timing data.

It is clear that, for a large corpus, larger table sizes incur lower runtime costs. However, it is perhaps surprising that lower table sizes are more efficient for a very small corpus. It is obvious from the design that the number of instructions required for a search will never be higher just due to a larger table, but it is perhaps less obvious that a large table size requires considerably more memory accesses that incur runtime costs due to latency.

As increased table sizes necessarily incur some memory size overhead, with diminishing runtime returns past a certain (experimentally identifiable) point, it is suggested that
5.3. MINDEX PERFORMANCE

![Graph showing Mindex runtime over various corpus and table sizes](image1)

**Figure 5.4:** *Mindex* runtime over various corpus and table sizes

*Mindex* configuration be performed in response to experimental results on any system to be used in production.

### 5.3.2 Varying corpus and candidate sizes

This experiment merely confirms that candidate set size has only a linear effect on total search runtime, regardless of corpus size. Although this is intended and expected, it must be verified experimentally to confirm that this is not a misleading factor in other experiments or in practical use.

![Graph showing Mindex runtime over various corpus and candidate set sizes](image2)

**Figure 5.5:** *Mindex* runtime over various corpus and candidate set sizes

See appendix A.2.1 for raw timing data. The results unambiguously confirm that candidate size has no non-linear influence on total search runtime.
5.4 Threading scalability

The experiment explores the consequences of the domain partitioning design used for threaded search. In particular, it is not possible to achieve a performance improvement by increasing thread count beyond the candidate text count, as the extra threads will receive no work.

This experiment is the first to suffer considerably from being executed on a non-dedicated desktop machine. While previous experiments could easily be scheduled on to a single processor core and receive near-dedicated performance, this experiment must receive not only a maximum of total processor time across all 4 available cores, but also an unrealistic dedication from the operating system’s thread scheduler. Even so, it is illustrative of the efficiency challenges faced by a real (if hardly recommendable) system.

The text set is generated with `makeRandomTexts(..., 81920, 81920, 20, size)` where `size` is the text set’s intended size, starting from 1 text. The corpus is generated with `makeRandomTexts(..., 8192, 8192, 20, 16384)`. The base search structure is a Mindex with a table size of $2^{22}$.

![Figure 5.6: Search runtime over various thread and candidate text counts](image)

See appendix A.3 for raw timing data.

Note the apparent discrepancy when 4 texts are matched with 3 threads. The runtime cannot be improved beyond that of 2 texts, because even though 2 of the 3 threads can each complete a single text’s search, the remaining thread must complete the search for two texts. This is only an artefact of the partitioning design, and is negligible at high enough candidate set sizes.

Although linear performance improvement has not been achieved in this experiment, the results are close, reaching a 376% throughput gain using 4 threads, equivalent to an efficiency of 94%. The remaining inefficiencies are the result of software and hardware elements, as well as runtime pollution on a non-dedicated system. Optimisation efforts from here would be mostly wasted, giving a further efficiency improvement of 6% at theoretical best (from 376% to 400%).
5.5 Batch processing performance

A key argument of this thesis is that batch processing affords many opportunities for both micro and macro optimisation, especially by minimising redundant overheads. The benefits of batch processing have been evaluated for many fields and problems, and this thesis aims to contribute further resources for batch processing as applied to plagiarism detection and similar text search tasks.

5.5.1 Specifications

To quantify the extent to which batch processing improves the efficiency of certain realistic tasks, several such tasks will be executed with text sets of varying cardinalities, starting from a set of only one text (representing a single-user interactive submission) and progressing in several stages to sets of 4096 texts (representing a large group submission). This is tested against different corpus sizes, starting from 64 MiB and rising by a factor of two to 1024 MiB.

Runtime is measured as the total real time to complete the search. The corpus size is run through the batch matcher at a rate of 30MB/s, where the batch size is counted by *Texts*. In order to completely eliminate the effects of actual disk performance, and achieve precisely the data rate desired, the IO is all simulated. The texts are random, and are released in chunks of 16 MiB (see section 4.4). Lower chunk sizes could smooth out runtime during a search, but would necessarily incur higher processing overheads, as more result sets must be combined.

The candidate set is populated with `makeRandomTexts(\ldots, 8192, 8192, 200, N)` where N is the set cardinality declared in the leftmost result column.

Each chunk is populated with `makeRandomTexts(\ldots, 8192, 8192, 200, 511)`, except that instead of being written to an `std::vector`, each text is written to the chunk buffer. As each text has a small metadata overhead (hash + number of words), the full 512 texts will not fit into a 16 MiB chunk.

Each index structure is given a table size of $2^{22}$. 

Figure 5.7: Search throughput improvement over various thread and candidate text counts
5.5.2 Results

The results presented are for a corpus of 1024 MiB in size. See appendix A.4 for raw timing data for many corpus sizes. It is important to note that the runtime will never be below the total IO time; the intent is to minimise the extent to which it is above the IO time.

Four engine configurations are tested: Mindex and Dexin with 1 and 4 threads each. It is clear that, for small candidate set sizes, the engine configuration is essentially irrelevant, as the search is bound by IO time. When the CPU time grows beyond the IO time, the engine configuration is then critical, with the results showing that, for 4096 candidate texts, Mindex with one thread takes over 8 times as long as Dexin with 4 threads.

Dexin is specifically designed for this type of batch search (see section 3.6), and it is encouraging to note that its runtime is extremely efficient even for a very high number of candidate texts, and that higher thread counts for both matchers help significantly in reducing CPU load.

This experiment, repeated on a deployment server, helps to influence deployment configuration decisions for maximum efficiency.

Figure 5.8: Search runtime for various batch engines and candidate set sizes
5.6 Distributed search efficiency

A realistic deployment of the system on several processing nodes, with one master, will have its performance evaluated. Performance conclusions are difficult to draw as they vary largely on the work performed and the environment in which it is performed. However, it is possible to estimate the scalability limitations in the system’s techniques for distributed search.

5.6.1 Available distributed system

All experiments are to be performed on the Monash eScience and Grid Engineering Laboratory Enterprise Grid (see MeSsAGE Lab — Enterprise Grid (2009)) within Monash University (EAST nodes). It is extremely likely that the computer grid’s resources are shared between any number of other experiments, so performance will be neither deterministic nor homogeneous — for this reason, simple sampling techniques will be used to extract the cleanest possible performance data.

Unfortunately, there has been no opportunity to load a usefully large text corpus into the distributed system. Thus all experiments are performed using only randomly generated texts.

5.6.2 Experimental aims

Given the experimental limitations, only a few simple measurements will be taken. Because the computer grid performs unevenly due to shared load, the total search time is not representative of software performance, and thus no large-scale scalability conclusions can be drawn. However, because there are always nodes which are under little or no other load, the minimum time selected from all node results is realistically representative of a dedicated homogeneous grid.

Two measurements have been defined for this experiment.

First, the experiment measures the per-node round trip time for text searches at different batch sizes. This is representative of every stage of processing and communication specific to a processing node, including result aggregation in the master itself, which also occurs once per node and is amortised over the lifetime of the search.

Second, the experiment measures only the per-node communication overhead for the same text searches. The in-node search time is completely eliminated from the results, by forcing the search to run in 20 seconds (sleeping for as much time as remains of the 20 seconds), then subtracting the same 20 seconds from the round trip time. What remains is the communication overhead in software and hardware, including the time taken to request the search and the time taken to return the results.

Although forcing each search to take 20 seconds also forces all of the result communications to occur at the same time, this is arguably a desirable consequence, as it simulates a highly homogeneous cluster in which this sort of burst response is to be expected. It is therefore important to identify if the communication overheads rise super-linearly, as this indicates a bottlenecks problem in the software and/or hardware.

Each experiment is conducted with a set of candidate text batch sizes, starting from one text (a simulation of interactive search) and rising in powers of two to 1024 texts (a simulation of a large-scale batch search). For each set of candidate texts, the same experiment will be performed 16 times, and the minimum and maximum time samples will be recorded in the data table. Each node is populated with 512 MiB of random texts as its corpus. To verify that the experiment is performing a real search over the stated amount of data, the number of copies found is also recorded, indicating the direct proportionality between text count and copy count.
5.6.3 Results

See appendix A.1.2 for raw timing data.

Apart from a fixed latency visible at low text counts, the round trip time is directly proportional to the candidate text count.

With the search runtime subtracted, the fixed communication latency is far more visible. For low text counts, the number of copies is low enough that the time is essentially constant. At much higher text counts, the propagation time begins to grow slightly more than linearly, though it is still a very small component of the overall search time. For searches with more results returned, this may be more of an issue.

5.7 Conclusion

Many critical performance aspects of the system and its constituent techniques have been evaluated. The results are presented in reduced form in this chapter, and as raw data tables in appendix A.

It is encouraging to note that the performance of the system meets intentions and expectations, and all performance characteristics have been accounted for in terms of the design and implementation decisions. With this robust foundation in place, further work may be focused to address specific limitations highlighted by these experiments, and real system use may be guided by the experimental results and recommendations.
Chapter 6

Conclusion

It is believed that this project has delivered all of the contributions outlined in section 1.2.

6.1 Primitives

The search primitives have been designed, implemented and experimentally evaluated.

The most common primitive template is the $n$-gram hash table (section 3.3), supporting very efficient match enumeration and confirmation when configured with a sufficiently large table size and appropriate optimisation techniques.

*Index* (section 3.4) is a straightforward realisation of this design, tested with 64 bit $n$-gram references. It enables very efficient match searches when configured as appropriate for its corpus content.

*Mindex* (section 3.5) is very promising as a highly optimised and configurable $n$-gram hash table, with several techniques implemented and evaluated. XOR hash filtering reduces the runtime cost of hash collision elimination at no cost to memory. Partial indexing reduces the memory consumption of an index by excluding $n$-grams that are not essential to enumerate match candidates, at a considerable but often acceptable runtime cost. Filtered indexing also reduces memory consumption if the texts to be searched have a known limited set of $n$-grams. Offset encoding allows small $n$-gram references to identify specific positions in the corpus unambiguously. Overall, *Mindex* is a prime candidate for use in interactive duplication detection where sufficient memory is available to index the entire corpus, even if only using partial indexing.

*Dexin* (section 3.6) is an inverted index supporting efficient batch search for longest non-overlapping copies (see sections 5.1.2 and 5.5). It minimise redundant work during a batch search, by retaining only long matches that are finally collected aggressively. *Dexin* is a prime candidate for use in high-throughput duplication detection where the corpus is too large to index in memory.

6.2 Techniques

Content hashes (section 4.1) are useful for portable unique identification of texts. Although naïve use admits security holes, well-known techniques such as HMAC hashing largely eliminate such vulnerabilities.

Batch search (section 4.4) has been shown to enable high-throughput search that is every economical in memory use, reading an on-disk corpus through a search engine. Although *Mindex* is adequate for batch use, *Dexin* has been shown to be far more efficient in batch searches with a large candidate text set, which is precisely the intended use of batch search.
Threaded search (section 4.3) has been shown to improve match throughput even with the very limited domain partitioning and work balancing techniques implemented. Threading is also especially useful in batch search, reducing CPU load when CPU load is a bottleneck.

Distributed search (section 4.5) has been shown to have low overheads in processing and communication, enabling efficient horizontal scaling when multiple systems are available. Although reduction in total runtime is best achieved with more memory and faster CPUs, when neither can be improved, adding more processing nodes allows the corpus to be grown without (significantly) increasing total search time.

6.3 Results

A breadth of experimental results has been collected, presented and discussed (chapter 5). The results confirm that performance goals have been met, and that the system as a whole is practical for use in real-world duplication detection scenarios. Specific sets of results can be used to guide implementation and configuration optimisations for real systems.

6.4 Further work

It is clear that the field of remaining optimisations in text search and large-scale processing is only expanding as new techniques and systems are developed. Further efforts in the field will improve existing search techniques and initiate new techniques suitable for new sets of requirements. Even more so, the user-facing aspects of duplication detection systems — though outside the scope of this project in particular — warrant further development and refinement, as the underlying technology is only as useful as the results it presents to users.
Appendix A

Raw results

A.1 Primitive performance

A.1.1 Varying corpus size

<table>
<thead>
<tr>
<th>Texts</th>
<th>Index</th>
<th>Index-NF</th>
<th>Mindex</th>
<th>Mindex-1</th>
<th>Mindex-2</th>
<th>Dexin</th>
<th>Bindex</th>
</tr>
</thead>
<tbody>
<tr>
<td>2⁰⁸</td>
<td>2005</td>
<td>2105</td>
<td>1996</td>
<td>3614</td>
<td>21623</td>
<td>3168</td>
<td>8958</td>
</tr>
<tr>
<td>2⁰⁹</td>
<td>2212</td>
<td>2382</td>
<td>2192</td>
<td>5097</td>
<td>45354</td>
<td>4419</td>
<td>13196</td>
</tr>
<tr>
<td>2¹⁰</td>
<td>2564</td>
<td>2911</td>
<td>2467</td>
<td>7527</td>
<td>80993</td>
<td>6937</td>
<td>18572</td>
</tr>
<tr>
<td>2¹¹</td>
<td>3180</td>
<td>3839</td>
<td>2889</td>
<td>11822</td>
<td>137796</td>
<td>11877</td>
<td>23896</td>
</tr>
<tr>
<td>2¹²</td>
<td>4427</td>
<td>5737</td>
<td>3679</td>
<td>19903</td>
<td>237089</td>
<td>21756</td>
<td>35307</td>
</tr>
<tr>
<td>2¹³</td>
<td>6949</td>
<td>9364</td>
<td>5205</td>
<td>35834</td>
<td>441716</td>
<td>41657</td>
<td>63582</td>
</tr>
<tr>
<td>2¹⁴</td>
<td>12466</td>
<td>17062</td>
<td>8569</td>
<td>69175</td>
<td>870628</td>
<td>81627</td>
<td>118279</td>
</tr>
</tbody>
</table>

Table A.1: Search runtime for different primitives and corpus sizes (high match rate)

<table>
<thead>
<tr>
<th>Texts</th>
<th>Index</th>
<th>Index-NF</th>
<th>Mindex</th>
<th>Mindex-1</th>
<th>Mindex-2</th>
<th>Dexin</th>
<th>Bindex</th>
</tr>
</thead>
<tbody>
<tr>
<td>2⁰⁸</td>
<td>1256</td>
<td>1639</td>
<td>1248</td>
<td>1028</td>
<td>1086</td>
<td>2007</td>
<td>6949</td>
</tr>
<tr>
<td>2⁰⁹</td>
<td>1490</td>
<td>2286</td>
<td>1469</td>
<td>1217</td>
<td>1370</td>
<td>2449</td>
<td>9073</td>
</tr>
<tr>
<td>2¹⁰</td>
<td>1698</td>
<td>3301</td>
<td>1659</td>
<td>1445</td>
<td>1832</td>
<td>3330</td>
<td>11549</td>
</tr>
<tr>
<td>2¹¹</td>
<td>1900</td>
<td>5095</td>
<td>1865</td>
<td>1672</td>
<td>2503</td>
<td>5129</td>
<td>14163</td>
</tr>
<tr>
<td>2¹²</td>
<td>2210</td>
<td>8492</td>
<td>2231</td>
<td>1998</td>
<td>3661</td>
<td>8667</td>
<td>16769</td>
</tr>
<tr>
<td>2¹³</td>
<td>2791</td>
<td>14940</td>
<td>3045</td>
<td>2737</td>
<td>5917</td>
<td>15791</td>
<td>19550</td>
</tr>
<tr>
<td>2¹⁴</td>
<td>3942</td>
<td>29849</td>
<td>5647</td>
<td>5019</td>
<td>11190</td>
<td>29969</td>
<td>23097</td>
</tr>
</tbody>
</table>

Table A.2: Search runtime for different primitives and corpus sizes (low match rate)
## APPENDIX A. RAW RESULTS

### A.1.2 Varying table size

<table>
<thead>
<tr>
<th>Buckets</th>
<th>Index</th>
<th>Index-NF</th>
<th>Mindex</th>
<th>Mindex-1</th>
<th>Mindex-2</th>
<th>Dexin</th>
</tr>
</thead>
<tbody>
<tr>
<td>$2^{16}$</td>
<td>7385</td>
<td>75868</td>
<td>7147</td>
<td>21988</td>
<td>235730</td>
<td>42067</td>
</tr>
<tr>
<td>$2^{18}$</td>
<td>5095</td>
<td>22639</td>
<td>4567</td>
<td>20240</td>
<td>235275</td>
<td>27230</td>
</tr>
<tr>
<td>$2^{20}$</td>
<td>4610</td>
<td>8922</td>
<td>3942</td>
<td>19953</td>
<td>235295</td>
<td>23555</td>
</tr>
<tr>
<td>$2^{22}$</td>
<td>4315</td>
<td>5524</td>
<td>3664</td>
<td>19803</td>
<td>235330</td>
<td>21750</td>
</tr>
<tr>
<td>$2^{24}$</td>
<td>4282</td>
<td>4711</td>
<td>3665</td>
<td>19821</td>
<td>235329</td>
<td>21751</td>
</tr>
</tbody>
</table>

Table A.3: Search runtime for different primitives and hash table sizes

### A.2 Mindex performance

#### A.2.1 Varying corpus and candidate sizes

<table>
<thead>
<tr>
<th>Candidates</th>
<th>$2^{00}$</th>
<th>$2^{02}$</th>
<th>$2^{04}$</th>
<th>$2^{06}$</th>
<th>$2^{08}$</th>
<th>$2^{10}$</th>
<th>$2^{12}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$2^{00}$</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>$2^{02}$</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>4</td>
<td>6</td>
<td>13</td>
</tr>
<tr>
<td>$2^{04}$</td>
<td>11</td>
<td>12</td>
<td>12</td>
<td>14</td>
<td>20</td>
<td>27</td>
<td>37</td>
</tr>
<tr>
<td>$2^{06}$</td>
<td>53</td>
<td>53</td>
<td>55</td>
<td>62</td>
<td>82</td>
<td>110</td>
<td>146</td>
</tr>
<tr>
<td>$2^{08}$</td>
<td>212</td>
<td>214</td>
<td>221</td>
<td>247</td>
<td>331</td>
<td>442</td>
<td>588</td>
</tr>
<tr>
<td>$2^{10}$</td>
<td>850</td>
<td>857</td>
<td>885</td>
<td>993</td>
<td>1330</td>
<td>1789</td>
<td>2325</td>
</tr>
<tr>
<td>$2^{12}$</td>
<td>3394</td>
<td>3422</td>
<td>3541</td>
<td>3976</td>
<td>5347</td>
<td>7272</td>
<td>9535</td>
</tr>
<tr>
<td>$2^{14}$</td>
<td>13656</td>
<td>13765</td>
<td>14211</td>
<td>16001</td>
<td>21693</td>
<td>29577</td>
<td>38895</td>
</tr>
</tbody>
</table>

Table A.4: Search runtime for Mindex with different candidate set and corpus set sizes

#### A.2.2 Varying corpus and table sizes

<table>
<thead>
<tr>
<th>Buckets</th>
<th>$2^{12}$</th>
<th>$2^{14}$</th>
<th>$2^{16}$</th>
<th>$2^{18}$</th>
<th>$2^{20}$</th>
<th>$2^{22}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$2^{12}$</td>
<td>130</td>
<td>173</td>
<td>283</td>
<td>689</td>
<td>2374</td>
<td>8859</td>
</tr>
<tr>
<td>$2^{14}$</td>
<td>119</td>
<td>149</td>
<td>198</td>
<td>317</td>
<td>821</td>
<td>2792</td>
</tr>
<tr>
<td>$2^{16}$</td>
<td>98</td>
<td>129</td>
<td>166</td>
<td>230</td>
<td>426</td>
<td>1039</td>
</tr>
<tr>
<td>$2^{18}$</td>
<td>104</td>
<td>119</td>
<td>178</td>
<td>273</td>
<td>381</td>
<td>574</td>
</tr>
<tr>
<td>$2^{20}$</td>
<td>174</td>
<td>180</td>
<td>202</td>
<td>276</td>
<td>377</td>
<td>486</td>
</tr>
<tr>
<td>$2^{22}$</td>
<td>210</td>
<td>212</td>
<td>219</td>
<td>245</td>
<td>329</td>
<td>439</td>
</tr>
</tbody>
</table>

Table A.5: Search runtime for Mindex with different corpus and table sizes
### A.3 Threaded search scalability

<table>
<thead>
<tr>
<th>Threads</th>
<th>Texts</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>626</td>
<td>1250</td>
<td>1877</td>
<td>2508</td>
<td>3129</td>
<td>3757</td>
<td>4385</td>
<td>5039</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>619</td>
<td>644</td>
<td>1260</td>
<td>1284</td>
<td>1895</td>
<td>1927</td>
<td>2533</td>
<td>2570</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>620</td>
<td>641</td>
<td>655</td>
<td>1270</td>
<td>1297</td>
<td>1310</td>
<td>1920</td>
<td>1953</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>620</td>
<td>642</td>
<td>655</td>
<td>666</td>
<td>1280</td>
<td>1299</td>
<td>1321</td>
<td>1331</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>619</td>
<td>643</td>
<td>687</td>
<td>667</td>
<td>1124</td>
<td>1299</td>
<td>1317</td>
<td>1331</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>620</td>
<td>643</td>
<td>731</td>
<td>722</td>
<td>1114</td>
<td>1224</td>
<td>1333</td>
<td>1408</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>619</td>
<td>643</td>
<td>713</td>
<td>747</td>
<td>1149</td>
<td>1231</td>
<td>1349</td>
<td>1375</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>620</td>
<td>644</td>
<td>679</td>
<td>782</td>
<td>1180</td>
<td>1252</td>
<td>1325</td>
<td>1387</td>
<td></td>
</tr>
</tbody>
</table>

Table A.6: Threaded search runtime for different candidate set sizes and thread counts (raw timing)

<table>
<thead>
<tr>
<th>Threads</th>
<th>Texts</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>100%</td>
<td>100%</td>
<td>100%</td>
<td>100%</td>
<td>100%</td>
<td>100%</td>
<td>100%</td>
<td>100%</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>101%</td>
<td>194%</td>
<td>148%</td>
<td>195%</td>
<td>165%</td>
<td>194%</td>
<td>173%</td>
<td>196%</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>100%</td>
<td>195%</td>
<td>286%</td>
<td>197%</td>
<td>241%</td>
<td>286%</td>
<td>228%</td>
<td>258%</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>100%</td>
<td>194%</td>
<td>286%</td>
<td>376%</td>
<td>244%</td>
<td>289%</td>
<td>331%</td>
<td>378%</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>101%</td>
<td>194%</td>
<td>273%</td>
<td>376%</td>
<td>278%</td>
<td>289%</td>
<td>332%</td>
<td>378%</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>100%</td>
<td>194%</td>
<td>256%</td>
<td>347%</td>
<td>280%</td>
<td>306%</td>
<td>328%</td>
<td>357%</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>101%</td>
<td>194%</td>
<td>263%</td>
<td>335%</td>
<td>272%</td>
<td>305%</td>
<td>325%</td>
<td>366%</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>100%</td>
<td>194%</td>
<td>276%</td>
<td>320%</td>
<td>265%</td>
<td>300%</td>
<td>330%</td>
<td>363%</td>
<td></td>
</tr>
</tbody>
</table>

Table A.7: Threaded search runtime for different candidate set sizes and thread counts (throughput increase)
## APPENDIX A. RAW RESULTS

### A.4 Batch search performance

#### A.4.1 One Mindex per thread

<table>
<thead>
<tr>
<th>Texts</th>
<th>64 MiB</th>
<th>128 MiB</th>
<th>256 MiB</th>
<th>512 MiB</th>
<th>1024 MiB</th>
</tr>
</thead>
<tbody>
<tr>
<td>$2^0$</td>
<td>3208</td>
<td>5205</td>
<td>9204</td>
<td>18206</td>
<td>35207</td>
</tr>
<tr>
<td>$2^2$</td>
<td>3255</td>
<td>5259</td>
<td>9255</td>
<td>18260</td>
<td>35256</td>
</tr>
<tr>
<td>$2^4$</td>
<td>3369</td>
<td>5365</td>
<td>9365</td>
<td>18364</td>
<td>35362</td>
</tr>
<tr>
<td>$2^6$</td>
<td>3540</td>
<td>5542</td>
<td>9541</td>
<td>18544</td>
<td>35541</td>
</tr>
<tr>
<td>$2^8$</td>
<td>4295</td>
<td>7350</td>
<td>13702</td>
<td>26305</td>
<td>51472</td>
</tr>
<tr>
<td>$2^{10}$</td>
<td>10787</td>
<td>19118</td>
<td>35973</td>
<td>69850</td>
<td>137531</td>
</tr>
<tr>
<td>$2^{12}$</td>
<td>35209</td>
<td>63029</td>
<td>115154</td>
<td>224304</td>
<td>436265</td>
</tr>
</tbody>
</table>

Table A.8: Batch search runtime varying by batch size and corpus size (Mindex)

#### A.4.2 One Dexin per thread

<table>
<thead>
<tr>
<th>Texts</th>
<th>64 MiB</th>
<th>128 MiB</th>
<th>256 MiB</th>
<th>512 MiB</th>
<th>1024 MiB</th>
</tr>
</thead>
<tbody>
<tr>
<td>$2^0$</td>
<td>3455</td>
<td>5447</td>
<td>9447</td>
<td>18446</td>
<td>35451</td>
</tr>
<tr>
<td>$2^2$</td>
<td>3454</td>
<td>5454</td>
<td>9476</td>
<td>18453</td>
<td>35455</td>
</tr>
<tr>
<td>$2^4$</td>
<td>3481</td>
<td>5479</td>
<td>9475</td>
<td>18479</td>
<td>35474</td>
</tr>
<tr>
<td>$2^6$</td>
<td>3583</td>
<td>5581</td>
<td>9582</td>
<td>18583</td>
<td>35588</td>
</tr>
<tr>
<td>$2^8$</td>
<td>4033</td>
<td>6534</td>
<td>12548</td>
<td>23282</td>
<td>45874</td>
</tr>
<tr>
<td>$2^{10}$</td>
<td>6808</td>
<td>9139</td>
<td>13713</td>
<td>24367</td>
<td>45597</td>
</tr>
<tr>
<td>$2^{12}$</td>
<td>18977</td>
<td>27583</td>
<td>45024</td>
<td>375905</td>
<td>136128</td>
</tr>
</tbody>
</table>

Table A.9: Batch search runtime varying by batch size and corpus size (Dexin)
A.5 Distributed search performance

A.5.1 Round trip time

<table>
<thead>
<tr>
<th>Candidate texts</th>
<th>Copies found</th>
<th>Min time (ms)</th>
<th>Max time (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>$2^0$</td>
<td>1097</td>
<td>12</td>
<td>117</td>
</tr>
<tr>
<td>$2^1$</td>
<td>2175</td>
<td>12</td>
<td>103</td>
</tr>
<tr>
<td>$2^2$</td>
<td>4403</td>
<td>15</td>
<td>151</td>
</tr>
<tr>
<td>$2^3$</td>
<td>8801</td>
<td>30</td>
<td>290</td>
</tr>
<tr>
<td>$2^4$</td>
<td>17664</td>
<td>60</td>
<td>333</td>
</tr>
<tr>
<td>$2^5$</td>
<td>35307</td>
<td>117</td>
<td>588</td>
</tr>
<tr>
<td>$2^6$</td>
<td>70549</td>
<td>230</td>
<td>1219</td>
</tr>
<tr>
<td>$2^7$</td>
<td>140956</td>
<td>455</td>
<td>2302</td>
</tr>
<tr>
<td>$2^8$</td>
<td>281985</td>
<td>934</td>
<td>5033</td>
</tr>
<tr>
<td>$2^9$</td>
<td>564070</td>
<td>1963</td>
<td>10212</td>
</tr>
<tr>
<td>$2^{10}$</td>
<td>1128299</td>
<td>4053</td>
<td>26446</td>
</tr>
</tbody>
</table>

Table A.10: Per-node round trip time results

A.5.2 Response communication time

<table>
<thead>
<tr>
<th>Candidate texts</th>
<th>Copies found</th>
<th>Min time (ms)</th>
<th>Max time (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>$2^0$</td>
<td>1092</td>
<td>5</td>
<td>24</td>
</tr>
<tr>
<td>$2^1$</td>
<td>2204</td>
<td>5</td>
<td>256</td>
</tr>
<tr>
<td>$2^2$</td>
<td>4410</td>
<td>5</td>
<td>259</td>
</tr>
<tr>
<td>$2^3$</td>
<td>8836</td>
<td>6</td>
<td>263</td>
</tr>
<tr>
<td>$2^4$</td>
<td>17676</td>
<td>8</td>
<td>658</td>
</tr>
<tr>
<td>$2^5$</td>
<td>35258</td>
<td>15</td>
<td>619</td>
</tr>
<tr>
<td>$2^6$</td>
<td>70439</td>
<td>20</td>
<td>910</td>
</tr>
<tr>
<td>$2^7$</td>
<td>141056</td>
<td>23</td>
<td>1800</td>
</tr>
<tr>
<td>$2^8$</td>
<td>282244</td>
<td>43</td>
<td>3665</td>
</tr>
<tr>
<td>$2^9$</td>
<td>563872</td>
<td>108</td>
<td>7662</td>
</tr>
<tr>
<td>$2^{10}$</td>
<td>1128351</td>
<td>257</td>
<td>19205</td>
</tr>
</tbody>
</table>

Table A.11: Per-node communication overhead time results
References


